Mail Exchange Protocol (MEP):
A draft for a new electronic mail protocol

by
C. C. Magnus Gustavsson

LITH-IDA-EX--04/066--SE

2004-06-04
Final thesis

Mail Exchange Protocol (MEP):
A draft for a new electronic mail protocol

by C. C. Magnus Gustavsson

LITH-IDA-EX--04/066--SE

Supervisor: Dr. Juha Takkinen
Dept. of Computer and Information Science
at Linköpings universitet

Examiner: Dr. Juha Takkinen
Dept. of Computer and Information Science
at Linköpings universitet
SMTP, the current protocol for sending electronic mail (e-mail) over the Internet, has for many years suffered from several problems and limitations. When it was designed, well over twenty years ago, the requirements for e-mail were very different from those of today. A message was a text message in English, and both user and machine were explicitly named in the address. The protocol was not designed to transfer other types of messages, and no mechanism was included to verify the identity of the sender.

In order to solve these shortcomings, a new e-mail protocol needs to be defined. This report specifies a basis for what such a protocol may look like. The protocol has been designed to be easy to modify and expand, as well as to benefit from more recent ideas and technology. Binary message content is transferred without conversion, sender addresses are verified, and the address format is flexible. Along with the specification of the protocol, a sample implementation has been provided.
Abstract

SMTP, the current protocol for sending electronic mail (e-mail) over the Internet, has for many years suffered from several problems and limitations. When it was designed, well over twenty years ago, the requirements for e-mail were very different from those of today. A message was a text message in English, and both user and machine were explicitly named in the address. The protocol was not designed to transfer other types of messages, and no mechanism was included to verify the identity of the sender.

In order to solve these shortcomings, a new e-mail protocol needs to be defined. This report specifies a basis for what such a protocol may look like. The protocol has been designed to be easy to modify and expand, as well as to benefit from more recent ideas and technology. Binary message content is transferred without conversion, sender addresses are verified, and the address format is flexible. Along with the specification of the protocol, a sample implementation has been provided.

Keywords: Electronic mail, protocol design, network protocol, Internet, SMTP, X.400
Acknowledgements

The author would like to thank the following people:

My parents and relatives – for much support and encouragement
Juha Takkinen – for supervising and examination
Anders Edqvist – for comments on the report
Contents

1 Introduction ................................................. 1
  1.1 Background ............................................. 1
  1.2 Problem ............................................... 2
  1.3 Protocol design goals ................................. 2
  1.4 Delimitations .......................................... 3
  1.5 Method ................................................. 3
  1.6 Report structure ....................................... 3

2 Current standards ............................................. 5
  2.1 Internet e-mail ......................................... 5
    2.1.1 SMTP structure ..................................... 6
    2.1.2 SMTP addresses ..................................... 6
    2.1.3 SMTP extensions .................................... 7
  2.2 X.400 .................................................... 9
    2.2.1 X.400 formats ....................................... 9
    2.2.2 X.400 addresses .................................... 9
    2.2.3 X.400 routing ....................................... 10

3 Protocol features ............................................. 11
  3.1 Single-part addresses .................................. 12
  3.2 Content types .......................................... 13
  3.3 Sender verification ..................................... 14
    3.3.1 Security considerations .............................. 15
  3.4 Time and date .......................................... 16
3.4.1 International standard .................................. 16
3.4.2 One-letter time zones ................................. 17

4 Protocol commands .............................................. 19

4.1 Introduction ..................................................... 19
4.1.1 Protocol version .......................................... 20
4.1.2 Message identity string ................................. 20
4.1.3 Opening a session ....................................... 20
4.2 INIT ............................................................... 21
4.3 QUIT ............................................................... 21
4.4 VERI mode ......................................................... 21
4.4.1 VERI INTE .................................................. 21
4.4.2 VERI FINA .................................................. 22
4.5 NOTI mode ....................................................... 22
4.5.1 NOTI READ ............................................... 22
4.5.2 NOTI EXPL ............................................... 22
4.6 SEND mode ..................................................... 22
4.6.1 SEND HEAD ............................................... 23
4.6.2 SEND PART ............................................... 23
4.7 Replies .......................................................... 23
4.7.1 OK .......................................................... 23
4.7.2 ACCE ...................................................... 23
4.7.3 ERRO ...................................................... 23
4.8 Examples ......................................................... 24
4.9 Syntax ........................................................... 25

5 Protocol header .................................................. 27

5.1 Informational fields ......................................... 27
5.2 Address fields ............................................... 28
5.3 Other fields ..................................................... 30
5.3.1 Reply ........................................................ 30
5.3.2 Track ....................................................... 31
5.4 Syntax ........................................................... 31
# Contents

## Implementation
- 6.1 Message storage ........................................ 33
- 6.2 Hash table entries ........................................ 34
- 6.3 IP sockets ............................................. 34
- 6.4 Mepcl .................................................. 34
- 6.5 Mepd .................................................. 36

## Conclusions and future work
- 7.1 Conclusions ................................................ 37
- 7.2 Future work ............................................... 38
  - 7.2.1 New modes ........................................... 38
  - 7.2.2 Security ............................................. 38
  - 7.2.3 Fallback to SMTP .................................... 39
- 7.3 Concluding remarks ....................................... 39

## Bibliography

## Appendix

### A Augmented Backus-Naur Form (ABNF)
- A.1 Rules .................................................. 45
- A.2 Terminals ............................................. 46
- A.3 Operators ............................................. 46

### B Source code
- B.1 Makefile ............................................. 50
- B.2 mep.h ................................................ 50
- B.3 mepcl.c ............................................. 51
- B.4 mepd.c ............................................. 60
- B.5 mx.c ................................................ 65
- B.6 head.c ............................................... 69
- B.7 send.c ............................................... 75
- B.8 unix.c ............................................. 79
Chapter 1

Introduction

1.1 Background

The current protocol for sending e-mail over the Internet (SMTP, Simple Mail Transfer Protocol [1]) is well over twenty years old and has for many years suffered from several problems and limitations. It was designed to transfer text messages in English only to a user at a specific machine, where both user and machine were explicitly named in the address; hence the at sign ("@") to separate the two parts [2].

The e-mail of today looks very different. For example, various kinds of content, such as images, audio or video, constitute a significant part of e-mail. The domain part (right side) of an address does not in itself reveal the name of the machine to which an e-mail will be delivered (much like telephones have number portability, so the subscriber may keep a number even when moving to another telephone station or service provider).

Many of the shortcomings of SMTP have got more or (often) less satisfactory solutions over the years. Others, such as verifying that the sender is who she claims to be, are not possible to solve simply by extending the existing protocol. Many of these problems had been possible or significantly easier to solve if the protocol had used a version number, for example of the kind used by more recent application protocols, such as HTTP (Hypertext
1.2 Problem

The main purpose of this thesis is to specify a basis for a protocol as discussed in the previous section. I will call it the Mail Exchange Protocol (MEP). Goals for this part are discussed in further detail below. To test the protocol and improve the quality of the specification, it needs to be implemented. Part of the task is thus to provide a working sample implementation as well.

The work is focused on defining a foundation for a protocol. It is not a goal to cover all parts which could have and eventually should be covered. Neither is it a goal to provide a user friendly interface. It is, however, a goal to provide enough functionality to produce something that could work as an advanced message transfer system.

1.3 Protocol design goals

The primary goal for the protocol design is robustness. By this is meant that the protocol must be as easy as possible to adapt to meet any future needs, whatever these may be. This issue is clearly a paramount one in the design.

Second, the protocol should be up to date, both by excluding obsolete features of SMTP and by including new features. It should make use of modern standards and technologies, whether these are Internet standards from IETF (Internet Engineering Task Force) [5] or others, such as those defined by ISO (International Organization for Standardization) [6].
Finally, the design should be complete. As an e-mail protocol, it should be able to send data to others as well as retrieve the data sent by others. In other words, it should cover all the steps of the process from sender to recipient.

1.4 Delimitations

To completely fulfill the completeness goal mentioned above, by including the full functionality of IMAP (Internet Message Access Protocol) [7], would be a too complex and time consuming task. Thus, I have limited myself to covering the functionality of SMTP only. The design has, however, been made with the intention of expanding it in this area later on.

1.5 Method

For the basic design I will try to keep an open mind. Where possible, I will try to use solutions which in my experience appear to be the best possible design choices, rather than sticking with the ways things have been done in the past. Since some of the solutions are new and untested, future experience might prove them to be bad ideas.

Along the same lines, I will strive to keep the design simple by adopting a concept from previous work only if it has a demonstrated use. To that, a number of new ideas will be added.

1.6 Report structure

The material in this document is organized in a chronological and bottom-up fashion. The most influential e-mail standards to date (SMTP and X.400 [8]) are reviewed in chapter 2. In chapter 3 a number of important features of the protocol are discussed. Then the protocol itself is defined, separated by first covering the protocol commands in chapter 4 and then the protocol header in chapter 5. Next, the implementation details are discussed in chapter 6, and finally, chapter 7 deals with conclusions and future improvements.
Due to the nature of this work it will not result in a single, easily stated conclusion. Neither do problems or motivations allow themselves to be briefly summarized here. Therefore, rather than attempting to isolate them, the problems are covered in their natural places in the next two chapters. When needed, motivations are given together with the design choices throughout the report. At the end, the work is discussed and evaluated.

The reader is expected to have a basic knowledge of UNIX, computer networking and e-mail.
Chapter 2

Current standards

This chapter details the state of global electronic mail and the previous major work in this area. The purpose is to gain an understanding of the problems with the current situation and the motivations for a new protocol.

2.1 Internet e-mail

In the early days of the Internet (in the seventies), network mail was transported using FTP (File Transfer Protocol) [2]. In time, a format for those messages was standardized, and finally evolved into RFC 822 [9]. Through MTP (Mail Transfer Protocol), the protocol part of the standard became SMTP (Simple Mail Transfer Protocol), and was defined in RFC 821 [1]. These two have since been obsoleted by updated versions published in 2001 [10, 11], but for nearly twenty years they were the most important standard documents for Internet e-mail.

SMTP (henceforth understood to include the message format definition in RFC 822) was created as a protocol to transfer text messages in English, or more precisely, the 7-bit ASCII character set, also known as ISO 646 [12]. The inability to transport anything else than ASCII has perhaps been the most important drawback of the standard. It is especially clear when attempting to create gateways to more capable systems, such as X.400.
(discussed later in this chapter). Extensions have been defined, which will be reviewed later on in this document, but today, more than two decades later, transporting English text is for the most part still all that SMTP does.

Another Internet standard is UUCP (UNIX-to-UNIX Copy) [13, 14], which has been used for mail delivery as well, commonly together with dial-up lines or other serial connections. Under UUCP, knowledge about exactly how hosts are connected is needed in order to send mail to another system. Recipients are specified by an explicit path for the transport of the message, such as host1!host2!user (called “bang paths”). This kind of e-mail is hardly used anymore.

2.1.1 SMTP structure

The message format of SMTP consists of a collection of header fields and the message body, separated by an empty line. The header fields are ASCII strings followed by a colon, whitespace and the content of the field. Common fields include “Date”, “Received”, “From”, “To”, and “Subject”.

A basic SMTP transaction is simple. Typically, a session is opened by the HELO/EHLO command, followed by specifying the effective sender (“reverse-path”) in the MAIL command and the recipient (“forward-path”) in the RCPT command. The last two commands specify the so-called “envelope”, which is what the mail server looks at for routing the mail. The envelope may be completely different from the “From” and “To” header fields in the message. Then follows the message, which is sent to the server using the DATA command. Finally, the session is terminated using QUIT and the mail is delivered.

2.1.2 SMTP addresses

An SMTP address is of the format user@domain, where domain is a DNS (Domain Name System) [15] domain. The domain part used to be a specific machine at which the user resided, but nowadays it is an abstraction. That is, it is an extra layer mapping between a mail domain, and one or more specific machines that handle mail for the domain in question. The data record in DNS for specifying this is called an MX (Mail eXchanger) record.
A more general variant of the MX record, the SRV record [16], also exists and may be used for other protocols than e-mail.

2.1.3 SMTP extensions

Support has been added to SMTP to transfer various kinds of data in addition to ordinary English text. One such extension is part of ESMTP (Extended SMTP) [17]. This extension allows text in 8-bit character sets to be transferred without conversion, but only if such a practice is negotiated and agreed upon by both parties in the communication. The 8-bit extension is not commonly used, however, and it is not enough for transferring content other than text.

To solve the problem with non-7-bit data over SMTP, a standard called MIME, Multipurpose Internet Mail Extensions [18], was introduced. The important MIME header fields are “Content-Type” and “Content-Transfer-Encoding”. Only the second one is relevant to the transfer problem. Thus, the first one will be ignored for now, but we return to it in 3.2.

Before MIME was defined, uuencode [19] (originally used with the above mentioned UUCP) was commonly used to encode binary or 8-bit text data for transport over 7-bit systems. As an example, assume that we want to transport the following text:


Encoded with uuencode, it would look like this:

```
begin 644 berling
M4VKV;B!H87(0<VEN82!KY&QL;W(9V%%N<VMA(&SE;F=T('5P<"!I&)YO<G(0M;V-H(&3D<B#D<B1!E="0%:1R;&EG="!L86YD(&;V<IE;%N(7-X;V=E M;B!I08V@85R9V55('5P<&CV<F$@86QD<FEG(&%T="!S86UL821V871T96X0 MY70@9&5N+B3='+V;6UA@0I08V@8N1C;V%R(7-T)G72!N9600;2!D96X0 ?Y7)E="!0;2XO1&5N(&AA<B1F:6X0=FET('7-A;F0N"0'
'',
end
```

Here, the 8-bit data has been split into segments of 6 bits and encoded as 64 different characters. The first character on each line specifies the length of the line.
In MIME, however, the common encodings in addition to “7bit” are “base64” and “quoted-printable”. All are 7-bit and thus compatible with the limitations of SMTP. The principle, to code 8-bit data into segments of 6 bits, is the same for base64 as for uuencoded data. The main differences are that the length of a line is not specified, and different characters are used to encode the 6-bit segments. Some of the characters in uuencode that are not letters or numbers are not consistent across all gateways. Because of this, base64 uses the English lower-case characters. The text in the above example would be encoded as follows in base64:

U2r2biBoYXIgc2luYSBr5Gxsbs1g22Fuc2thIGz1bmd0IHWwccBpIG5vncIgb2NoIGTkcIDkciB1
dHQga0RybG1ndCBSyWj5kIGb2cg1ibiBzavYuIFNrb2dibiBvY2gyMvyZ2VuIHwvcGj2cmEyYwkd

cm1nIGFodCBzYW1sYSB2YXROZW4g5XQgZGVuLiBTdHLL2biWhcgpvY2gyYuRja2PyIHNo9nJ0YSBu
ZWQgaSBkZW4g5XJldCBvbSB4gRGVuIGhhciBmaW4gdm10IHNhbmtcGg==

Data encoded into base64 uses about 1/3 more space compared to unencoded data. For data that is mostly English characters anyway (such as text in Swedish), this is a waste of space. It is for such data that “quoted-printable” (often referred to simply as “QP”) was meant. Characters that are part of the English alphabet are left as they are, and the other ones are encoded into an equal sign ("=") followed by the character code in hexadecimal. Our example would then look like this in quoted-printable:

Sj=F6n har sina k=E4llor ganska l=E5ngt upp i norr och d=E4r =E4r ett h=E4r ligt land f=F6r
en sj=F6. Skogen och bergen upph=F6ra aldrig att samla vatten =E5t den. Str=
=F6mmar
och b=E4ckar st=F6rta ned i den =E5ret om. Den har fin vit sand.

Data, such as programs, music, pictures, video, or Swedish text, is thus with few exceptions transferred as English text when sent by SMTP. The data has a limited character set and a maximum line length of 76 characters. All this is, at least in theory, hidden to the user. Transferring binary data in such a fashion is not only ugly, but also inefficient. A purely binary transport would allow us to transfer the same data with only 75% (6 bits coded as 8) of the space needed for base64, and with no conversion.
2.2 **X.400**

For X.400 [8], as for SMTP, there are two separate parts of the protocol: the format of the message itself, and the way that the message is transferred. We look at both in turn, starting with the format of the message.

### 2.2.1 X.400 formats

X.400 uses two different and incompatible formats for describing the content of a message: P2 (defined in the 1984 version of X.400 [20]) and P22 (from the 1988 standard [8]). One major difference between these formats is that P2 is limited to a number of predefined formats, while P22 allows external body-part definitions. Because of the incompatibility, the sending agent has to know whether the recipient understands P22, and if necessary downgrade the message to P2. The process of downgrading includes converting (with a minimum of information loss) to a format the recipient can understand.

Among the common body-part formats (i.e. character sets in this case) for X.400 are IA5 (International Alphabet 5, also known as ISO 646 [12] or ASCII, American Standard Code for Information Interchange), T.61 (ISO 6937 [21]), ISO Latin (ISO 8859 [22]) and ODIF (Open Document Interchange Format, also known as ISO 8613 [23] or T.411).

### 2.2.2 X.400 addresses

An X.400 address is a number of key-value pairs (fields) separated by semicolons. Extra spaces may be inserted for readability. For example, to send a message to the office of the Swedish prime minister, one would use:

S=Registrar; O=Primeminister; P=Ministry; A=SIL; C=SE

In the address above we have the fields S (surname, although in this particular case it is used to specify a function rather than a person), O (organization), P (private administrative domain or PRMD), A (public administrative domain or ADMD), and finally C (country according to ISO 3166 [24], and thus essentially the same as the ccTLDs, or country-code Top-Level Domains, used on the Internet). Other common fields not visible in this
example are G (given name), CN (common name, both the given name and the surname) and OU (organizational unit, for specifying the address with even more precision than with O alone).

X.400 addresses have traditionally been written by giving the top level first, i.e. the country first (the way the Britons used to write SMTP domains in the past). The order does not matter, however, and lately it has become common to give the top level last, likely because of the present Internet dominance. For example, the corresponding SMTP address for the above is registrator@primeminister.ministry.se.

We note that an X.400 address is dependent on the operator or service provider. That is, a business that moves to a different geographic location, or for some other reason uses another operator to connect to the global network, will need new addresses. On the Internet, this is the case for IPv6 [25] network addresses, but not for domains in the DNS and thus not for e-mail addresses. The operator dependence is clearly a disadvantage in an era where number portability for telephones is becoming increasingly common; people want to be able to keep the identifiers used to reach them, even when they change providers or move to a different area.

### 2.2.3 X.400 routing

The transfer of a message through an X.400 network is much more complex than in the SMTP equivalent. The transfer is based on routing domains, such as an administrative domain or a company. These domains can be further divided into areas, which share routing information within a domain. Much like on the Internet, the address of the recipient is examined and compared to a table in order to make a routing decision. If the address is local and thus known to the system, the message is delivered to the appropriate server. If not, it is routed to the level above, which might be the public administrative domain in the case of a company.

With SMTP, these kinds of routing functions are referred and restricted to lower network levels only. The SMTP server simply tries to connect to the host with the highest priority and leaves the routing issues to the IP protocol. With the high degree of connectivity in the Internet of today, there is seldom a need to deliver mail in more than one step, although the possibility to do so remains.
Chapter 3

Protocol features

The Mail Exchange Protocol (like SMTP and HTTP) is by design a somewhat “chatty” protocol, meaning that it uses more data than is necessary. The purpose of such a design is to make the protocol robust by being easy to expand or modify, as well as (to a much lesser degree) easy to implement. As mentioned in 1.3, robustness is the main design goal and significant effort is put into ensuring it.

The redundant data is, however, on the magnitude of a few bytes and more or less constant, as opposed to being dependent on message size. From a network load perspective, this is highly insignificant even for the comparatively small amounts of protocol data used by application level protocols. The design choice was acceptable at the time SMTP was defined and is even more so today.

It is of much greater importance to come to terms with the (nowadays common) millions or billions times larger amounts of redundant data that are the result of transporting message content in a non-binary form. This is one of three reasons why mail is considered unfit for transporting large amounts of data. The other two are the facts that a mailbox traditionally has been stored in a single file, among other things making it very time consuming to open a mailbox with large messages, and that the recipient has no control of the transferred data.

By introducing a new binary protocol and from scratch designing mail
systems to handle large amounts of binary data rather than just text, the first two of the problems above are easy to come to terms with. When the sender’s identity can be verified it is also more feasible to restrict the maximum message size a user is willing to accept on a sender address basis, within bounds set by system policy of course. This should allow users to use mail as a realistic and efficient means of transporting large files.

As a first step towards defining the Mail Exchange Protocol (MEP), a few of the concepts and constructs need to be looked at and discussed in some detail. In the remainder of this chapter I introduce and motivate single-part addresses, content types, sender verification, and finally, the time and date notation I will use.

3.1 Single-part addresses

I propose the use of “single-part addresses”, which are e-mail addresses without a separating at (“@”) sign. The at sign was originally introduced to separate the user name from the machine name in the e-mail address [2]. In the present days of MX records (see 2.1.2), the at sign has long since been outdated since the right side of the address is usually not the machine that accepts the mail anymore. In the same way, the left side of an address often does not represent an actual user account on the server anymore. By keeping the separating at sign we might give users a false impression that this is still true. A modern day e-mail address is really nothing more than a distributed identifier. The only reason to keep the at sign is that people are used to it; it is what an e-mail address is “supposed” to look like.

The address corresponding to the SMTP address adam@example.com would thus simply be adam.example.com in MEP. The address resolution process, however, gets a little more complicated because of this. In the example, the MX records for adam.example.com would be looked up. If such records are missing, records for example.com will instead be looked up. Finally, MX records for the top level domain (.com) will be checked for. On the Internet, such attempts will generally fail since top level domains seldom have MX records. If all lookups fail, the e-mail is rejected. If not, the message is delivered to the first one found, which we will call “best MX”. Lookup of MX records for the root is not implemented; I cannot foresee a
need to be able to use the "." address.

MEP will thus allow any two e-mail addresses to be handled by different servers. There is, for example, no reason for adam.example.com and carl.example.com to be handled by the same server. It would also be easy to configure a system so that carl.example.com is handled by one server, while other addresses under example.com are handled by another. Furthermore, only two MX records would be needed to do this.

Removing the at sign is not entirely without problems. By doing so, we remove a distinguishing character, and thus the address space becomes somewhat more limited in comparison. For example, while the addresses magnus.gustavsson@example.se and magnus@gustavsson.example.se are different in SMTP, if we replace the at sign by a dot they become identical. Furthermore, in SMTP the user part of an e-mail address is case dependent. Magnus.Gustavsson@example.se and magnus.gustavsson@example.se might thus reach two different users. To differentiate between the two in reality would generally be considered bad practice and most modern systems do not, but at least in theory this limits the address space even more. Both of these are, most likely, very unusual problems in practice. Yet, the mere possibility of ambiguity becomes a problem when attempting to implement a solution for automatic translation of addresses from one system to another. It does not mean that such translation is not possible, but policies for handling it must be made on a site-by-site basis.

In cases where it needs to be explicitly specified that a domain is an address and not simply a domain, the notation “x-mep:adam.example.com” (URL format, as defined by RFC 1738 [26]) should be used.

### 3.2 Content types

The concept of content types in their current form was introduced by MIME, which was discussed in 2.1.3. Content types are used for labeling message content and were originally defined in RFC 1341 [18], although work such as the “Mail Content Type” in RFC 1049 [27] had been previously published.

Content types have grown to be adopted by other protocols, most notably the HTTP protocol where they are called “Internet Media Types” [3].
Most of these differ from the MIME specification in that multipart body-
parts are not needed and thus not included. Since multipart body-parts
have no use in MEP either, we only make use of the types “text”, “application”, “image”, “audio”, and “video”, and not the SMTP-specific “multipart”
and “message”.

I furthermore believe the term “MIME type” has outlived itself by now.
Therefore, I will simply call this subset “content type” or, if it is necessary
to be even more precise, “Internet content type”.

### 3.3 Sender verification

In SMTP, the sender address of a message can be set to anything. This
allows for exploitation by impostors, spammers, viruses, and other kinds
of malicious persons or programs [28]. In my opinion, a user should always
be able to trust that the stated sender of a message actually sent it. This
should be ensured by any capable and modern e-mail protocol.

In MEP I introduce a “callback” feature to verify the identity of the
sender. It is similar to a modem with callback security, where the modem
breaks the connection and dials back on a predetermined telephone number
[13]. The feature will be defined in detail in 5.2, but the concept is briefly
outlined here. The idea is that whenever a message is sent, the unique
string that identifies the message (see 4.1.2) is stored in a hash table on a
server together with the “message digest”. The message digest works as a
fingerprint for the content of a message, not including the header. MD5,
defined in RFC 1321 [4], is used for this purpose.

We will call the server on which the hash entry is stored the “digest
server”, and it must be the same server that mail to the sender would be
delivered to. At every step along the distribution path of a message, the
receiving server then verifies the message by connecting to the digest server
the same way it would when delivering a message. Since only the recipient
of an address is able to update the hash table, we can ensure that the
sender of a message is who she claims to be.

Note that the digest server may be different from the machine that the
message originates from, which is possible as long as the proper hash entry
is stored on the digest server. However, this requires logging in on the
Sender verification is illustrated in figure 3.1. Suppose an e-mail is sent to adam.example.com. No MX exists for adam.example.com but the MX for example.com is mail.example.com. This is thus the best MX and the server to which the mail will be delivered. However, since Adam has moved his e-mail services to example.se, he has forwarded his e-mail to adam.example.se, for which the best MX is home.example.se. The sender is magnus.gustavsson.se, for which the best MX is gustavsson.se. When the server at mail.example.com receives the message (1) it looks up the best MX for the sender, connects to gustavsson.se (2) and attempts to verify the received message. If it checks out, the server accepts the message and connects to home.example.se (3). This server then connects to gustavsson.se (4) in a similar way before accepting the message from mail.example.com. As we will see in section 5.2, the receiving server does the verification in a slightly different way to tell the digest server that the message has reached the final destination and may be removed from the hash table.

![Figure 3.1: An example of sender verification.](image)

### 3.3.1 Security considerations

As a general reference for this section, see an introduction to cryptographic hash functions, such as [29].

Since the message digest (see 3.3) is of a finite length, it follows that
for each message digest there is an unlimited number of inputs that would produce the same result. If we get hold of a message identity (see 3.3) with a corresponding signature (message digest), for example by being the recipient of an e-mail, it would thus in theory be possible to create a false message with the same signature. To find such an input is commonly referred to as finding a second preimage.

This is exactly what algorithms such as MD5 have been designed to prevent. While it is on the order of $2^{64}$ operations to find a collision, that is, two inputs which produce the same signature, the difficulty of finding a second preimage for a given signature is on the order of $2^{128}$ operations. Finding a message that says something that we want it to say is even harder. Finally, all of this has to be accomplished within the short amount of time that the identity-digest pair is stored on the server.

It has been demonstrated that MD5 is not collision resistant [29], meaning that it is no longer computationally infeasible to find a collision. Because of the much greater difficulty of finding a second preimage, MD5 is, however, secure enough for use with MEP. Still, there might be other reasons to consider newer, stronger although slower algorithms, such as those of the RIPEMD and SHA families; for example, IETF might choose a specific algorithm for use with all Internet protocols.

3.4 Time and date

3.4.1 International standard

MEP uses the international standard for representation of dates and times, ISO 8601 [30], with minor extensions (for reasons explained below). Only the extended, complete format of the standard is used. It should also be noted that although the protocol uses a specific and exact notation, a client is free to present date and time to the user in any notation.

In the extended, complete format of ISO 8601, the notation for date is YYYY-MM-DD, where YYYY is the year in the Gregorian calendar, MM is the month (01–12) and DD is the day (01–31) of the month. A fixed number of digits (e.g. two for the month) is always used. The twenty-eighth day of March in the year 2004 is thus written as 2004-03-28.
According to ISO 8601, time is specified as hh:mm:ss, where hh is the hour in 24 hour notation (00–23), mm is the minute (00–59) and ss is the second (00–60, where 60 is a special and very rare case beyond the scope of this document). Midnight is always written as 00:00:00 (the beginning of the day). The point of time exactly three hours past midnight is written as 03:00:00, and three hours, thirty minutes and forty-five seconds past noon is written as 15:30:45.

Finally, time zones are written as +hh:mm or −hh:mm, where hh (as before) is hours and mm is minutes. These are offsets relative to Coordinated Universal Time (UTC). A positive zone is east of (or ahead of) UTC, while a negative is west of (or behind) UTC. UTC itself is written as +00:00. Time zones are always written directly after the time of the day, without space. Three hours past midnight in UTC is thus written as 03:00:00+00:00.

3.4.2 One-letter time zones

There are well over ten times as many time zone abbreviations, such as CET or EST, as there are time zones [31]. For example, BST, IST, WEST, CET, and WAT all mean +01:00. These are neither specified by any international standard, nor using any kind of common format. The same abbreviation is even in some cases used for different time zones. In the United States, time zone abbreviations end with −ST or, in the case of daylight saving time, −DT. In Europe the corresponding endings are −ET and −ST. For example, EEST (Eastern European Summer Time) is +03:00, while EST (Eastern Standard Time) is either −05:00 (e.g. Washington D.C.) or +10:00 (e.g. Canberra).

Due to the complexity and lack of standardization, the above system is not fit for use in technical applications. It should, however, be noted that the notation used in MEP is meant to be readable by humans. If it was not, then the string 20040328T030000+0200, another format specified in ISO 8601, might as well have been used. Another alternative would have been to use a single number, such as 1080435600. In POSIX [19] systems this represents the number of elapsed seconds since the point in time called the “Epoch”, which is 1970-01-01 00:00:00+00:00.

Therefore, I introduce an alternative notation as well. To denote a time
zone I use a single character in the English alphabet (ISO 646 [12]), a practice already in use since many years and commonly referred to as “U.S. military standard”. In the case of the letter Z, it is also defined in ISO 8601. This notation is not quite as flexible as the numerical notation, nor is it as easy to deduce the time difference between two zones. On the other hand, it is shorter, easier to remember and more manageable for humans, compared to the numerical notation. For example, the time zone two hours east of UTC could be referred to as “B time”.

The format of this notation is defined as follows: The time zone +00:00, i.e. UTC, is denoted by Z (as defined by ISO 8601). The zones +01:00 to +12:00 are denoted by A to M with the exception of J, which is reserved for local time (not used here). The zones −01:00 to −12:00 are denoted by N to Y. As with the numerical format, the time zone follows immediately, without spaces, after the time of the day. Three hours past midnight in UTC is thus written as 03:00:00Z.

Optionally, a plus sign may be appended to the letter to specify that the zone is adjusted half an hour to the east, that is, in advance of the corresponding full hour time zone. For example: “E+” means “+05:30”, while “Q+” means “−03:30”. Time zones that are not a full or half hour, and between −11:30 and +12:30 relative to UTC cannot be specified using this notation. Instead, the full notation of ISO 8601 has to be used.

In the combined (date and time) representation, the date and time are separated by a single space in MEP. For example: Because of the switch to daylight saving time, the second following 2004-03-28 01:59:59A in central Europe (e.g. Brussels) is 2004-03-28 03:00:00B. At this moment, the time in Washington D.C. is 2004-03-27 20:00:00R, while in Tokyo the time is 2004-03-28 10:00:00I and in New Delhi 2004-03-28 06:30:00E+. 
Chapter 4

Protocol commands

For the scope of the specification described here, a client is understood to be the software initiating a MEP session, while the server is the party that binds the port and accepts connections.

4.1 Introduction

The Internet Engineering Task Force (IETF) demands [32] that all new Internet protocols use UTF-8 [33], which is an encoding of the ISO 10646 [34] character set. It is also, in my opinion, an encoding which is very well suited to be used in a protocol. For these reasons, UTF-8 is the chosen encoding in MEP for commands as well as for the header. It is also the default encoding for the text type, if none is specified.

Only the four first letters in a command are significant, and a server must disregard any characters following between the first four and the next space. The exception is “INIT”, since it is used when the protocol version has not yet been agreed upon. All commands are case independent; “VERI” and “verify” are thus equivalent. For the specification described in this document, the former notation is used. Note that, in theory, commands are in UTF-8 and thus any server must be able to cope with that. In practice, however, defined commands are limited to the ASCII range as for
now and in the foreseeable future.

For protocol commands the concept “mode” is introduced as a means to
group a collection of commands for a specific purpose together. A command
may be either a simple command or a mode-specific command. In the
latter case the mode is given first. Modes may have mode-specific replies,
although such a practice should be avoided if possible and is therefore not
used here. Commands from different modes may be mixed, e.g. by giving
a VERI command between two SEND PART commands (see below).

For simpler implementation as well as for security reasons, the max-
imum length of a command or reply line is set to 2048 bytes. Characters
beyond this length may be discarded.

4.1.1 Protocol version

The protocol version is specified in the same format as used in HTTP [3].
It is indicated by the string “MEP/” followed by a major number, which
is an integer with any number of digits, then a dot (“.”) and finally a
minor number, which is also an integer with any number of digits. The
only current legal version is “MEP/0.9”.

4.1.2 Message identity string

The message identity is used to identify a message in the SEND HEAD,
VERI and NOTI commands (described below), as well as in the header
(the Reply field). There are no restrictions for what this string may look
like, except that it must be globally unique. Because of this, it is strongly
recommended that the hostname is included in the string.

4.1.3 Opening a session

The server must answer a connection request with “OK”, optionally followed
by an informational string such as hostname and software version. Since
the protocol version has not yet been agreed upon at this stage, the client
must use this optional string for informational and logging purposes only.
That is, the client may not malfunction or terminate the session because it
did not get the reply it expected.
4.2 INIT

INIT is used to initiate a session. It must be the first command given to the server. The purpose is to provide the client and server with a way to agree on the protocol version to be used. This allows for easy future revision and expansion of the protocol. The command is followed by the version, as specified in 4.1.1. A session following the current specification is thus always opened with “INIT MEP/0.9” and answered with an “OK” reply.

The INIT command is a special case in regard to length. While the rest is defined to be no more than 2048 bytes, the length of the INIT string is undefined. It may also be composed of another coding than UTF-8, or even of two- or four-byte characters. Fortunately, we only need to care about the first thirteen bytes. We check that the first twelve match the string “INIT MEP/0.9” in UTF-8 and that the next byte is not a number. If everything is all right, the server gives an “OK” reply and discards the rest of the line (if any). If not, it gives an “ERRO” reply. In the first case, we have agreed on the protocol version to use for the remainder of the session. Otherwise, the client may attempt to negotiate another version with the server or terminate the session.

4.3 QUIT

QUIT is used to terminate a session. Having received this command, the server must close the connection with an “OK” reply (see 4.7.1).

4.4 VERI mode

The VERI mode is used to verify the identity of the sender, as described in 3.3. It is followed by the message identity, the text “MD5” to identify the digest type and finally the MD5 sum of the message.

4.4.1 VERI INTE

VERI INTE is verification done by an intermediary server, meaning that the message has not yet reached its final destination.
4.4.2 VERI FINA

VERI FINA is verification of completed delivery. Given a correct verification, the digest server may thus remove the hash entry for the message permanently. By doing so, further delivery becomes impossible. Details about this are given in 3.3 and 5.2.

4.5 NOTI mode

NOTI is notification that the message has not only been delivered, but read as well. It comes in two forms, and must always be answered with “OK” by the server, regardless of whether such a message is known to the server or not. The server may choose to ignore the command and the recipient may also choose not to issue it at all. The reply informs the client that the command has been received correctly.

4.5.1 NOTI READ

NOTI READ means that the message has been read. It may be used when the message is opened, in contrast to merely reaching its destination. The command is followed by the identity string as specified in 4.1.2.

4.5.2 NOTI EXPL

NOTI EXPL means that the recipient has read the message and taken explicit action, such as clicking a button, to notify the sender of this fact. The command is followed by the message identity as well.

4.6 SEND mode

The SEND mode is used for message delivery. After receiving a correct SEND mode command, the server should issue an “ACCE” reply, accept the message and verify that everything went well by sending an “OK” reply.
4.6.1 SEND HEAD

SEND HEAD is followed by a message identity string, as specified in 4.1.2. It is used to send the header (see chapter 5), and it may only be used once while sending a message.

4.6.2 SEND PART

SEND PART is used for transferring a file (commonly text) as part of a message. It is followed by the size of the file, the content type as specified in 3.2 and optionally a file name.

4.7 Replies

There are three kinds of replies in the protocol: “OK”, “ERRO” and “ACCE”. Only these letters are significant. The remainder of the line may thus only be used for informational purposes, such as an explanatory message.

4.7.1 OK

OK is the equivalent of the SMTP 2xx reply codes [1], or Positive Completion, signaling that the command has been completed successfully.

4.7.2 ACCE

ACCE is the equivalent of the SMTP 3xx reply codes [1], or Positive Intermediate, and signals that further information is requested and expected. In the current specification it is only used by the SEND mode commands.

4.7.3 ERRO

ERRO signals that the server has been unable to complete the request. Thus, it is the equivalent of the SMTP 5xx reply codes [1], or Permanent Negative Completion.
4.8 Examples

The following is an example of a sending session:

\[
\begin{align*}
\Rightarrow & \text{ INIT MEP/0.9} \\
\Leftarrow & \text{ OK} \\
\Rightarrow & \text{ SEND HEAD 30030.1080435600.mail.example.com} \\
\Leftarrow & \text{ ACCE} \\
\Leftarrow & \text{ OK} \\
\Rightarrow & \text{ SEND PART 211 text/plain;charset=iso-8859-1 berling.txt} \\
\Leftarrow & \text{ ACCE} \\
\Leftarrow & \text{ OK 211 bytes} \\
\Rightarrow & \text{ SEND PART 564 image/png blackwhite.png} \\
\Leftarrow & \text{ ACCE} \\
\Leftarrow & \text{ OK 564 bytes} \\
\Rightarrow & \text{ SEND PART 4040290 video/quicktime gollum.mov} \\
\Leftarrow & \text{ ACCE} \\
\Leftarrow & \text{ OK 4040290 bytes} \\
\Rightarrow & \text{ SEND PART 5224760 application/postscript rfc2616.ps} \\
\Leftarrow & \text{ ACCE} \\
\Leftarrow & \text{ OK 5224760 bytes} \\
\Rightarrow & \text{ SEND PART 422279 text/plain;charset=utf-8 rfc2616.txt} \\
\Leftarrow & \text{ ACCE} \\
\Leftarrow & \text{ OK 422279 bytes} \\
\Rightarrow & \text{ QUIT} \\
\Leftarrow & \text{ OK}
\end{align*}
\]

The corresponding verification session is:

\[
\begin{align*}
\Rightarrow & \text{ INIT MEP/0.9} \\
\Leftarrow & \text{ OK} \\
\Rightarrow & \text{ VERI FINA 30030.1080435600.mail.example.com MD5} \\
& \text{c0afc110106a54f2e2dd232aa0161a5b} \\
\Leftarrow & \text{ OK} \\
\Rightarrow & \text{ QUIT} \\
\Leftarrow & \text{ OK}
\end{align*}
\]
4.9 Syntax

Augmented Backus-Naur Form (ABNF), as defined by RFC 2234 [35] and detailed in appendix A, is a common choice for syntax definitions in Internet specifications. The following is the complete command syntax of MEP in ABNF. The syntax for the replies is trivial and not included here.

\[
\begin{align*}
\text{command} & = ( \text{mode} / \text{single} ) \text{ CRLF} \\
\text{mode} & = \text{"SEND" } *\text{char SP send} / \\
& \phantom{=} \text{"VERI" } *\text{char SP veri} / \text{"NOTI" } *\text{char SP noti} \\
\text{single} & = \text{"INIT" SP "MEP/0.9" / "QUIT" } *\text{char} \\
\text{send} & = \text{"HEAD" } *\text{char SP token} / \\
& \phantom{=} \text{"PART" } *\text{char SP number SP ict [ SP token ]} \\
\text{veri} & = ( \text{"INTE" / "FINA" } ) *\text{char SP token SP digest} \\
\text{noti} & = ( \text{"READ" / "EXPL" } ) *\text{char SP token} \\
\text{number} & = 1*\text{DIGIT} \\
\text{ict} & = \text{token } "/" \text{ token [ } ";" \text{ token ]} \\
\text{digest} & = \text{"MD5" SP 32HEXDIG} \\
\text{token} & = 1*\text{char} \\
\text{char} & = \text{VCHAR / UTF8-2 / UTF8-3 / UTF8-4} \\
\end{align*}
\]

; From RFC 2234: Core Rules

\[
\begin{align*}
\text{CRLF} & = \text{CR LF } \quad \text{; Internet standard newline} \\
\text{DIGIT} & = \%x30-39 \quad \text{; 0-9}
\end{align*}
\]
HEXDIG = DIGIT / "A" / "B" / "C" / "D" / "E" / "F"
CR = %x0D  ; carriage return
LF = %x0A  ; linefeed
SP = %x20  ; space
VCHAR = %x21-7E  ; visible (printing) characters

; From RFC 3629: UTF-8

UTF8-char = UTF8-1 / UTF8-2 / UTF8-3 / UTF8-4
UTF8-1 = %x00-7F
UTF8-2 = %xC2-DF UTF8-tail
UTF8-3 = %xE0 %xA0-BF UTF8-tail / %xE1-EC 2UTF8-tail /
         %xED %x80-9F UTF8-tail / %xEE-EF 2UTF8-tail
UTF8-4 = %xF0 %x90-BF 2UTF8-tail / %xF1-F3 3UTF8-tail /
         %xF4 %x80-8F 2UTF8-tail
UTF8-tail = %x80-BF
Chapter 5

Protocol header

The protocol header consists of lines referred to as “fields”. Field names are case independent. The current specification defines ten different field types, which are classified as belonging to one of the categories “informational”, “address” or “other”. The header is terminated by an empty line.

Note that even though DNS puts restrictions on what a domain name may look like, the entire header should be considered to be UTF-8 [33]. Data should be passed unmodified to an underlying system, such as a DNS resolver [15], as long as doing so is possible without getting unmanageable errors. There is one important exception to this rule: Any at sign (“@”) occurring in an address should be treated as if it were a dot (“.”).

5.1 Informational fields

The informational fields consist of “Subject”, “Sender” and “Recipient”. They contain data that is of interest to the user but not to the system. There must be exactly one of each field. If there is not, all but one may be discarded. The informational fields may include an arbitrary UTF-8 string.

Examples:

Subject: Hello
Sender: Adam B. Example
5.2. Address fields

The address fields are the fields that specify the different types of addresses needed for the letter. They are “To”, “Copy”, “From”, “Deliver”, and “Verify”. Each contain a single-part address as detailed in 3.1.

“To” and “Copy” do the same thing from the system’s point of view. The only difference between them is informational, i.e. of interest only to the user. Both specify the addresses that the sender meant the letter to reach and they remain the same during distribution. Each of these fields may be duplicated an unlimited number of times in the header. “Copy” is optional and can be left out altogether.

There must be exactly one of each of the “From”, “Deliver” and “Verify” fields. The “From” field specifies the original sender of a message. The other two are important in message routing and are covered in detail below.

Examples (the fields for the example in 3.3):

To: adam.example.com
Copy: carl.example.com
From: magnus.gustavsson.se
Deliver: adam.example.se
Verify: magnus.gustavsson.se

The topic of message routing was partially covered in 3.3. Routing involves two fields of interest: “Deliver” and “Verify”. “Deliver” roughly correspond
Protocol header

29

to the “forward-path” and “Verify” to the “reverse-path” of the SMTP protocol, together commonly called the “envelope” (see 2.1.1). They may also be thought of as the effective versions of the “To”/“Copy” and “From” fields, respectively.

“Deliver” specifies the current recipient for the message. It can be different from the recipient specified by the sender in the “To” field. In the example in 3.3, the “To” field contains adam.example.com, while “Deliver” contains adam.example.se. If there is only one recipient, “Deliver” has the same content as the “To” field when the message is sent. The content of “Deliver” is also used for the “Track” field (see below).

The “Verify” field specifies the address of the server that is responsible for sending the message, and for which the sender verification discussed in 3.3 is performed. This server is in other words the digest server. For a normal (one-to-one) e-mail message the “Verify” field remains the same as the “From” field during the entire distribution. In figure 5.1, the digest server is node A during distribution from A to B, as well as from B to C. Delivery is not completed (see 4.4) until the message reaches node C. Thus, C uses VERI FINA to verify the message, while B uses VERI INTE.

![Figure 5.1: Node A remains the digest server.](image)

The typical case where the “Verify” and “From” fields do differ is the mailing list, in which case the list becomes the effective sender. In figure 5.2, the digest server is node A during distribution from A to B, and node B during distribution from B to C. Delivery is completed after each session,
i.e. at node B and C. A message sent from magnus.gustavsson.se to mep-discuss.example.com might thus have magnus.gustavsson.se in the “From” field and mep-discuss.example.com in “Verify”.

![Diagram](image)

Figure 5.2: Node B takes over as digest server.

Multiple recipients will result in more than one VERI FINA command. Although the server could ignore them all and simply let the hash entry (see 3.3) time out, the recommended method is to generate a new message identity for every address on the list.

### 5.3 Other fields

#### 5.3.1 Reply

The “Reply” field contains a message identity string and may be used to specify what message the present one is a reply to. Among other things, this is useful for presenting a threaded view of an e-mail discussion in a client. It corresponds to “In-Reply-To” in SMTP. The “Reply” field is optional but must not be included more than once.

Example:

Reply: 30030.1080435600.mail.example.com
5.3.2 Track

The “Track” field is used to track the servers that a message reaches during the distribution from sender to recipient. It consists of a time stamp as specified in 3.4, followed by the host name of the server and by the effective recipient (as specified by the “Deliver” field). The “Track” field corresponds to “Received” in SMTP. The header may contain any number of “Track” fields. They must always be in chronological order, with the first server’s “Track” field being the first one in the header, and subsequent ones being added after the others.

Example:

Track: 2004-03-28 03:00:00B mail.example.com adam.example.com

5.4 Syntax

The following is the complete header syntax in Augmented Backus-Naur Form (ABNF), as defined by RFC 2234 [35] and detailed in appendix A. See 4.9 for Core Rules and UTF-8.

```
field = ( info / address / reply / track ) newline
newline = CR / LF / CRLF
info = "Subject:" SP text /
      "Sender:" SP text / "Recipient:" SP text
address = "From:" SP token / "To:" SP token /
          "Copy:" SP token /
          "Verify:" SP token / "Deliver:" SP token
reply = "Reply:" SP token
track = "Track:" SP date SP time SP token SP token
date = year "-" month "-" day
```
year = 4DIGIT
month = %x30 %x31-39 / %x31 %x30-32
day = %x30 %x31-39 / %x31-32 DIGIT / %x33 %x30-31

time = hour "::" minute "::" second zone
hour = %x30-31 DIGIT / %x32 %x30-33
minute = %x30-35 DIGIT
second = %x30-35 DIGIT / %x36 %x30
zone = %x41-49 / %x4B-5A
text = *( char / SP )
token = 1*char

char = VCHAR / UTF8-2 / UTF8-3 / UTF8-4
Chapter 6

Implementation

The implementation has been provided primarily to demonstrate, evaluate and improve the quality of the protocol specification (see 1.2). It may, however, also be used as a basis for future extensions. The implementation consists of a command line interface (mepcl) and a daemon (mepd). The code has been tested to my best effort, and hopefully most of the serious bugs have been eliminated. However, so far little concern has been given to security issues. The complete code is available in appendix B.

6.1 Message storage

To begin with, a few things about how the daemon mepd stores received messages should be mentioned. The path to the mail spool is compiled into both mepd and the command line interface mepcl. In this directory there is a directory for every message. The message directories have eight-character names (e.g. “hv9hg000”), of which the first six characters are derived from the current time, and the last two are a sequence number. The representation has been chosen so that the sorting of the names will order the messages in chronological order (“hvmws000” is a newer message than “hv9hg000”).

In the message directory, each part of the message (see 4.6.2) is stored as
a separate file, each beginning with a number and followed by an underscore (“_”) and optionally a file name. The header always has the number 0. The underscore is included only as a means to allow the software to make correct interpretations of whether a number is part of a file name or not, such as in the case of “12be.ps” and “12be.ps”. The header of a specific message might thus be stored as “/mep/spool/hv9hg000/0_” while the first part is stored as “/mep/spool/hv9hg000/12be.ps”.

6.2 Hash table entries

For the sender verification (see 3.3) to work there is a need to provide a method to insert new entries into the hash table. Since no such method is defined as part of the protocol, a simple implementation-specific solution is used. Specifically, mepd uses a UNIX socket named /tmp/.mep_socket to accept new entries.

No time stamps are stored in the hash table as of now, so entries cannot be removed based on time. The only way to remove an entry from the table is by a VERI command (see 4.4). Unless such a command is given, the entry will remain indefinitely.

6.3 IP sockets

I have chosen to use Internet Protocol version 6 (RFC 2460 [25]) sockets for the implementation. The only reason for this is that IPv6 compared to the currently dominant version 4 is, in my opinion, much more elegant and capable. Hopefully it represents the future of the Internet. Since this in an example implementation, and portability therefore is of little concern, the implementation requires IPv6 support. If need be, it should be relatively easy to rewrite the code to use IPv4 sockets as well.

6.4 Mepcl

Mepcl is the command line interface used for sending and retrieving messages. It is entirely command-line based and basically has three functions:
send messages, list messages and extract messages. It must be compiled to use the same mail spool as mepd for the system to work.

Usage is as follows:

mepcl [address file ... | message]

If the command is used with no arguments the content of the spool directory is listed. If the first argument begins with a hash mark (“#”), it is interpreted to be a message number and the specified message is extracted into the current directory. Otherwise, the first argument is assumed to be an address to deliver a message to. In this case it also takes the name of at least one file to send to the recipient.

Examples:

Send all the files in the current directory:

% mepcl adam.example.com *
Sending "berling.txt"... OK
Sending "blackwhite.png"... OK
Sending "gollum.mov"... OK
Sending "rfc2616.ps"... OK
Sending "rfc2616.txt"... OK
%

List the content of the spool directory:

% mepcl
Message 1. hv9hg000 5 parts
%

Extract the message parts into the current directory:

% mepcl #1
Extracting 1_berling.txt... OK
Extracting 2_blackwhite.png... OK
Extracting 3_gollum.mov... OK
Extracting 4_rfc2616.ps... OK
Extracting 5_rfc2616.txt... OK
%
6.5  Mepd

Mepd is a simple daemon which listens on port 60030 (as of now), accepts incoming connections and stores the messages on file (as described in 6.1) to be displayed and extracted by mepcl. Mepd does not accept any kinds of arguments or flags.
Chapter 7

Conclusions and future work

7.1 Conclusions

I have designed and implemented MEP, the Mail Exchange Protocol. The improved flexibility and performance detailed in chapter 3 has been achieved and the functionality I had planned to include has been implemented. In this report I have introduced the new concept of single-part addresses (see 3.1). The idea of sender verification (see 3.3) is not new, but to my knowledge it has never before been used to verify the identity of the sender in the way described in this work.

The goals as presented in 1.3 have been achieved and, in the wider perspective, the result has turned out as well as I could have expected. The protocol has been designed to be easy to expand. Together with the use of a version number, as detailed in 4.2, robustness is ensured. The features described in chapter 3 have been included, bringing the protocol up to date with modern requirements. Within the delimitations stated in 1.4, the design is fully functional and complete, as demonstrated by the implementation.
7.2 Future work

There are a number of issues that can be improved and expanded in this protocol specification. I will outline a few such things here, but leave out the obvious need for a more user-friendly implementation. However, a mature application would have to include such things as a built-in editor and a graphical user interface.

7.2.1 New modes

The protocol was from the beginning meant to eventually become something more than the equivalent of SMTP. Specifically, it was meant to include at least the functionality of IMAP (Internet Message Access Protocol) [7] or POP (Post Office Protocol) [36]. As mentioned in 1.4, this has been beyond the scope of this document.

Adding such functionality is what the modes are meant for. Something very similar to IMAP could be defined as a new mode which we might call "RETR" (from "retrieve"). It would be something like a subprotocol, and might be handled by a separate program on the server. The mode could even have a set of mode-specific reply codes, if there is a need for it.

As an alternative approach, IMAP could be modified to be used with MEP. A single password could be remembered by the user agent to be used with both protocols. However, implementation becomes more complicated and there is a need to authenticate twice. Furthermore, if two separate daemons are used, configuration parameters, such as mail box format, system paths and user database, need to be duplicated.

7.2.2 Security

The ability to negotiate an encrypted session over TLS/SSL (Transport Layer Security [37] and the older Secure Sockets Layer specification) is required for any modern protocol standard to be complete. Because of the design of MEP, there is no need to use a separate port to handle encrypted sessions.
7.2.3 Fallback to SMTP

To increase the level of availability in a mixed environment of MEP and SMTP, fallback to SMTP might be attempted. A MEP server may only make such an attempt when delivery by MEP is not possible at any level in the DNS hierarchy. Doing otherwise would break the functionality of MEP as specified in this document. The existence of SMTP servers should not have to be taken into account when configuring servers for MEP.

To deliver by SMTP, the characters up to the first dot in the address, which is called the first "label", should be left out and a best MX should be found for the remaining part. If an SMTP server is found, the at sign is simply inserted at the appropriate place and the message delivered as usual. If the recipient is not found on the SMTP server, another label may be removed and a best MX for the remaining part (if any) may be searched for.

7.3 Concluding remarks

Once I got started on the programming and report writing I have changed very few things. The syntax for the protocol commands has been altered slightly and some of the header fields have got new names. In the beginning I envisioned a somewhat more configurable sample implementation, but later that seemed unnecessary, given the rudimentary state so far.

The material presented in this report has been on my mind for several years, and during that time it has evolved from vague thoughts to clear concepts. This work has been a chance to summarize my ideas of how e-mail should work, get it down on paper and make it available to the world. Knowledge gained through the project consists mainly of a much better understanding of X.400.

To try to replace the existing e-mail protocols may indeed appear to be a very ambitious project. No matter what happens, at least I got to complete my degree and it was fun to do. Still, I would not have done it without hope that some part of it (or something similar) would come to use eventually. E-mail has always been my favourite Internet application. It deserves to be realized by something better than SMTP.
7.3. Concluding remarks
Bibliography


Appendix A

Augmented Backus-Naur Form (ABNF)

This appendix specifies a subset of the Augmented Backus-Naur Form (ABNF) notation [35] used in chapters 4 and 5 of this thesis.

A.1 Rules

A rule is defined by the sequence name = elements, ended by a newline. name is the name of the rule and elements is one or more rule names or terminal specifications. The equal sign separates the name from the definition of the rule. The elements form a sequence of one or more rule names and/or value definitions, combined according to the various operators, defined in section A.3. When a rule requires multiple lines, the continuation lines are indented.

The name of a rule is a sequence of characters, beginning with an letter, and followed by a combination of letters, digits and hyphens (“-”). Rule names are case-insensitive. Unlike original Backus-Naur Form [38], angle brackets (“<”, “>”), or other delimiters, are not used.
A.2 Terminals

Rules resolve into a string of terminal values, also called characters. In ABNF a character is merely a non-negative integer. Terminals are specified by one or more digits with the base interpretation of those characters indicated explicitly. We use only the hexadecimal base, represented by “\x”. For example, the rule \texttt{CR} = %x0D specify the hexadecimal representation of UTF-8 [33] for carriage return.

ABNF permits specifying a literal text string enclosed in quotation-marks. Hence \texttt{command = "command string"}. Literal text strings are interpreted as a concatenated set of printable characters. Strings are case-insensitive and the character set for these strings is ASCII [12]. Hence, \texttt{rulename = "MEP"} matches “mep”, “Mep”, “mEp”, “MEp”, “mEP”, “MeP” and “MEP”. To specify a rule which is case-sensitive, the characters need to be specified individually. For example: \texttt{rulename = %x4D %x45 %x50} will match only the string which comprises the upper-case characters “MEP”.

A.3 Operators

Presented in order of precedence, from lowest (binding loosest) to highest (binding tightest), the operators are:

- **Alternatives** \texttt{Rule1 / Rule2}
  - Elements separated by slash (“/”) are alternatives. Therefore, \texttt{adam / carl} will accept \texttt{adam} or \texttt{carl}.

- **Concatenation** \texttt{Rule1 Rule2}
  - A sequence of elements defines a concatenation of contiguous characters. For example, in this case the rule \texttt{CRLF} matches the Internet standard newline:
    
    \begin{align*}
    \texttt{CRLF} &= \texttt{CR LF} \\
    \texttt{CR} &= \texttt{%x0D} \quad ; \text{carriage return} \\
    \texttt{LF} &= \texttt{%x0A} \quad ; \text{linefeed}
    \end{align*}
Sequence group

Elements enclosed in parentheses are treated as a single element, whose contents are strictly ordered.

Since concatenation has higher operator precedence than alternatives, the sequence emma adam / carl sara matches emma adam or carl sara. The sequence emma (adam / carl) sara, on the other hand, matches emma adam sara or emma carl sara.

Variable repetition

The operator “*” preceding an element indicates repetition. The full form is <l>*<m>element, where <l> and <m> are optional decimal values, indicating at least <l> and at most <m> occurrences of element.

Default values are 0 and infinity, so that *<element> allows any number, including zero; 1*<element> requires at least one; 3*3<element> allows exactly 3 and 1*2<element> allows one or two.

Specific repetition

A rule of the form <n>element is equivalent to <n>*<n>element; that is, exactly <n> occurrences of element. Thus, 2DIGIT is a 2-digit number.

Optional sequence

Square brackets enclose an optional element sequence. Thus, [adam carl] is equivalent to *1(adam carl).

Value range alternatives

A range of alternative numeric values can be specified compactly, using hyphen (“-”) to indicate the range of values. Hence, DIGIT = %x30-39 is equivalent to DIGIT = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9".
Comment ;

A semicolon starts a comment that continues to the end of line. This is a simple way of including useful notes in parallel with the specifications.
Appendix B

Source code

This appendix contains the Makefile and the complete source code in C, consisting of the following files:

mep.h    Common header files, prototypes and settings
mepcl.c  Command to send, list and retrieve messages
mepd.c   Listen on the port and handle the connections
mx.c     Find the best MX and establish a connection
head.c   Read, store and retrieve data from the header
send.c   Handle data transfer and sender verification
unix.c   Manage the hash table and the UNIX socket

The code has been tested on various Linux systems with the GNU and Intel compilers. It is available on file at the following address:

http://magnus.gustavsson.se/mep/thesis-source.tar.gz

Note that the files global.h, md5.h and md5c.c in RFC 1321 [4] are required for MD5_CTX, MD5Init(), MD5Update(), and MD5Final(). They are not included in this appendix, but are available in the above tar archive.
B.1 Makefile

CC=gcc -O3
RM=rm

WARNFLAGS=-Wall

SOBJECT=md5c.o mx.o head.o send.o unix.o mepd.o
COBJECT=md5c.o mx.o mepcl.o
RESOLVER=/usr/lib/libresolv.a

all: mepd mepcl

.c.o:
    $(CC) $(WARNFLAGS) -c <$

mepd: $(SOBJECT)
    $(CC) -Wl,-s -o $@ $(SOBJECT) $(RESOLVER) -lpthread

mepcl: $(COBJECT)
    $(CC) -Wl,-s -o $@ $(COBJECT) $(RESOLVER) -lmagic

clean:
    $(RM) -f *.o mepd mepcl core

B.2 mep.h

#include <sys/types.h>
#include <sys/socket.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "global.h"
#include "md5.h"

/* Hardcoded sender address */
```c
#define SENDER "magnus.gustavsson.se"

#define CHUNK_SIZE 10000
#define MAX_CONNECTIONS 16
#define PORT "60030"
#define SPOOL_PATH "/tmp/spool"
#define UNIX_SOCKET "/tmp/.mep_socket"

typedef struct tsd {
    MD5_CTX context;
    int desc, send;
} tsd_t;

extern char *getstring(int, const int);
extern char *strip(char *, char);
extern char *getverify(void);
extern void scanheader(int);
extern void writeheader(int);
extern void clearheader(void);
extern void terminate(void);
extern void unixsocket(void);
extern int getmxsocket(char *);
extern int mep_init(char *);
extern void mep_notify(char *);
extern void mep_quit(char *);
extern void mep_send(char *);
extern void mep_verify(char *);
extern void sendout(char *);

B.3 mepcl.c

#include <sys/un.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
#include <netdb.h>
#include <dirent.h>
#include <iconv.h>
```
#include <locale.h>
#include <langinfo.h>
#include <magic.h>
#include <libgen.h>
#include "mep.h"

int sock, error;

char *getid(char *host)
{
    char *id;

    id = (char *) malloc((20 + strlen(host)) * sizeof(char));
    sprintf(id, "%u.%u.%s", (int) getpid(), (int) time(NULL), host);

    return id;
}

char *getzone(long timezone, int isdst)
{
    char *zone = (char *) malloc(3 * sizeof(char));
    char *zones = "YXWVUTSRQPONZABCDEFGHIKLM??";
    int halfhours = 24 - timezone / 1800;

    zone[0] = zones[halfhours / 2 + (isdst > 0)];
    zone[1] = halfhours % 2 ? '+' : '\0';
    zone[2] = '\0';

    return zone;
}

char *gettime(void)
{
    char timestring[22], *zone;
    struct tm *local;
    time_t t;

    time(&t);
    local = localtime(&t);
zone = getzone(timezone, local-&gt;tm_isdst);

sprintf(timestring, "%04d-%02d-%02d %02d:%02d:%02d%s",
        1900 + local-&gt;tm_year, 1 + local-&gt;tm_mon, local-&gt;tm_mday,
        local-&gt;tm_hour, local-&gt;tm_min, local-&gt;tm_sec, zone);

free(zone);

return strdup(timestring);
}

void md5send(char *id, char *md5)
{
    char string[NI_MAXHOST + 53];
    int sock;
    struct sockaddr_un addr;

    addr.sun_family = AF_UNIX;
    strcpy(addr.sun_path, UNIX_SOCKET);

    if ((sock = socket(PF_UNIX, SOCK_STREAM, 0)) &lt; 0) {
        fprintf(stderr, "Unable to get a socket: %s\n",
                strerror(errno));
        exit(EXIT_FAILURE);
    }

    if (connect(sock, (struct sockaddr *) &amp;addr,
                sizeof(struct sockaddr_un)) &lt; 0) {
        fprintf(stderr, "Unable to connect: %s\n",
                strerror(errno));
        exit(EXIT_FAILURE);
    }

    sprintf(string, "%s %s", id, md5);
    send(sock, string, strlen(string), 0);

    close(sock);
}
int parts(const struct dirent *list)
{
    char first;

    return (first = *(list->d_name)) > 0x30 && first < 0x40;
}

void freelist(struct dirent **namelist, int n)
{
    while (n--)
        free(namelist[n]);
    free(namelist);
}

void extract(int msg)
{
    char dir[PATH_MAX + 1], *temp;
    char from[PATH_MAX + 1], to[PATH_MAX + 1];
    struct dirent **namelist, **namelist2;
    int i, n, n2;

    if ((n = scandir(SPOOL_PATH, &namelist, 0, alphasort)) >= 0) {
        sprintf(dir, "%s/%s", SPOOL_PATH, namelist[msg + 1]->d_name);
        if ((n2 = scandir(dir, &namelist2, parts, alphasort)) >= 0) {
            for (i = 0; i < n2; i++) {
                printf("Extracting ... ", namelist2[i]->d_name);

                if ((temp = strchr(namelist2[i]->d_name, '_')) != NULL) {
                    sprintf(from, "%s/%s", dir, namelist2[i]->d_name);

                    if (strlen(++temp) != 0)
                        sprintf(to, "./%s", temp);
                    else
                        sprintf(to, "./message%d", i);

                    if (rename(from, to) == 0)
                        printf("OK");
                }
            }
        }
    }
}
void listspool(void)
{
    char dir[PATH_MAX + 1];
    struct dirent **namelist, **namelist2;
    int i, n, n2;

    if ((n = scandir(SPOOL_PATH, &namelist, 0, alphasort)) >= 0) {
        for (i = 0; i < n; i++) {
            if (strlen(namelist[i]->d_name) == 8) {
                printf("Message %d.\t%s", i - 1, namelist[i]->d_name);
                sprintf(dir, "%s/%s", SPOOL_PATH, namelist[i]->d_name);
                if ((n2 = scandir(dir, &namelist2, parts, alphasort)) >= 0)
                    printf("\t\t%d parts\n", n2);
            }
        }
    }
    freelist(namelist2, n2);
}
    freelist(namelist, n);
}

void check_reply(char *reply, char *message)
{ char buffer[CHUNK_SIZE];
    buffer[recv(sock, buffer, CHUNK_SIZE, 0)] = '\0';
    if (strncmp(buffer, reply, strlen(reply)) != 0) {
        fprintf(stderr, message);
        close(sock);
        exit(EXIT_FAILURE);
    }
}

char *utf8_recode(char *string) {
    char buffer[PATH_MAX + 1];
    char *outbuf = buffer;
    size_t insize, outsize = PATH_MAX + 1;
    iconv_t desc;
    insize = strlen(string);
    setlocale(LC_CTYPE, "");
    if ((desc = iconv_open("UTF-8", nl_langinfo(CODESET))) < 0) {
        fprintf(stderr, "Unable to get a conversion descriptor: %s\n",
                strerror(errno));
        exit(EXIT_FAILURE);
    }
    iconv(desc, &string, &insize, &outbuf, &outsize);
    iconv_close(desc);
    *outbuf = '\0';
    return strdup(buffer);
}

char *mepify_mime(const char *string) {

char *new, *mime, *charset;

new = strdup(string);
mime = strsep(&new, "\t,;");

if (new && (charset = strstr(new, "charset=")) != NULL) {
  if (strcmp(charset + 8, "us-ascii") == 0)
    strcpy(charset + 8, "utf-8");

    strcat(mime, ";");
    strcat(mime, charset);
}

if(strcmp(mime, "") == 0) {
  free(mime);
  mime = strdup("application/octet-stream");
}

return mime;
}

int main(int argc, char *argv[]) {

  unsigned char digest[16];
  char md5[32], buffer[CHUNK_SIZE];
  char host[NI_MAXHOST], *id;
  const char *magic;
  magic_t cookie;
  struct stat st;
  MD5_CTX context;
  int i, file, bytes;

  if (argc < 2) {
    listspool();
    exit(EXIT_SUCCESS);
  }

  if (argv[1][0] == '#') {
    extract(atoi(argv[1] + 1));
  }
exit(EXIT_SUCCESS);
}

if ((sock = getmxsocket(argv[1])) == 0) {
    fprintf(stderr, "Unable to connect to server\n");
    exit(EXIT_FAILURE);
}

check_reply("", ");

if (gethostname(host, NI_MAXHOST) < 0)
    sprintf(host, "[unknown]\")

sprintf(buffer, "INIT MEP/0.9\n");
send(sock, buffer, strlen(buffer), 0);

check_reply("OK", "Incorrect reply from server\n");

sprintf(buffer, "SEND HEAD %s\n", id = getid(host));
send(sock, buffer, strlen(buffer), 0);

check_reply("ACCE", "Message refused by server\n");

sprintf(buffer, "Recipient: \nSender: \nSubject: \n");
send(sock, buffer, strlen(buffer), 0);

sprintf(buffer, "To: %s\nFrom: %s\n", argv[1], SENDER);
send(sock, buffer, strlen(buffer), 0);

sprintf(buffer, "Deliver: %s\nVerify: %s\n", argv[1], SENDER);
send(sock, buffer, strlen(buffer), 0);

sprintf(buffer, "Track: %s %s %s\n", gettime(), host, argv[1]);
send(sock, buffer, strlen(buffer), 0);

check_reply("OK", "Message refused by server\n");

MD5Init(&context);

for (i = 2; i < argc; i++) {
if (stat(argv[i], &st) < 0) {
    fprintf(stderr, "Unable to stat file %s: %s\n",
            argv[i], strerror(errno));
    exit(EXIT_FAILURE);
}

if (!S_ISREG(st.st_mode)) {
    fprintf(stderr, "%s is not a regular file\n", argv[i]);
    exit(EXIT_FAILURE);
}

if ((cookie = magic_open(MAGIC_MIME)) == NULL ||
    magic_load(cookie, NULL) == -1 ||
    (magic = magic_file(cookie, argv[i])) == NULL) {
    fprintf(stderr, "%s\n", magic_error(cookie));
    exit(EXIT_FAILURE);
}

sprintf(buffer, "SEND PART %d %s %s\r\n", (int) st.st_size,
        mepify_mime(magic), utf8_recode(basename(argv[i])));
send(sock, buffer, strlen(buffer), 0);

magic_close(cookie);

check_reply("ACCE", "Message refused by server\n");

fprintf(stderr, "Sending \"%s\"... ", argv[i]);

if ((file = open(argv[i], O_RDONLY)) < 0) {
    fprintf(stderr, "Unable to open file %s: %s\n",
            argv[i], strerror(errno));
    exit(EXIT_FAILURE);
}

while ((bytes = read(file, buffer, CHUNK_SIZE)) != 0) {
    send(sock, buffer, bytes, 0);
    MD5Update(&context, buffer, bytes);
}
```c
#include <ctype.h>
#include <netdb.h>
#include <pthread.h>
#include "mep.h"

char host[NI_MAXHOST];
FILE *logfile = NULL;
pthread_key_t key;

void logging_on(void)
{
    if ((logfile = fopen("sessionlog", "w")) == NULL) {
        fprintf(stderr, "Unable to open log file\n");
        exit(EXIT_FAILURE);
    }

    MD5Final(digest, &context);
    for (i = 0; i < 16; i++)
        sprintf(md5 + 2 * i, "\%02x", digest[i]);

    md5send(id, md5);
    free(id);

    sprintf(buffer, "QUIT\r\n");
    send(sock, buffer, strlen(buffer), 0);

    check_reply("OK", "Incorrect reply from server\n");

    return EXIT_SUCCESS;
}
```
void logging_off(void)
{
    if (logfile != NULL) {
        fclose(logfile);
        logfile = NULL;
    }
}

void log_in(char *string)
{
    fprintf(logfile, "\=> %s", string);
}

void log_out(char *string)
{
    fprintf(logfile, "\<= %s", string);
}

void terminate(void)
{
    pthread_key_delete(key);
    logging_off();
    unlink(UNIX_SOCKET);
    exit(EXIT_SUCCESS);
}

void sendout(char *string)
{
    send(((tsd_t *) pthread_getspecific(key))->desc,
         string, strlen(string), 0);
    log_out(string);
}

/* We accept commands of any length for now... */
char *getstring(int desc, const int crlfonly)
char *string = NULL, *temp;
int read = -1, size = 0;

while (read != 0) {
    string = (char *) realloc(string, (size + 21) * sizeof(char));
    temp = string + size;
    size += (read = recv(desc, temp, 20, 0)) > 0 ? read : 0;
    temp[read] = '\0';

    if ((crlfonly == 0 && read == 0 && size > 0) ||
        strncmp(string + size - 2, "\r\n", 2) == 0)
        return string;
}

return NULL;

int mep_init(char *string)
{
    if (strncasecmp(string, "MEP/0.9", 7) == 0) {
        sendout("OK\r\n");
        return 0;
    } else {
        sendout("ERRO\r\n");
        close(((tsd_t *) pthread_getspecific(key))->desc);
        return 1;
    }
}

void mep_notify(char *string)
{
    sendout("OK\r\n");
}

void netsocket(int desc)
{
char *string;
tsd_t *tsd;

string = (char *) malloc((20 + strlen(host)) * sizeof(char));
sprintf(string, "OK MEP/0.9 %s ready\r\n", host);
send(desc, string, strlen(string), 0);
free(string);

tsd = (tsd_t *) malloc(sizeof(tsd_t));
pthread_setspecific(key, tsd);
ts->desc = desc;

while (((string = getstring(desc, 1)) != NULL) {
    log_in(string);

    switch (tolower(string[0])) {
    case 'i':
        if (strncmp(string, "INIT", 4) == 0) {
            if (mep_init(strip(string, ' ')) != 0)
                return;
        }
    case 'n':
        if (strncmp(string, "NOTI", 4) == 0) {
            mep_notify(strip(string, ' '));
            break;
        }
    case 'q':
        if (strncmp(string, "QUIT", 4) == 0) {
            mep_quit(strip(string, ' '));
            return;
        }
    case 's':
        if (strncmp(string, "SEND", 4) == 0) {
            mep_send(strip(string, ' '));
            break;
        }
    case 'v':
        if (strncmp(string, "VERI", 4) == 0) {

mep_verify(strip(string, ' '));
break;
}
default:
    sendout("ERRO\r\n");
}

free(string);
}
}

void mepd(void)
{
    pthread_t tid;
    int desc, sock;
    struct sockaddr_in6 addr;
    socklen_t size = sizeof(struct sockaddr_in6);

    logging_on();

    pthread_key_create(&key, NULL);
    pthread_create(&tid, NULL, (void *(*)(void *)) &unixsocket, NULL);

    addr.sin6_family = AF_INET6;
    addr.sin6_port = htons(atoi(PORT));
    addr.sin6_addr = in6addr_any;

    if (gethostname(host, NI_MAXHOST) < 0)
        sprintf(host, "[unknown]\n");

    if ((sock = socket(PF_INET6, SOCK_STREAM, 0)) < 0) {
        fprintf(stderr, "Unable to get a socket: %s\n",
                strerror(errno));
        exit(EXIT_FAILURE);
    }

    if (bind(sock, (struct sockaddr *) &addr, size) < 0) {
        fprintf(stderr, "Unable to bind socket: %s\n",
                strerror(errno));
    }
```c
exit(EXIT_FAILURE);
}

if (listen(sock, MAX_CONNECTIONS) < 0) {
    fprintf(stderr, "Unable to listen on socket: %s
", strerror(errno));
    exit(EXIT_FAILURE);
}

while (1) {
    if ((desc = accept(sock, (struct sockaddr *) &addr, &size)) < 0) {
        fprintf(stderr, "Unable to accept connections: %s
", strerror(errno));
        exit(EXIT_FAILURE);
    }
    pthread_create(&tid, NULL, (void *(*)(void *)) &netsocket,
                   (void *) desc);
}

int main(void)
{
    if (fork() == 0)
        mepd();

    return EXIT_SUCCESS;
}

B.5 mx.c

#include <netdb.h>
#include <resolv.h>
#include "mep.h"

typedef struct mxlist {
    int pref;
```
char *host;
struct mxlist *next;
} mxlist_t;

void prefinsert(mxlist_t **list, mxlist_t *element)
{
    while (*list != NULL && (*list)->pref < element->pref)
        list = &(*list)->next;

    element->next = *list;
    *list = element;
}

void atsignreplace(char *string)
{
    while (*string != '\0') {
        if (*string == '@')
            *string = '.';

        string++;
    }
}

char *strip(char *string, char delim)
{
    while (*string != '\0') {
        string++;

        if (*string == delim) {
            string++;
            break;
        }
    }

    return string;
}

cchar *getprefmx(const char *domain)
{

static mxlist_t *reply = NULL;
static const char *previous;
u_char nsreply[NS_PACKETSZ];
mxlist_t *temp;
ns_msg handle;
ns_rr record;
int len, rnum;

if (reply != NULL) {
    temp = reply;
    reply = temp->next;
    free(temp->host);
    free(temp);
}

if (previous != domain) {
    previous = domain;
    while (reply != NULL) {
        temp = reply;
        reply = temp->next;
        free(temp->host);
        free(temp);
    }
}

if ((len = res_search(domain, ns_c_in, ns_t_mx,
    nsreply, NS_PACKETSZ)) < 0)
    return NULL;

if (ns_initparse(nsreply, len, &handle) < 0) {
    fprintf(stderr, "ns_initparse: %s\n", strerror(errno));
    return NULL;
}

for (rnum = 0; rnum < ns_msg_count(handle, ns_s_an); rnum++) {
    if (ns_parserr(&handle, ns_s_an, rnum, &record) < 0) {
        fprintf(stderr, "ns_parserr: %s\n", strerror(errno));
        return NULL;
    }
}
temp = (mxlist_t *) malloc(sizeof(mxlist_t));
temp->pref = ns_get16(ns_rr_rdata(record));
temp->host = (char *) malloc(MAXDNAME * sizeof(char));

if (dn_expand(ns_msg_base(handle), ns_msg_end(handle),
              ns_rr_rdata(record) + 2, temp->host, MAXDNAME) < 0) {
    fprintf(stderr, "dn_expand: %s\n", strerror(errno));
    return NULL;
}

prefinsert(&reply, temp);
}
}

if (reply == NULL)
    return NULL;

return reply->host;

int getmxsocket(char *domain)
{
    char *server;
    int error, sock;
    struct addrinfo hints, *addr;

    if (domain == NULL)
        return 0;

    atsignreplace(domain);

    while (*domain != '\0') {
        if ((server = getprefmx(domain)) != NULL) {
            memset(&hints, 0, sizeof(hints));
            hints.ai_family = AF_INET6;
            hints.ai_socktype = SOCK_STREAM;

            if ((error = getaddrinfo(server, PORT, &hints, &addr)) != 0) {
                }
fprintf(stderr, "Unable to resolv address: %s\n",
    strerror(errno));
exit(EXIT_FAILURE);
}

if ((sock = socket(PF_INET6, SOCK_STREAM, 0)) < 0) {
    fprintf(stderr, "Unable to get a socket: %s\n",
        strerror(errno));
    exit(EXIT_FAILURE);
}

if (connect(sock, addr->ai_addr, sizeof(struct sockaddr_in6))
    < 0) {
    fprintf(stderr, "Unable to connect: %s\n",
        strerror(errno));
    exit(EXIT_FAILURE);
}

return sock;
}

domain = strip(domain, '.');
}

return 0;
}

B.6 head.c

#include <ctype.h>
#include "mep.h"

typedef struct field {
    char *string;
    struct field *next;
} field_t;

typedef struct fields {
typedef struct header {
    char *hd_subject;
    char *hd_sender;
    char *hd_recipient;
    char *hd_from;
    fields_t hd_to;
    fields_t hd_copy;
    char *hd_verify;
    char *hd_deliver;
    char *hd_reply;
    fields_t hd_track;
} header_t;

header_t head = {
    NULL, NULL, NULL, NULL, {NULL, NULL},
    {NULL, NULL}, NULL, NULL, NULL, {NULL, NULL}
};

char *getverify(void)
{
    return head.hd_verify;
}

cchar *line(char *name, char *string)
{
    char *temp;
    int len;

    if (strncmp(name, string, len = strlen(name)) != 0)
        return NULL;

    for (string += len; *string == ' '; string++)
        if ((temp = strstr(string, "\r\n")) != NULL)
            *temp = '\0';
return strdup(string);
}

void addstring(char *name, char *string, char **fields)
{
    if (*fields != NULL)
        free(*fields);

    *fields = line(name, string);
}

void addfield(char *name, char *string, fields_t *fields)
{
    field_t *temp;

    if ((string = line(name, string)) != NULL) {
        temp = (field_t *) malloc(sizeof(field_t));
        temp->string = string;
        temp->next = NULL;

        if (fields->last == NULL)
            fields->first = fields->last = temp;
        else
            fields->last = fields->last->next = temp;
    }
}

void scan(char *string)
{
    switch (tolower(string[0])) {
    case 'c':
        addfield("Copy: ", string, &head.hd_copy);
        break;
    case 'd':
        addstring("Deliver: ", string, &head.hd_deliver);
        break;
    case 'f':
        addstring("From: ", string, &head.hd_from);
        break;
    }
break;
case 'r':
    switch (tolower(string[2])) {
        case 'c':
            addstring("Recipient: ", string, &head.hd_recipient);
            break;
        case 'p':
            addstring("Reply: ", string, &head.hd_reply);
            break;
    }
    break;
case 's':
    switch (tolower(string[1])) {
        case 'e':
            addstring("Sender: ", string, &head.hd_sender);
            break;
        case 'u':
            addstring("Subject: ", string, &head.hd_subject);
            break;
    }
    break;
case 't':
    switch (tolower(string[1])) {
        case 'o':
            addfield("To: ", string, &head.hd_to);
            break;
        case 'r':
            addfield("Track: ", string, &head.hd_track);
    }
    break;
case 'v':
    addstring("Verify: ", string, &head.hd_verify);
    break;
}

void scanheader(int desc)
{
    char *field = NULL, *string = NULL, *crlf;

int len = 0;

while (len != 2) {
  while (string == NULL ||
        (crlf = strstr(string, "\r\n")) == NULL)
    string = getstring(desc, 1);

  len = crlf - string + 2;
  field = (char *) realloc(field, (len + 1) * sizeof(char));
  strncpy(field, string, len);
  field[len] = '\0';
  string += len;
  scan(field);
}

free(field);

void putstring(int file, char *name, char *string)
{
  if (string != NULL) {
    write(file, name, strlen(name));
    write(file, string, strlen(string));
    write(file, "\n", 1);
  }
}

void putfields(int file, char *name, fields_t fields)
{
  field_t *temp;

  for (temp = fields.first; temp != NULL; temp = temp->next) {
    write(file, name, strlen(name));
    write(file, temp->string, strlen(temp->string));
    write(file, "\n", 1);
  }
}
void writeheader(int file)
{
    putstring(file, "Subject: ", head.hd_subject);
    putstring(file, "Sender: ", head.hd_sender);
    putstring(file, "Recipient: ", head.hd_recipient);
    putstring(file, "From: ", head.hd_from);
    putfields(file, "To: ", head.hd_to);
    putfields(file, "Copy: ", head.hd_copy);
    putstring(file, "Verify: ", head.hd_verify);
    putstring(file, "Deliver: ", head.hd_deliver);
    putstring(file, "Reply: ", head.hd_reply);
    putfields(file, "Track: ", head.hd_track);
}

void clearstring(char **string)
{
    free(*string);

    *string = NULL;
}

void clearfields(fields_t *fields)
{
    field_t *next, *temp;

    for (temp = fields->first; temp != NULL; temp = next) {
        next = temp->next;
        free(temp->string);
        free(temp);
    }

    fields->first = fields->last = NULL;
}

void clearheader(void)
{
    clearstring(&head.hd_subject);
    clearstring(&head.hd_sender);
    clearstring(&head.hd_recipient);
clearstring(&head.hd_from);
clearfields(&head.hd_to);
clearfields(&head.hd_copy);
clearstring(&head.hd_verify);
clearstring(&head.hd_deliver);
clearstring(&head.hd_reply);
clearfields(&head.hd_track);
}

B.7 send.c

#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
#include <pthread.h>
#include "mep.h"

#define FLAGS O_CREAT|O_EXCL|O_WRONLY|O_TRUNC
#define MODE S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH /* 0644 */
#define DMODE MODE|S_IXUSR|S_IXGRP|S_IXOTH /* 0755 */

#define pchar(x) (i = x % 36) > 9 ? 0x57 + i : 0x30 + i

extern pthread_key_t key;
char *id;

char *dirname(int time)
{
    static int count = 0, oldtime = 0;
    int i, pos = 9;
    char string[pos];

    if (oldtime != time) {
        count = 0;
        oldtime = time;
    }

    string[--pos] = '\0';
}
B.7. send.c

```c
string[--pos] = pchar(count);
string[--pos] = pchar(count++ / 36);

while (pos > 0) {
  string[--pos] = pchar(time);
  time /= 36;
}

return strdup(string);
}

void verify(tsd_t *tsd)
{
  unsigned char digest[16];
  char md5[32], *buffer;
  int i, sock;

  MD5Final(digest, &tsd->context);
  for (i = 0; i < 16; i++)
    sprintf(md5 + 2 * i, "%02x", digest[i]);

  buffer = (char *) malloc((50 + strlen(id)) * sizeof(char));

  if ((sock = getmxsocket(getverify())) != 0) {
    sprintf(buffer, "INIT MEP/0.9\r\n");
    send(sock, buffer, strlen(buffer), 0);
    free(getstring(sock, 1));

    sprintf(buffer, "VERI FINA %s MD5 %s\r\n", id, md5);
    send(sock, buffer, strlen(buffer), 0);
    free(getstring(sock, 1));

    sprintf(buffer, "QUIT\r\n");
    send(sock, buffer, strlen(buffer), 0);
    free(getstring(sock, 1));
  }
}
```
close(sock);
}

free(buffer);
free(id);
tsd->send = 0;
clearheader();
}

void mep_quit(char *string)
{
    tsd_t *tsd;
    
    tsd = (tsd_t *) pthread_getspecific(key);
    
    if (tsd->send == 1)
        verify(tsd);
        sendout("OK\r\n");
        close(tsd->desc);
    }

void mep_send(char *string)
{
    static int num;
    char path[PATH_MAX + 1], filename[PATH_MAX + 1];
    char buffer[CHUNK_SIZE], *temp;
    int file, declared, bytes, btotal;
    time_t t;
    tsd_t *tsd;
    
    tsd = (tsd_t *) pthread_getspecific(key);
    
    if (strncasecmp(string, "HEAD", 4) == 0) {
        if (tsd->send == 1)
            verify(tsd);
            string += 4;
while (*(++string) == ' ');

if (((temp = strstr(string, "\r\n")) != NULL)
    *temp = '\0';)

id = strdup(string);

num = 1;
tsd->send = 1;
MD5Init(&tsd->context);

time(&t);
sprintf(path, "%s/%s", SPOOL_PATH, temp = dirname(t));
mkdir(path, DMODE);
free(temp);

sendout("ACCE\r\n");

scanheader(tsd->desc);

snprintf(filename, PATH_MAX, "%s/0_", path);
if ((file = open(filename, FLAGS, MODE)) < 0)
    return;

writeheader(file);
close(file);

sendout("OK\r\n");

return;
}

if (tsd->send == 1 && strncasecmp(string, "PART", 4) == 0 &&
    (declared = atoi(string = strip(string, ' '))) != 0) {

btotal = 0;

if (((temp = strstr(string, "\r\n")) != NULL)
    *temp = '\0';)

string = strip(strip(string, ' '), ' ');

snprintf(filename, PATH_MAX, "%s/%d_%s", path, num++, string);
sendout("ACCE\r\n");
if ((file = open(filename, FLAGS, MODE)) < 0)
    return;
while ((bytes = recv(tsd->desc, buffer, CHUNK_SIZE, 0)) != 0) {
    write(file, buffer, bytes);
    MD5Update(&tsd->context, buffer, bytes);
    if (((btotal += bytes) >= declared)
        break;
}
close(file);
if (btotal == declared) {
    sprintf(buffer, "OK %d bytes\r\n", btotal);
    sendout(buffer);
} else {
    sprintf(buffer, "ERRO read %d bytes\r\n", btotal);
    sendout(buffer);
}
return;

sendout("ERRO\r\n");

B.8  unix.c

#include <sys/un.h>
#include <pthread.h>
#include "mep.h"

typedef struct md5list {
    char *id, *md5;
struct md5list *next;
} md5list_t;

md5list_t *md5hash[10] = {
    NULL, NULL, NULL, NULL, NULL,
    NULL, NULL, NULL, NULL, NULL
};

pthread_mutex_t hashmutex = PTHREAD_MUTEX_INITIALIZER;

void hash_put(char *id, char *md5)
{
    md5list_t *temp;
    int total = 0;
    char *ch;

    for (ch = id; *ch != '\0'; ch++)
        total += (int) *ch;
    total %= 10;

    temp = (md5list_t *) malloc(sizeof(md5list_t));
    temp->id = id;
    temp->md5 = md5;
    temp->next = md5hash[total];
    md5hash[total] = temp;
}

char *hash_get(char *id)
{
    md5list_t **p, *temp;
    int total = 0;
    char *ch;

    for (ch = id; *ch != '\0'; ch++)
        total += (int) *ch;

    for (p = &md5hash[total % 10]; *p != NULL; p = &(*p)->next)
        if (strcmp(id, (*p)->id) == 0) {

        }
free((*p)->id);
ch = (*p)->md5;
temp = *p;
*p = (*p)->next;
free(temp);

return ch;
}

return NULL;
}

void mep_verify(char *string)
{
    char *client_hash, *server_hash, *temp;

    string = strip(string, ' ');

    if ((client_hash = strchr(string, ' ')) != NULL) {
        *(client_hash++) = '\0';

        if (strncasecmp(client_hash, "MD5 ", 4) == 0) {
            client_hash += 3;
            while (*(++client_hash) == ' ');

            if ((temp = strstr(client_hash, "\r\n")) != NULL)
                *temp = '\0';

            pthread_mutex_lock(&hashmutex);

            server_hash = hash_get(string);

            pthread_mutex_unlock(&hashmutex);

            if (server_hash != NULL &&
                strcmp(client_hash, server_hash) == 0) {
                sendout("OK\r\n");
                return;
            }
        }
    }
}
```c
void unixsocket(void)
{
    char *hash, *string;
    int desc, sock;
    struct sockaddr_un addr;
    socklen_t size = sizeof(struct sockaddr_un);

    addr.sun_family = AF_UNIX;
    strcpy(addr.sun_path, UNIX_SOCKET);

    if ((sock = socket(PF_UNIX, SOCK_STREAM, 0)) < 0) {
        fprintf(stderr, "Unable to get a socket: %s\n",
                strerror(errno));
        exit(EXIT_FAILURE);
    }

    if (bind(sock, (struct sockaddr *) &addr, size) < 0) {
        fprintf(stderr, "Unable to bind socket: %s\n",
                strerror(errno));
        exit(EXIT_FAILURE);
    }

    if (listen(sock, MAX_CONNECTIONS) < 0) {
        fprintf(stderr, "Unable to listen on socket: %s\n",
                strerror(errno));
        exit(EXIT_FAILURE);
    }

    while (1) {
        if (((desc = accept(sock, (struct sockaddr *) &addr, &size))
             < 0) {
            fprintf(stderr, "Unable to accept connections: %s\n",
                    strerror(errno));
        }
    }
}
```
exit(EXIT_FAILURE);
}

if ((string = getstring(desc, 0)) != NULL) {
    if (strncasecmp(string, "QUIT", 4) == 0) {
        close(sock);
        terminate();
    }

    if ((hash = strchr(string, ' ')) != NULL) {
        *(hash++) = '\0';
        pthread_mutex_lock(&hashmutex);
        hash_put(strdup(string), strdup(hash));
        pthread_mutex_unlock(&hashmutex);
    }
    free(string);
}

close(desc);
}
Copyright

Svenska

Detta dokument hålls tillgängligt på Internet - eller dess framtidiga ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera åktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

English

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© C. C. Magnus Gustavsson
Linköping, 6th January 2005