Emm: An Electronic Mail Management System for UNIX

Chester Ramey

Information Network Services
and
Department of Computer Engineering and Science
Case Western Reserve University
Cleveland, OH 44106

ABSTRACT

This document describes the design and implementation of **emm**, a new electronic mail management system for the UNIX† operating system, and discusses the requirements and goals which inspired it. The emm system provides a powerful set of tools for sending and reading electronic mail.

The emm system includes a number of new features for handling incoming and outgoing mail; these features are discussed briefly. A series of examples highlighting features of the emm system is presented, and short comparisons of emm to some existing UNIX mail systems are included. The paper concludes with a summary of emm's strengths and weaknesses and a discussion of possible future work.

1. Introduction

Electronic mail (*email*) is quickly becoming a common part of the world of computer networks. Nearly everyone using a computer today has used email in one form or another. Unfortunately, most of the tools being used on UNIX systems to manage this flood of electronic correspondence are not up to the task. These tools are not as easy to use as they should be, nor do they scale well from users who receive and process a few messages a day to those who process thousands of messages a week. In fact, as is the case with most complex application domains, those email management systems which attempt to cater to novice users are generally unsuitable for sophisticated users, and vice versa.

† UNIX is a trademark of Bell Laboratories.

emm is an electronic mail management system which attempts to solve these problems. It is sufficiently full-featured to satisfy experienced email users, and may still be used to introduce new UNIX users to electronic mail ‘‘painlessly’’. It is suitable for storing, retrieving, and managing thousands of pieces of electronic mail. It allows users to read and send mail easily and quickly. In addition, it contains a number of novel features previously unseen in UNIX email management tools or found only in experimental systems which improve user interaction and mail organization.

Though **emm** contains much that is new, what it does not contain is equally important. There are certain features whose functionality might be desirable that were not included for reasons of compatibility or complexity. For example, while it is interesting to consider an email system which is fully integrated with a database system such as Ingres, such an implementation would be incompatible with other UNIX tools, negating one of the primary advantages of the UNIX environment! Performance and resources would also become an issue -- imagine having to install Ingres on every system simply to read email.

emm provides a robust, unified interface to electronic correspondence that fits smoothly into the UNIX environment.

2. Background

Electronic mail systems have existed for many years. At its base, email is computer–based and managed asynchronous communication between users (originators and recipients) of the same or different machines. Email is organized into discrete units called *messages*, which are the basic components of communication.

2.1. The Office Model

Email began as the electronic evolution of the office memorandum, and email systems bear a strong resemblence to the conventional office desktop. Early email systems were developed to replace the common mechanisms of office communication: phone calls, letters, and face–to–face meetings. These early systems were primarily meant for ‘‘personal’’ messages between individuals. While this use persists to this day, other more ambitious applications are being found for electronic mail, and the tools used to manage it must keep pace.

Electronic mail management systems are quite similar to the office desktop. Such systems commonly provide:

- an **inbox**, to keep incoming messages in the order of their arrival
- an **outbox**, containing outgoing messages
- a **notepad** for message composition
- a private message **filing area**
- a **personal profile** to tailor the system to individual tastes

While this model is a good beginning, there are some obvious dissimilarities. In both evironments, for instance, the user has a wastebasket for messages which he intends to dispose of. Unlike the office wastebasket, from which messages may be retrieved, uncrumpled, and dealt with again, most UNIX email systems do not offer that sort of undeletion. In the standard office environment, the outbox contains messages which have not yet been collected and sent. In the electronic environment, on the other hand, the system normally dispatches messages immediately, so the outbox functions as a very temporary holding area. Instead of this sort of holding area, electronic mail outboxes can be used for messages being *tracked*, using a return receipt facility. However, receipt tracking is another feature not normally found in UNIX mail systems.

2.2. Email Advantages

Electronic mail offers a number of advantages over ‘‘traditional’’ communication mechanisms. These include:

- permanent records of communication
- allowing some freedom from interruption and an opportunity for reflection while remaining timely
- allowing better use of the sender’s and receiver’s time
- solving the problem of both sender and receiver interacting synchronously, which is occasionally inconvenient

Most of these advantages are apparent when the sender and receiver are on different, widely−separated systems, but the benefits apply to local email systems as well.

2.3. Mail System Organization

Electronic mail systems are generally organized into three levels, similar to the ISO Reference Model of computer networks. At the highest level, closest to the user, are *user agents*. These are the programs with which the users interact to send and retrieve their mail. A user agent provides facilities to *view, save, discard, file* or *archive, compose,* and *send* messages. It is the portion most closely corresponding to the office desktop.

At the next level, a message is transferred from its originator to the recipient’s computer by a *message transfer system*, consisting of one or more *message transfer agents*, each of which may reside on a separate machine. These agents utilize whatever physical network exists between the machines to effect message transfer. If the sender and receiver are on the same machine, the message transfer agent simply passes the message to the bottom level.

At the lowest level resides the *message delivery agent*, which transfers the message to a repository owned by the recipient. This repository provides storage of messages that have been delivered to their final destination but not yet retrieved by the recipient's user agent. The repository is usually one of a set of system mailboxes.

On some systems, the message delivery agent's duties are performed by the message transfer agent. On many versions of UNIX, by contrast, *sendmail* performs message transfer and programs like */bin/mail* or *m2mbox* perform actual message delivery. These message transfer and message delivery agents provide a store−and−forward path between the originator's and the recipient's user agents.

2.4. Terms

A number of terms are used to describe electronic mail systems. These include:

message

The base unit of electronic mail communication. It consists of a set of headers and a body.

headers

A set of named fields, each with a specific semantic meaning. A header consists of the header name (such as *From*), a colon, and the header contents (e.g., *chet@po.cwru.edu*). Headers are *metadata*: in addition to being part of the message, they specify properties of the message such as sender and recipient.

body

The collection of text and other data that contains the message being transmitted between sender and recipient.

folder

A collection of messages, usually related, kept in the same file.

draft

A message that is in the process of being composed and sent.

A message is similar in structure to an interoffice memo. There is header information denoting the sender, recipient, and subject of the message, then the body of the memo. Some email systems alter messages as they present them to their users to make the resemblence more obvious (and, presumably, more comforting).

2.5. Email Standards

The Internet community has defined a number of standards concerning email. These standards cover everything from the format of mail headers to the transmission of mail from one machine to another. Internet standards documents are called *Requests For Comments* (RFCs).

2.5.1. RFC 821 [1]

This defines the Simple Mail Transfer Protocol (**SMTP**) for transferring email messages between Internet hosts. It specifies a set of commands and responses used by processes to transfer a message to a named set of recipients and report on the transfer's success or failure. It is based on a model of *peer-to-peer* communication between message transfer agents on different machines using a two-way transmission channel. SMTP says nothing about the format of the messages transferred, neither headers nor body, beyond the requirement that a message be transferred in units of lines of ASCII text, with eight-bit characters converted to seven bits by clearing the high-order bit. The lines are not permitted to exceed 1000 characters.

2.5.2. RFC 822 [2]

This RFC describes the format of email message headers. It specifies what can appear in a header field, which headers are required to exist, how headers are separated from the message body, and how headers are to be recognized by sending and receiving message transfer and user agents. It specifies nothing about the body of a message other than that it contains a sequence of lines of text.

RFC 822 does not attempt to define a standard mechanism or internal format for storing messages after delivery, nor does it specify any details concerning user agents used to read the messages. It is intended only as a specification of a format for use when passing messages between machines. In practice, however, nearly all email systems which use the Internet generate RFC 822 headers for outgoing messages by default, preserve the headers when storing messages, and present them unmodified to the user when displaying messages. This practice is useful when gatewaying mail between different mail environments, and is standard in the Internet environment.

2.5.3. RFC 1341 [3]

It has become apparent in the years since RFC 821 and RFC 822 were written that they were too restrictive. In particular, the RFC 821 specifications concerning legal character values and line lengths inhibited sending messages in languages and character sets other than those used in the United States, and limited messages to text. RFC 1341 redefines the format of message bodies to allow the representation of multi−part textual and non−text message bodies, and standardizes mechanisms by which these bodies may be exchanged without loss of information. This format is known as **MIME** (Multipurpose Internet Mail Extensions).

MIME allows a message to contain multiple objects, text with unlimited line length, text in character sets other than US-ASCII, multiple-font text, binary data, images, audio, and video, regardless of the limitations of the intervening mail transport systems. It does not define any extensions to the syntax of Internet email message headers to permit anything other than US−ASCII text data.

MIME defines a number of data types (called *content types*), each with a small number of subtypes, to specify the contents of mail objects. Each object becomes part of the body of the message. The available content types are

`text`

to represent textual information in a number of character sets and formatted text description languages

`message`

to encapsulate messages within others and indirectly refer to data external to the message

`multipart`

which can be used to combine several body parts, possibly of differing data types, into a single message

`application`

to transmit application−specific or binary data

`image`

to transmit still image data

`audio`

to transmit audio or voice data

`video`

to transmit video or moving image data, possibly with embedded audio

Some of the available subtypes of these content types are text/plain, for unformatted ASCII text, image/gif, for GIF files, application/postscript, for PostScript files, and message/external-body, which indicates that the actual message data is not directly contained in the message, but merely referenced, and specifies how to retrieve it. MIME also specifies encodings for data, so that binary data is not altered when passing through mail transports having data or character set limitations.

2.6. Basic Features

There is a basic set of features provided by every mail user agent. For reading mail, facilities are provided to view messages, possibly paging them through a filter that displays them a screenful at a time, using some selection criteria to choose which message to display. Mail readers also allow the user to save or file messages. Most UNIX mail readers allow the user to *pipe* a message to an external program, since UNIX makes this easy. Lest mailboxes grow without bound, mail readers also allow the user to discard messages, removing them from the mailbox. When sending mail, user agents allow composition of the message to be sent, with varying degrees of sophistication, and will transfer control of the message to the underlying mail transfer system when the user indicates.

2.7. UNIX Email User Agents

UNIX offers a number of mail user agents to choose from. There is the ‘‘original’’ UNIX mail user agent, */bin/mail* (often referred to as *binmail*). This has fallen out of favor and given way to the slightly more featureful *Mail* and *mailx*. These are two variants of the same program, and are quite similar. There are several other public−domain user agents, the most popular of which are *elm* and its descendent *pine*, and the system which some consider the most UNIX−like of the available user agents, *MH*. The strengths and weaknesses of each of these systems will be described briefly below. This is by no means a comprehensive list of available user agents, simply those in use by emm’s future users when emm was designed.

2.7.1. Binmail [4]

binmail is useful to examine only as an example of the base level of functionality provided by UNIX user agents. On most modern systems, it is used only as a message delivery agent. It is severely limited: it presents mail in a strictly sequential last–in first–out fashion, using a fixed mailbox name. It provides no opportunity to choose which messages to view. Its user interface is composed entirely of single–letter commands, which it uses to direct the disposition of each message.

2.7.2. Mail and mailx [5] [6]

Mail and *mailx* are surprisingly sophisticated user agents, once the user deciphers them. They preserve the *binmail* model of single–letter commands and provide only rudimentary help. The message composition facilities they offer are cryptic at best. On the good side, they offer a useful basic set of message grouping mechanisms, so the user is not limited to *binmail*'s strict LIFO viewing order. *Mail* and *mailx* also offer access to multiple mail folders, but again in a cryptic fashion. They require that the user know a great deal about the underlying mail transfer system and UNIX system, making them unsuitable for novice email users.

2.7.3. Elm and pine [7] [8]

elm and *pine* offer a screen–based interface to mail, which makes them easier than most user agents to learn and use. They are, however, menu–oriented, which generally inhibits expert users, and retain the single–letter command model. These mail agents are reasonably full–featured, and have evolved to provide most of what users require. Nevertheless, they do have their shortcomings. In addition to the single–letter commands, they do not offer a way to group messages and act on a set of messages as a single unit. They also try to do too much, more than a user agent really needs to, incorporating functions such as mail routing which are best left to transfer agents. This has resulted in a great deal of ‘‘bloat’’.

2.7.4. MH [9] [10]

MH is an interesting system. As opposed to the majority of user agents, which encapsulate their functionality in a single program, *MH* consists of a set of programs, each performing one of the tasks required of a user agent. This makes it one of the best examples of the UNIX ‘‘tools’’ philosophy. This modularity, on the other hand, while MH’s greatest strength, is also a weakness. MH is very hard to learn and use: there are too many commands, with too many options, and not enough ‘‘glue’’. The user is expected to write shell programs to tie all the pieces together, which quickly overwhelms the beginner. Many users feel more comfortable in a recognizable ‘‘mail program’’ –– while the power of the shell is attractive, the result is often a mess. MH users also wind up trading its functionality for speed. *MH* is not without its strengths, though, which are considerable. There are good folder and draft capabilities, and powerful message grouping commands exist. If one is willing to take the time, and write a certain amount of shell script glue, MH can be made to do nearly everything. There is a downside: once a user chooses MH, there is no turning back –– it uses a mail file format incompatible with all other UNIX user agents.

3. Emm System Requirements and Goals

The *emm* (**e**lectronic **m**ail **m**anager) system was created both to remedy the perceived misfeatures of other available mailers and to experiment with a set of new mail user interface ideas. It provides new user interface features as well as an interesting set of message operations not usually found in UNIX mail user agents. This section discusses the requirements and goals for emm, with an emphasis on what is new and interesting about the emm system.

3.1. UNIX Mail User Agent Shortcomings

The specification process began with a look at the shortcomings of the UNIX mail user agents in common use. These mailers had much to recommend them, but each had one or more flaws that rendered it unsuitable. The first problem was one of scale. The existing UNIX mailers seemed poorly designed for users who handled more than a few dozen messages daily, and the did not seem to have been designed to manage mail from a variety of different sources.

These mailers were not sufficiently flexible: they did not allow the user to group messages in many different ways or use multiple grouping criteria so that operations could be performed on the set of messages rather than singly. Not enough operations on messages themselves were provided.

A third area of problems was their unsophisticated interaction with the user. UNIX mail system designers are too fond of single–character commands, and menus are not a great improvement. This has resulted in clumsy mnemonics for commands which happen to begin with the same letter (*read*, *reply*, and *resend*, for instance), to the confusion of users. These mailers generally provide only rudimentary help, and do not offer any command entry aids such as command line editing, word completion, or command history. There are several other minor problems with these mailers, including their weak support for multiple mail folders and draft messages, and an annoying inability to handle anything but textual mail. The basic problem appears to be one of philosophy: existing UNIX mailers want to use the ‘‘stream’’ model (messages flow in, are read and possibly reacted to, and discarded) while a better model for modern systems appears to be the ‘‘database’’ model (mail is received, viewed and possibly acted upon, and then transferred to a folder together with related messages for later reference).

3.2. Emm System Requirements

With those shortcomings in mind, a set of requirements for the emm system emerged. The requirements involved aspects of both user interaction and operations on messages.

The first objective was to provide a testbed for exploring new (for UNIX, at least) mail user interface ideas. Emm provides command line editing, to assist in command entry, and a novel help and command completion system.

Another area of interest was providing new ways of operating on messages. The emm system provides *active messages*, which can transmit not only data, but also allow the sender to specify his intentions about how the message should be disposed. Also provided are *read receipts*, which allow the sender to be notified when the recipient actually reads a message, as opposed to when it was delivered.

Emm set out to provide more flexible mechanisms for managing and archiving large volumes of mail (another requirement imposed by its future users). The concept of *message sequences* was introduced to effect this. A message sequence is a set of messages sharing some trait, and capable of being acted upon as a unit. A mail filtering system is provided to organize and possibly discard incoming mail. Facilities to manage multiple mail folders to group related messages were another basic requirement.

The world is moving quickly beyond simple textual email. Emm, therefore, had to make it possible to manage non-textual mail even in a text-based environment. It is possible to manipulate mail containing different types of *mail objects* with emm. Facilities to both view and send multimedia mail are provided. Users may attach arbitrary files to outgoing messages, and have them delivered without loss of data.

In addition to mail reading, emm was designed to permit more flexible management of message sending. It is easy to save records of both incoming and outgoing mail, convenient for coordinating all sides of a multi−way email conversation. A further requirement was the ability to manage *drafts* of outgoing messages, with the system able to track multiple messages in various stages of completion.

There were a number of minor requirements, as well. The system had to provide a limited amount of programmability, at least to the point of being able to take commands from a file. Support for an integrated editor was also desirable, which allows certain editor−specific things to be done while sending mail (separate editor windows for the message header and body, for example).

3.3. Design Alternatives

Aside from the requirements for the system, there were a few explicit design alternatives not taken. Emm does not provide a mail database in the sense of Ingres or Informix, nor does it provide a query language like SQL. It does provide a means to select messages based on certain criteria, but does not go so far as to implement a true query language. That was judged too complex for these purposes. The emm system does not have an internal programming language like a shell −− the inclusion of such a facility was considered more complex than the need warranted. The system is not window−based, or even screen−based, as the majority of its potential users felt more comfortable with a command−line style of interaction. The MH approach of a set of individual programs, each implementing a single command, was considered, but rejected as unwieldy. The emm system is not a monolithic single program, but the user needs to run only one, giving the impression of a single ‘‘mail application’’.

4. Design

As was the case with the requirements, the emm system design was driven by its features. This section describes the design of the emm system features which implement the requirements and goals outlined in the previous section.

4.1. System and Program Structure

The first aspect of the design considered was that of program structure and division of functionality. The decision was made to split functionality between a few separate modules. As a consequence, the emm system is actually composed of several different pieces: *emm*, the program used to read mail, *sm*, used to send mail, and *se*, an editor integrated with sm and used to compose outgoing messages. These three applications cooperate with each other and a set of auxiliary utilities (e.g., a program to pretty−print individual messages on a PostScript printer, and utilities to display message parts in formats other than ASCII text) to present a unified interface: the user needs to run only emm to take advantage of the full system. Hereafter, the complete system will be denoted as ‘‘the emm system’’, while individual components will be referred to by name.

Another design decision concerning structure was to make emm *modal*: there is a *read* mode, which emm enters while reading a sequence of messages, and a *top−level* mode. The primary difference between the two modes is that read mode forces operations on the *current* message (the message from the sequence supplied to the *read* command that is currently being read), while top-level mode permits operations on message sequences. Recursive read modes are disallowed.

4.2. The User Interface

The user interface concepts are the first aspect of the design the user encounters, so it seems reasonable to consider them next. Because the emm system eschews the traditional UNIX single−character style of commands, it quickly became obvious that much more typing would be required to use it, even if abbreviated commands were supported. With more typing comes more mistakes: the longer a command, the more likely it will need corrections before being submitted. With traditional UNIX mailers, the only way to correct such errors is by backspacing or killing the entire line. UNIX shells such as *bash* and *ksh* have solved this problem by offering *command−line editors*, which allow commands to be entered using the text editing directives of popular UNIX editors such as *emacs* and *vi*. The command editor included with emm is more advanced than most. It offers the user a choice of editors to emulate, offers complete freedom in binding key sequences to editing commands (including *macros*, which allow a key sequence to insert text or perform multiple editing commands), and provides a number of variables to control aspects of editing (for example, the choice of whether to scroll horizontally to display lines that are too long for the screen or wrap them onto successive lines). This editing ability became the centerpiece of the new user interface features. There is *command history*, which allows previously−entered commands to be recalled, corrected or altered if necessary, and resubmitted as new commands, to avoid forcing the user to retype frequently−entered commands, or correct errors in previously−typed command lines. This command history can be saved across mail sessions. The command editor and both emm and sm combine to offer *completion*, the ability to type only a portion of a command *field* (command name or argument), hit a key, and have the system complete as much of the word as it recognizes. This has been extended to perform filename completion as well, for commands which take a filename as an argument (*copy*, for instance), and that is where most of the benefits accrue.

The other interesting aspect of the user interface is the sophisticated help system. At any point while typing a command, the user may press *?*, and the system will respond with a list of the available valid alternatives for a partially typed word or a hint as to what input it is expecting. For example, if the user types a portion of a command and then *?*, the system will show the commands which match the current input. Both emm and sm also include a *help* command, and as a consequence of the help system, a list of topics on which help is available can be generated quickly by typing *help ?*.

4.3. Message Sequences

To support the requirement that it be easy to group messages and operate on such a group as a unit, the concept of *messages sequences* was incorporated into the design. A message sequence is simply a series of messages having some trait in common. There are a number of specifiers which identifty a message sequence; the intersection of the specifiers given determines the set of messages to be processed by a given command. Examples of message sequence specifiers include *from chet*, *to bash or cc bash*, and *from root since last thursday and before yesterday*. Message numbers and ranges of message numbers are also allowed as sequence specifiers. Nearly all of the attributes of a message are available as selection criteria. Since it is often desirable to operate on messages which are *not* members of a sequence, sequence negation is supported. The system also remembers the last sequence given to a command, and allows access to it through a special specifier. In the absence of a message sequence argument, most commands take the current message as the default sequence.

4.4. New Message Types

New mechanisms were also developed to support *active messages* and *read receipts*.

Active messages are messages which contain not only data, but a suggestion from the sender about how to treat that data. This suggestion is usually in the form of a UNIX command to be run with the body of the message as input when the recipient views it. For security reasons, the receiver is shown the command before it is executed, and given the choice of whether or not to allow execution to proceed. No such command is ever run without the recipient's permission. This feature can be used to send documents which automatically format and print themselves when received, to exchange archives which automatically unpack themselves on arrival, or to send a bug report in a format which allows a program maintainer to automatically reproduce the problem.

Read receipts are automatic notices returned to the sender of a message when the receiver views it for the first time. This sort of notification is more useful than a delivery receipt, such as that provided by *sendmail*, because a sender does not know that the recipient has actually seen his message when a delivery receipt is returned. Read receipts considerably reduce the number of nuisance "have you read my message yet?" questions. Since this is a controversial feature, it is optional. A message recipient may choose on a per−message basis whether or not to send a requested receipt. Generally, persons will send read receipts only to users with whom they have agreed to exchange them.

4.5. Additional Emm Features

In addition to these new features, emm includes the normal set of mail handling commands one would expect from a UNIX mailer, and a few more.

Messages may be *flagged*, to indicate priority; summary lines describing each flagged message are usually displayed on emm startup. Emm has a two−stage message deletion process: a message is marked for deletion with a command, but not actually removed until emm exits or an explicit command is given to purge the mailbox. Messages can be undeleted before expunging the mailbox, and there is an command to exit emm without removing deleted messages from the mailbox. Emm allows mail folders to be switched with a single command, since the support of multiple mail folders was one of the originally specified features.

4.6. Sm Features

Sm's features include a powerful aliasing facility, allowing users to associate an arbitrary word or name to a complex address or a list of addresses. Aliases may reference other aliases. Access to a user−defined spelling checker is available, so that messages may be checked before being sent. For maximum flexibility, sm allows nearly every message header to be edited by the user. Additional user−defined headers may be added during composition; sm ensures their syntactic validity before sending the message.

Emm and sm cooperate to make it easy to track electronic conversations. Emm can put messages into arbitrary folders, even saving messages to folders with the same name as the login name of the person who sent the mail. Similarly, sm can be customized to place copies of outgoing messages into any file, and it is possible to have outgoing messages automatically saved into folders whose name is the login name of the user who is to receive the mail. Files may be specified as message recipients, and sm performs delivery directly to those files. These features allow users to maintain mail folders which are a log of both sides of a discussion. Sm will also, at the user's direction, write a one−line summary of outgoing mail to a file (the equivalent of the contents of the office ''outbox'').

Another requirement placed on sm was the ability to manage *draft* messages. Consequently, sm was designed to be able to restore the state of a message sending session completely from a saved message file. The file contains the partial message and all headers, with addresses expanded. It is possible to have several incomplete messages in various stages of composition simultaneously. Sm allows the user to specify the directory where drafts are kept.

4.7. Multimedia and Non−Text Mail

The ability to cope with non−textual multimedia mail, even in a text−only environment, is part of the design of both emm and sm. The emm system uses the **MIME** message format defined in Internet Working Group RFC 1341, since it is fast becoming the standard for multimedia mail in the Internet community.

Sm uses the notion of *file attachments* to associate data in formats other than text with messages. A message is permitted to consist only of attachments, without a text message body.

Emm uses a multi–level architecture to display non-textual messages. Emm relies on a set of auxiliary programs to display MIME content types other than text. These auxiliary programs can display the messages up to the capability of the underlying system –– a workstation is obviously capable of displaying more MIME content types than a VT100. The user may specify programs to interpret specific types and subtypes; emm provides a default set of display programs. The user configures this via a *mailcap* file, which emm interprets. More details are given in the following section on implementation.

4.8. Customizability

The emm system was required to be customizable. There is an extensive set of variables which control nearly every aspect of emm and sm behavior. Commands exist to set and display the values of all variables, and there is internal help available for each. A user may set variables and command macros (and, in sm, define aliases) until he is satisfied with the state of his environment, then issue a command to write the settings to a *profile* file, which emm and sm read on startup. There is also a command to interactively create an initial profile.

4.9. Design Summary

The design of the emm system was primarily driven by the requirements placed upon it due to the features that it needed to support. A small set of abstractions was selected, based partly on the requirements and partly on an examination of the architecture dictated by these requirements. The primary abstractions chosen were message sequences, the user interface, and mail objects/body parts, and the system was designed around them. The implementation follows from this design in a straightforward fashion.

5. System Implementation

This section discusses certain aspects of the emm system implementation. The overall system architecture is described, and the implementation of the basic system abstractions is covered.

5.1. System Structure

The first implementation decision became apparent almost immediately. The system would not attempt to implement all of the required functionality in a single module. To do so, it was decided, would quickly cause the system to become more complex than would be manageable, even by a single developer. Consequently, the system was initially split into several parts:

emm The mail reader, the program that manages incoming mail

sm The program to compose and send mail

se An *emacs*-style editor closely integrated with sm, used to compose messages. This contains several mail support features, including separate editing windows for message headers and body, and MIME–compliant formatted text support.

There is also a suite of utilities, including programs to

- view messages with non-text body parts
- filter incoming mail using *delivery scripts*
- format and print mail on PostScript printers
- reliably deliver messages to files
- a Post Office Protocol client to retrieve mail from a server
- programs to watch for and notify of receival of new mail

- programs to print and manage folders of mail messages

Emm and sm are quite similar, and are implemented as separate applications sharing a large body of common library code.

5.2. Basic Data Structures

At the heart of **emm** is its representation of folders and messages. A folder is simply a text file containing a sequence of messages. This format was chosen for compatibility: the UNIX message delivery system delivers messages to a file in this format, and other UNIX tools operate on these files. One of the benefits of UNIX is that all tools are designed to work together, each performing one specific task. As a consequence, UNIX tools have wherever possible avoided defining special−purpose binary formats. Most tools deal with files containing lines of text. Using such a format for a mail folder allows the use of other UNIX tools on the folder contents, from using a tool such as *grep* to search a folder to building an index of the messages in a folder using a specialized indexing utility, to actually editing a folder's contents. Since all of these things are possible and desirable, the emm system does not attempt to create and use an entirely new mailbox format, nor does it attempt to maintain any persistant index containing absolute offsets into a folder. An emm folder is therefore a file containing a sequence of messages concatenated together with a special delimiting line to denote the beginning of a new message. This permits maximum flexibility and interoperability while still being reasonably efficient to parse.

Internally, a folder is represented using an index. When emm scans a folder, it builds an array of *message* structures, one for each message. Each array member has indices into the folder file telling where a message starts and where the body of a message starts. This internal representation mirrors the folder's external organization.

A message is represented internally as a data structure containing several pointers into the folder file and some auxiliary information obtained when the mailbox is scanned. The message structure contains indices into the folder file that mark the offset of the start of the message, the offset of the beginning of the message body, and the offset of the end of the message. These are used when displaying the message, or when searching for a message containing a specified pattern. The message structure also holds values used to make the parsing of message sequences efficient, using the principle of *don't read things multiple times, read once and remember*. These values are read and decoded when the mailbox is scanned or when new messages are incorporated into the mailbox. There are structure members to hold the mail addresses representing sender and recipient information in a format that allows them to be searched quickly, members to hold the set of headers from the message, and a decoded version of the date the message was sent, again to allow efficient comparisons.

Similarly, sm internally represents a message being sent in a way that mirrors its external representation. A message, to sm, is a list of headers, an optional supplementary list of user−defined headers, the text of the message, and a list of file attachments. When the message is sent, all of these pieces are combined into a single unit that is submitted to the message transfer agent. Each piece may be manipulated and edited individually − for example, the text of the message is simply a file which may be edited. The only required portion is the list of headers, since RFC 822 requires that certain headers appear in the outgoing message. Sm ensures that these headers are syntactially correct before submitting the message to the message transfer agent.

5.3. Implementation of the User Interface

At the base of the user interface is a modified version of the GNU *readline* library. It provides command−line editing, complete with the ability to rebind keys and bind key sequences to arbitrary editing commands or text macros. The editing library integrates the GNU *history* library, which allows commands to be recalled and reentered. Readline allows text to be *killed* and later *yanked* back into the same or another line, and implements infinite editing command undo.

The completion feature is called from the readline library when it reads a tab character. The completion behaves as one would expect: the readline library calls a completion function with the text of the line typed so far, if any, as an argument, a list of possible completions of that text is generated, and if the text to be completed is unique, the (single) completion is inserted into the line. If there is more than one completion, the characters completing the longest common prefix are inserted. Each command has its own completion function, so that completion of different types of arguments (filenames versus a list of mail aliases, for example) may be accomplished cleanly. The readline library handles the details of inserting the proper number of characters into the line; each completion function is responsible only for returning a list of strings which are the possible completions. Both emm and sm register a completion function with the readline library, and that "master" completion function is responsible for dispatching to the proper command−specific completion function. The master function receives as an argument the partially-typed line, incrementally parses it to find the command for which completion is being attempted, and calls that command's completion function, passing it the rest of the line. If a complete command has not been typed, the master function generates the list of completions itself from the list of valid commands.

The help system is another feature of the user interface initiated by the editing library. Whenever the "?" character is typed on a line, the library calls the currently-registered help function. As with completion, emm and sm both register "master" help functions with the readline library, and these functions dispatch to command−specific help functions in the same way that the master completion function dispatches to the command–specific completion functions. Each command has its own help function, though there are many commands which share the same one: the commands which take a message sequence as an argument, for example, all use a common function. Each help function generates a list of strings, as with the completion functions, and prints them through a common function interface. This function takes care of refreshing the line and setting things up for command entry to continue.

5.4. Message Sequence Implementation

Message sequences are the second design abstraction described. The implementation is straightforward. A message sequence is composed of a sequence of *specifiers*, the composition of which determines the final sequence passed to a command. Specifiers are short clauses, either a single word such as *deleted* or *unseen*, or a word and an argument, such as *from chet* or *since yesterday*.

Message sequences are parsed left–to–right, with an implicit AND between each specifier. Disjunction is available via *or*, and a sub–sequence may appear in parentheses. Parenthesized subsequences are primarily used to combine a global condition such as *unseen* with a more restrictive one: for example, *read unseen (to bash or cc bash)*. Without parentheses, the sequence would have to be entered as *read unseen from bash or unseen to bash* as a result of AND and OR having equal precedence and being parsed left–to–right.

A message sequence is represented internally as a data structure consisting of a bitmap of message numbers, the length of the bitmap, and several special values, such as the first and last messages in the sequence. A *sequence context* is a stack of sequence structures generated as the result of parsing several message specifiers. It exists to support sub–sequences and parenthesized sequences. The sequence parser creates a new sequence for each specifier it parses, and pushes all structures created while parsing a complete message sequence onto a stack in a sequence context. When the parser reaches the end of a sequence, the bits in all the bitmaps on the stack are ANDed together, and the resulting bitmap becomes the value of the entire sequence.

Most of the emm commands act on their message sequence arguments in the same way. The `copy` command is a representative example: it parses its first argument as a filename to which to copy the messages in the message sequence specified by the rest of the arguments, then loops through the sequence bitmap, calling a function for each bit set in the bitmap. This function appends the corresponding message to the file. If only a filename is supplied as an argument, the current message is used as a default message sequence.

5.5. Implementation of Multimedia Mail

The third implementation abstraction is mail objects and mail *body parts* to support MIME and multimedia mail. Emm and sm each have to support these mail objects, so the bulk of the code for manipulating them is implemented in a library shared between the two programs.

5.5.1. The Sm MIME Implementation

Sm uses the *attachment* paradigm to allow access to MIME mail. A user attaches files containing whatever type of data he wants to transmit, and tells sm how to convert it to one of the standard MIME content types. In some cases, sm can intuit what content type a file should be given by the file's extension (such as *.ps*) or a brief examination of the file's contents (GIF and PostScript files both begin with a known character sequence, for example). Sm determines whether or not the data is binary, and performs any necessary encoding. Both files and the output of programs may be attached to messages: if the filename specified by the user begins with a **|**, sm runs the program specified by the rest of the filename, puts the output into a temporary file, and attaches it as normal.

MIME defines a simple form of formatted text, `richtext`. Sm allows messages to be composed in richtext via interaction with se, the sm editor. Se has editing commands which allow the user to specify that certain text should be emboldened, italicized, or otherwise formatted using the MIME richtext capabilities, and inserts the appropriate richtext commands directly into the message text. Richtext allows users to exchange messages which can be displayed as if output by a word processor, but the syntax is simple enough so that the raw form of the data is readable in the event that the recipient does not have a MIME–conformant mail reader. (Of course, if the sender and recipient have access to the same word processor or text formatting system, files can be attached and be displayed by that system upon receipt using active messages.)

When the user directs sm to send the message, the list of attachments and the text of the message entered by the user need to be converted into a MIME–format message that can be submitted to the message transfer agent. Sm does this by creating a MIME *multipart* message. The first part is the text typed by the user; the second and subsequent body parts are the files in the attachment list. Sm performs the necessary encoding, adds headers describing the format of the entire message, and submits it as usual.

5.5.2. The Emm MIME Implementation

Emm's implementation of MIME message handling and display is done at several levels. Emm displays messages in some MIME content types itself, and relies on auxiliary programs to display the rest. The mapping of content type and subtype to display program is done via a *mailcap* file, as specified in Internet RFC 1343. The mailcap file is used for maximum flexibility: adding support for the display of a new content type becomes a matter of adding a line to the master mailcap file and installing a binary. There is a single system–wide mailcap file and an optional mailcap file for each user. Entries in user mailcap files override those in system files.

When emm encounters a message it cannot display internally, or, in the case of multipart messages, a message body part it cannot display, it consults the user's mailcap file, then the system−wide mailcap file, looking for a mailcap entry that matches the message's content type and subtype. If one is found, the display program specified in the matching entry is executed, with the portion of the message of that content type as input. The mailcap entry can specify arguments to the display command, and contains several flags used to control the environment in which the display program is run. If no display program is found, the user is given an opportunity to save the message to a file or view it as text.

The default display programs specified in the standard emm mailcap file are all shell scripts. These scripts need a mechanism to use as a ''back end'' to re−display data for content types such as message/external-body, where the message itself specifies how to retrieve the actual body, which may be of any MIME content type. Once the script retrieves the data, whether it is from a local file or from a file available for FTP somewhere on the Internet, this back end program re−interprets the MIME content types of this new data and recursively calls the correct display program. Since MIME messages and body parts are allowed to nest arbitrarily, this can get tricky. The back end also displayas the content types that emm handles internally without calling another external display program. Emm passes information about the disposition of such messages to other processes through the environment.

Emm, combined with the mailcap file, a set of display scripts, and the back end message display program, provides a complete and very flexible way to manage messages containing arbitrarily−typed data. The implementation makes the mechanism easily extensible by both the user and system administrator.

5.6. Implementation Statistics

The implementation was done in C. A very rough count of the number of lines of code required, including blank lines and comments (it is the sum of the line counts for each source file composing the program), is

component	code size
emm	14500
sm	10500
libemm	10200
se	17700

This is not intended to be anything more than a gross estimate.

6. Examples

This section presents examples of some of the notable new features of emm and sm.

The emm system is in everyday use by INS employees who keep between 500 and 2500 messages in a "main" mail folder and maintain a set of other folders in which to file messages by topic. These users receive between 50 and 150 new messages daily. Performance has not yet been reported to be a problem.

The rest of this section presents a number of examples of emm and sm use, basically following the progression of a session using emm to manage mail.

The user starts emm by typing *emm*. The system starts, displays an opening banner, and reads in the primary mailbox. After scanning that mailbox, emm checks for new mail and displays a summary of any new messages.

```
odin(2)$ emm
EMM version 2.0 (build 289) of Mon Jun 7 09:40:08 EDT 1993
Complaints/Bugs to chet@ins.CWRU.Edu
Reading /usr/homes/chet/Mail/mailbox...done
Mailbox is /usr/homes/chet/Mail/mailbox, with 1941 messages (read-write)
Reading new mail from /usr/homes/chet/mbox...
N   1942)   6/7 Klaus Wacker    Re: named directories           (2922 chars)
N   1943)   6/7 brown@odin.ins. CWRUnet Problem Report -- Novel (958 chars)
N   1944)   6/7 marla@alpha.ces PARTY-PICS                      (1295 chars)
N   1945)   6/7 Mail Delivery S Returned mail: Service unavaila (2195 chars)
N   1946)   6/7 Jyrki Kuoppala  glibc 1.06 sysdeps/unix/bsd/son (2643 chars)
N   1947)   6/7 Gert Lokhorst   Surveys to investigate informat (1445 chars)
N   1948)   6/7 Mail Delivery S Returned mail: Service unavaila (2693 chars)
N   1949)   6/7 Phil Ruddle     Broadcast of sun information    (2702 chars)
N   1950)   6/7 V.Sander        SUMMARY: Why large INODE-Number (4041 chars)
N   1951)   6/6 Chip Salzenberg Can't get CWRU bash             (2450 chars)
N   1952)   6/7 Jim Peters      Anything new?                   (1406 chars)
N   1953)   6/7 Dan Brown       rover..                         (750 chars)
12 new messages
Last read this mailbox: Mon Jun  7 08:43:48 1993
emm-2.0>
```

To find out what commands are available, it is necessary only to type a *?*. Emm's help system displays possible completions; because no input has been read, every possible command is listed as an alternative. The help system may also be used to display the available variables, and the values of variables.

```
emm-2.0> help  ? help topic, one of the following:
answer            flag              pipe              sort
blank             forward           previous          source
bug               get               print             status
cat               headers           profile           suspend
cd                help              push              take
check             history           pwd               type
copy              ignore            quit              undefine
count             jump              read              undelete
create-init       kill              remail            unflag
date              list              reply             unignore
define            mail              resend            unmark
delete            mark              save-init         unread
echo              message-sequence  send              update
exit              move              set               version
expunge           next              shell             write
emm-2.0> help set  ? emm variable, one of the following:
alternate-charsets      mail-file-mode          reply-to-all
alternate-names         mailbox-file            reply-using-richtext
blank-screen-in-read    new-mail-check-always   save-history
delete-empty-mailboxes  pager                   save-message-by-name
do-ignore-headers       prompt                  save-message-if-exists
exit-on-control-d       read-prompt             send-read-receipt
folder-directory        read-receipt-default    shell
incoming-mail-file      reply-cc-me             show-flagged-messages
list-command            reply-insert-message    use-pager
lpr-filter              reply-prefix-string
emm-2.0> help set incoming-mail-file

INCOMING-MAIL-FILE

The file where the system puts your new mail.
Current value: /usr/homes/chet/mbox
```

The help system can give hints to the user about how to enter message sequences and other arguments as well as commands.

```
emm-2.0> headers  ? enter a message sequence token, one of the following:
after              flagged            or                 text
all                from               previous-sequence  to
and                keywords           read               unanswered
answered           last               recent             undeleted
before             longer             seen               unread
cc                 new                sender             unseen
contains           newsgroups         shorter
current            not                since
deleted            on                 subject
or a message number between 1 and 1963
or range of message numbers, n-m
or range of message numbers, n:m
or range of message numbers, n+m (m messages beginning with n)
or '*' or '$' to specify the last message
or '.' to specify the current message
or a series of comma-separated message numbers or ranges
emm-2.0> headers
```

Most emm commands take message sequences as an argument. The *headers* command, which lists summary information about each message in a sequence, is representative of these commands.

```
emm-2.0> headers undeleted from chet to bfox after 5/1/93
    1774)  5/17 To: bfox@ai.mit Re: Bash and Releases.          (1052 chars)
    1775)  5/17 To: bfox@ai.mit Re: prompt chars and FAQ        (2196 chars)
    1882)   6/1 To: bfox@ai.mit Re: Bash releases and shit.     (1869 chars)
emm-2.0> head undel from bfox to chet after 5/1/93
 A  1860)  5/27 Brian Fox       Bash releases and shit.         (1864 chars)
emm-2.0> h undel from chet (to bash or cc bash) after may 15
    1768)  5/17 To: ckclark@mit Re: PATH=<value> <command> prob (2500 chars)
    1773)  5/17 To: mig@cunixb. Re: Bash/2 + $ENV + .bashrc     (1113 chars)
    1777)  5/17 To: mig@cunixb. Re: Bash/2 + $ENV + .bashrc     (2049 chars)
    1783)  5/18 To: kre@cs.mu.o Re: PATH=<value> <command> prob (1560 chars)
    1825)  5/24 To: ckclark@mit Re: mkinstalldirs sh non-portab (1435 chars)
    1827)  5/24 To: ckclark@mit Re: mkinstalldirs sh non-portab (2376 chars)
    1852)  5/27 To: andersok@ta Re: ISO Latin-1                 (1177 chars)
    1859)  5/27 To: andersok@ta Re: ISO Latin-1                 (2113 chars)
    1880)   6/1 To: tromey@cns. Re: Quoting !                   (1202 chars)
    1885)   6/1 To: tromey@cns. Re: Quoting !                   (2132 chars)
    1891)   6/2 To: jbevans@nl. Re: undefined symbol _alloca wh (1222 chars)
    1892)   6/2 To: friedman@gn Re: Quoting !                   (1493 chars)
    1893)   6/2 To: pst@cisco.c Re: bash-1.12 bug -- does not r (2820 chars)
    1905)   6/2 To: friedman@gn Re: Quoting !                   (2423 chars)
    1906)   6/2 To: jbevans@nl. Re: undefined symbol _alloca wh (2152 chars)
    1907)   6/2 To: pst@cisco.c Re: bash-1.12 bug -- does not r (3750 chars)
```

Emm makes it easy for the user to manipulate more than one mail folder, and to move mail between folders. The primary mail folder is available as the default argument for the *get* command; *get* without arguments returns the user to the primary folder. In the following example, the "+" is shorthand for the user's mail directory, where his mail folders reside.

```
emm-2.0> get +svr ? Name of a file to make new mail file, one of the following:
+svr4    +svr4.2
emm-2.0> get +svr4 ? confirm with carriage return
emm-2.0> get +svr4.2
Reading /usr/homes/chet/Mail/svr4.2...read 2 messages
emm-2.0> h a
       1)   7/9 tc-info@uel.com Re:  svr4.2                     (25366 chars)
       2)   4/1 tc-info@uel.com Re:  svr4.2                     (21853 chars)
emm-2.0> get +wks ? Name of a file to make new mail file, one of the following:
+wksh       +wksh.list
emm-2.0> get +wksh
Reading /usr/homes/chet/Mail/wksh...read 4 messages
emm-2.0> h a
       1) 12/30 pend@usl.com    Re: WKSH paper                  (1418 chars)
       2)   2/5 tc-info@uel.com Re:  wksh                       (4530 chars)
       3)  5/25 pend@usl.com    Menu Creation Convenience       (5127 chars)
       4)   6/2 pend@usl.com    WKSH FAQ #1                     (14153 chars)
emm-2.0> mov +wksh.list 3,4
3-4
emm-2.0> get +wksh.list
Reading /usr/homes/chet/Mail/wksh.list...read 5 messages
emm-2.0> h a
       1)  5/21 pend@usl.com    Re: wksh mailing list           (1108 chars)
       2)   6/2 pend@usl.com    WKSH FAQ #1                     (14153 chars)
       3)   6/4 pend@usl.com    DCE WKSH Prototype              (5486 chars)
       4)  5/25 pend@usl.com    Menu Creation Convenience       (5127 chars)
       5)   6/2 pend@usl.com    WKSH FAQ #1                     (14153 chars)
emm-2.0> d 2
2
emm-2.0> h a
       1)  5/21 pend@usl.com    Re: wksh mailing list           (1108 chars)
  D    2)   6/2 pend@usl.com    WKSH FAQ #1                     (14153 chars)
       3)   6/4 pend@usl.com    DCE WKSH Prototype              (5486 chars)
       4)  5/25 pend@usl.com    Menu Creation Convenience       (5127 chars)
       5)   6/2 pend@usl.com    WKSH FAQ #1                     (14153 chars)
emm-2.0>
```

Users mark messages to be discarded with the **delete** command, but those deleted messages are not removed from the current folder immediately. Instead, messages may be undeleted with the **undelete** before the command to expunge the mailbox removes the messages permanently.

```
emm-2.0> get +svr4.2
Reading /usr/homes/chet/Mail/svr4.2...read 2 messages
emm-2.0> h a
       1)   7/9 tc-info@uel.com Re:  svr4.2                     (25366 chars)
       2)   4/1 tc-info@uel.com Re:  svr4.2                     (21853 chars)
emm-2.0> d 2
2
emm-2.0> h a
       1)   7/9 tc-info@uel.com Re:  svr4.2                     (25366 chars)
  D    2)   4/1 tc-info@uel.com Re:  svr4.2                     (21853 chars)
emm-2.0> undel 2
2
emm-2.0> h a
       1)   7/9 tc-info@uel.com Re:  svr4.2                     (25366 chars)
       2)   4/1 tc-info@uel.com Re:  svr4.2                     (21853 chars)
emm-2.0> h deleted
emm-2.0>
```

While reading a message sequence, emm enters a special mode known as *read mode*. Commands in this mode take no arguments, and operate on the message currently being displayed from the message sequence given to the *read* command.

```
emm-2.0> h a
       1)   7/9 tc-info@uel.com Re:  svr4.2                     (25366 chars)
       2)   4/1 tc-info@uel.com Re:  svr4.2                     (21853 chars)
emm-2.0> r 2
From tc-info@uel.com Thu Apr  1 02:08:31 1993
Flags: 10
Received: from usl.usl.com by odin.INS.CWRU.Edu with SMTP (5.65b+ida+/CWRU-1.5.4
-ins)
        id AA00437; Thu, 1 Apr 93 02:08:31 -0500 (from tc-info@uel.com for /usr/
homes/chet/bin/mailfilter.sh /usr/homes/chet/mbox)
Message-Id: <9304010708.AA00437@odin.INS.CWRU.Edu>
From: tc-info@uel.com
Date: Thu, 1 Apr 93 01:54 EST
To: chet%odin.ins.cwru.edu%uknet@usl.com
Subject: Re:  svr4.2
Received: from uel by usl.com; Thu,  1 Apr 1993 01:54 EST
Content-Type: text
Content-Length: 21321


Thankyou for requesting information from the UNIX System Toolchest and
general USL product information server.

Customers in the United States and Canada who wish to access the
Toolchest directly, may do so 24 hours a day, 7 days a week by dialing
1-908-522-6900 and logging in as guest. This is a 1200/2400 baud line.

Latest toolchest product news is available using the subject field: new-tools.
An index of available information can be obtained using the subject
field: index

An index of available product information (other than toolchest products)
can be obtained using the subject field: pindex
emm-read>  ? read mode command, one of the following:
answer       define       history      pipe         save-init    undefine
blank        delete       ignore       previous     send         undelete
bug          echo         jump         print        set          unflag
cat          exit         kill         push         shell        unignore
cd           flag         list         pwd          source       unmark
check        forward      mail         quit         status       unread
copy         get          mark         remail       suspend      version
create-init  headers      move         reply        take
date         help         next         resend       type
emm-read> print 7
end-of-line expected, '7' found instead
emm-read> print
```

If the user decides to send a message during an emm session, he uses the *send* command. This executes the **sm** program, which handles the details of message sending.

```
emm-2.0> send
To: chet@po.cwru.edu
Subject: test message with sm
     [ user edits the message here ]
sm-2.0> set send-verbose yes
sm-2.0> send
chet@po.cwru.edu... Connecting to po.cwru.edu (tcp)...
220 po.CWRU.Edu ready for love at Mon, 7 Jun 93 10:11:07 -0400
>>> HELO odin.INS.CWRU.Edu
250 Hello odin.INS.CWRU.Edu, pleased to meet you
>>> MAIL From:<chet@odin.INS.CWRU.Edu>
250 <chet@odin.INS.CWRU.Edu>... Sender ok
>>> RCPT To:<chet@po.cwru.edu>
250 <chet@po.cwru.edu>... Recipient ok
>>> DATA
354 Enter mail, end with "." on a line by itself
>>> .
250 Ok
>>> QUIT
221 po.CWRU.Edu closing connection
chet@po.cwru.edu... Sent
```

When the user is composing a message, the editor **se** is used to enter the message text. Se provides several features specific to mail, one of which is the use of separate editor windows for the message headers and body.

```
From: Chet Ramey <chet@odin.INS.CWRU.Edu>
To: chet@po.cwru.edu
Subject: test message with sm
Reply-To: chet@po.CWRU.Edu

-----se 4.2: *SM-Headers*                 (Fundamental)----Top-----------------
This is a sample message.




















-----se 4.2: *SM-Message*                 (Fundamental)----96%-----------------
Wrote /usr/homes/chet/.sm-message, 1 line
```

Sm has the same type of command help as emm, and allows the same sort of system tailoring via user–settable variables.

```
sm-2.0> help  ? command, one of the following:
abort          date           history        restore-draft  suspend
add-command    delete         in-reply-to    save-draft     take
alias          detach         insert         save-init      to
attach         display        mail           send           type
bcc            edit           print          set            user-header
blank          erase          push           shell          version
cancel         exit           pwd            show
cc             expand         quit           source
cd             from           remove         spell
create-init    help           reply-to       subject
sm-2.0> help set  ? sm variable, one of the following:
append-signature-file   exit-editor-auto-send   save-mail-file
ask-to-save-outgoing    exit-on-control-d       save-outgoing-by-name
auto-cc-me              full-name               send-richtext
dead-letter-file        log-outgoing-summary    send-verbose
default-editor          lpr-filter              shell
default-pager           mail-directory          signature-file
default-reply-to        prompt                  spell-checker
deliver-in-background   prompt-for-bcc          summary-file
draft-directory         prompt-for-cc
exit-after-send         request-return-receipt
sm-2.0> help set log-outgoing-summary

LOG-OUTGOING-SUMMARY

If "yes", sm will save a one-line summary of each outgoing message to
the file named by the "summary-file" variable
Current value: no
sm-2.0> help set summary-file

SUMMARY-FILE

The file to which sm writes one-line summaries of outgoing messages if the
"log-outgoing-summary" variable is set
Current value: (null)
```

Sm provides commands to save and restore message drafts. The user may set a variable to specify the directory where drafts are kept. One may start a session, save the message as a draft, and later restore and send it.

```
odin(2)$ ./sm
To: chet
Subject: draft example
sm-2.0> save-draft  ? name of file in which to save draft, one of the following:
sss
sm-2.0> save-draft d1
sm-2.0> restore-draft  ? name of a file from which to restore draft, one of the following:
d1   sss
sm-2.0> restore-draft d1
Restoring message from /usr/homes/chet/Mail/drafts/d1...done
sm-2.0> set send-verbose no
sm-2.0> send
chet -- queued
sm-2.0> quit
odin(2)$
```

Sm provides a rich aliasing facility. Aliases can be used as shorthand for long or complicated addresses, or as mailing lists, in which case a single alias expands to multiple addresses. These are some examples of mail aliases.

```
alias bug-bash  bug-bash@ai.mit.edu
alias fb  Fast and Bulbous <bfox@ai.mit.edu>
alias bfox  bfox@gnu.ai.mit.edu
alias me  chet
alias bash-testers  arnold@cc.gatech.edu, composer@beyond.dreams.org,
        friedman@gnu.ai.mit.edu, joshua5@cs.bu.edu, dob@inel.gov,
        mjo@fmsrl7.srl.ford.com, penneyj@slc.com, bammi@cadence.com,
        tbunce@cix.compulink.co.uk, trost@cse.ogi.edu, zoo@cygnus.com,
        lubkin@cs.rochester.edu, james@bigtex.cactus.org,
        chip%fin@myrddin.sybus.com, dbrooks@osf.org, Greg.Onufer@Eng.Sun.COM,
        kre@munnari.oz.au, hvf@gr.osf.org (alpha), tmwalden@saturn.sys.acc.com,
        watson@kodak.com, glenn@mathcs.emory.edu, penningt@reason.psc.edu,
        devet@adv.win.tue.nl, khera@cs.duke.edu, ejnapier@spock.cs.twsu.edu,
        wieting@tweety.llnl.gov, geoff@well.sf.ca.us, de5@ornl.gov,
        kayvan@satyr.sylvan.com, smd@uunet.ca
```

The help system can be used to find all the aliases that are currently defined, and the **expand** command will show what they expand to.

```
sm-2.0> alias  ? name to alias
or existing alias to unset, one of the following:
ais-operations    bfox              freeport-testers  shell-list
bash-folks        bug-bash          me
bash-testers      fb                posix-folks
sm-2.0> expand bash-testers
bash-testers: arnold@cc.gatech.edu, composer@beyond.dreams.org,
        friedman@gnu.ai.mit.edu, joshua5@cs.bu.edu, dob@inel.gov,
        mjo@fmsrl7.srl.ford.com, penneyj@slc.com, bammi@cadence.com,
        tbunce@cix.compulink.co.uk, trost@cse.ogi.edu, zoo@cygnus.com,
        lubkin@cs.rochester.edu, james@bigtex.cactus.org,
        chip%fin@myrddin.sybus.com, dbrooks@osf.org, Greg.Onufer@Eng.Sun.COM,
        kre@munnari.oz.au, hvf@gr.osf.org (alpha), tmwalden@saturn.sys.acc.com,
        watson@kodak.com, glenn@mathcs.emory.edu, penningt@reason.psc.edu,
        devet@adv.win.tue.nl, khera@cs.duke.edu, ejnapier@spock.cs.twsu.edu,
        wieting@tweety.llnl.gov, geoff@well.sf.ca.us, de5@ornl.gov,
        kayvan@satyr.sylvan.com, smd@uunet.ca
```

Sm and emm both allow customizability via startup files. Here is an example of an emm startup file.

```
#
# emm init file for Chet Ramey
# saved automatically at Tue, 1 Jun 1993 15:34:59 -0400
#
set alternate-names  chet@*.INS.CWRU.Edu chet@po.CWRU.Edu
set exit-on-control-d yes
set incoming-mail-file /usr/homes/chet/mbox
set list-command /usr/local/lib/emm/mailp
set lpr-filter /usr/local/lib/emm/mailp
set mailbox-file /usr/homes/chet/Mail/mailbox
set new-mail-check-always no
set pager /usr/local/bin/less
set prompt "emm-2.0> "
set read-receipt-default no
set reply-cc-me yes
set reply-insert-message yes
set reply-to-all no
set save-history yes
set show-flagged-messages yes
set use-pager yes
```

7. Conclusions

The emm system described here is a robust and flexible mail handler for the UNIX environment, suitable for both beginners and experienced users. It uses a few design abstractions to achieve a clean interface and implementation. Like any sizeable system, it has a number of good points, and its share of problems.

The line editing, history, command completion, and dynamic help system together are a great benefit. Not only is line editing directly visible and beneficial to the user (to the point of emulating the standard UNIX terminal interface well enough to allow the user to ignore the editing features if he wishes), it forms the basis of the rest of the user interface. The readline library is well−written enough that it was extremely easy to incorporate and extend. Having used a set of tools with a common interface based on readline and similar interfaces for some time now, I would never go back. The benefits are too great to ignore.

Another of emm's strengths is the degree to which it can be customized. There are different schools of thought concerning this issue, but I believe that the user should be given the freedom to choose what he feels is best for each "controversial" feature. Nearly every point in the program where a decision must be made can be tailored using a variable setting. The system is customizable enough that the user can elect not to use portions of it: if a user is not comfortable with **se**, for instance, he may elect to use a different editor (with a consequent loss of functionality, of course).

Message sequences are a powerful referant, and another of emm's benefits. They make it possible to perform complicated searches of a mailbox in an intuitive fashion. The ability to tell emm to "show me everything since last Wednesday" is quite valuable.

An additional asset is the ability to manage non−textual mail. It creates a much simpler model: the user can now include nearly anything in a mail message without having to worry about limitations of the mail transfer system such as line length restrictions or inability to transparently handle eight−bit data. Emm's design allows the user to extend the system with body part display programs, and even create and specify how to display new body parts.

Now, the problems. The performance of the systems as a whole is not what it could be, though nobody has reported it as a problem so far. The techniques used to implement message sequences and mailbox searching are in need of improvement. This is not really a problem with the system design, though, but one of implementation, and certainly an area for future work.

The system's split into a set of discrete programs is not without its disadvantages as well. Certain functionality is duplicated, despite the extensive use of a common library, and the split sometimes forces a convoluted interface, especially when emm needs to pass information to sm about a message that is being forwarded or replied to. Most of this clumsiness is, thankfully, hidden from the user.

One thing that neither emm nor sm provide that some users would like is a real programming language. Emm and sm have the ability to take commands from a script file, but there are no flow control or conditional constructs. There is not even a way for a script file to tell whether emm is in read mode and alter its behavior accordingly. This is an area for future investigation, though a full programming language is probably more than this application requires.

The emm system could be more window−based than it is, and this is perceived by some as a weakness. I would argue that the current command−line interface is quite powerful, and that most of the mail being sent today does not require the capabilities of a window system. Users who have a window system may take advantage of it, since emm allows users to supply their own programs to display message bodies, and even to display textual bodies. Sm's ability to attach the output of a program can be used to provide more complicated cut−and−paste capabilities than text windows such as *xterm* provide, should that aspect of a window system be desired. I would further argue that existing window−based mail readers using a mouse as the primary input source provide no message selection capability comparable to emm's message sequences.

Overall, emm has fulfilled its objectives. No other UNIX mail reader provides its combination of features and ease of use, and it has proved itself capable of handling the large amounts of mail for which it was designed.

8. References

1. Postel, J.B. Simple Mail Transfer Protocol. August, 1982, USC/Information Sciences Institute, RFC-821.

2. Crocker, D. Standard for the format of ARPA Internet text messages. August, 1982, Uiversity of Delaware, RFC-822.

3. Borenstein, N. and Freed, N. MIME: Multipurpose Internet Mail Extensions. June, 1992, Bellcore, RFC-1341.

4. *binmail(1)*. In *UNIX User Manual, 4.3 BSD Edition*, Computer Systems Research Group. April, 1986. University of California at Berkeley.

5. *Mail(1)*. In *UNIX User Manual, 4.3 BSD Edition*, Computer Systems Research Group. April, 1986. University of California at Berkeley.

6. Schoens, K. Mail Reference Manual. In *UNIX User's Supplementary Documents, 4.3 BSD Edition*, Computer Systems Research Group. April, 1986. University of California at Berkeley.

7. Weinstein, S. The Elm Users Guide. Datacomp Systems, Inc. 1992.

8. Weinstein, S. The Elm Reference Guide. Datacomp Systems, Inc. 1992.

9. The Rand MH Message Handling System. In *UNIX User's Supplementary Documents, 4.3 BSD Edition*, Computer Systems Research Group. April, 1986. University of California at Berkeley.

10. Rose, M. MH.5: How to process 200 messages a day and still get some real work done. In *Proceedings of the Summer 1985 Usenix Conference*. June, 1985. Usenix Association.

9. Appendix

This section contains the manual pages for several of the emm system applications, in UNIX *man* format. Manual pages are included for

- emm
- sm
- se, the sm editor, and
- mdispmsg, a utility used to display some of the MIME content types.