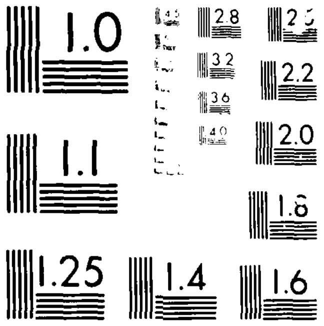
AD-A168 594 MAIL SYSTEM INTERFACE ISSUES WITH A UNIX-MACINTOSH IMPLEMENTATION(U) BROWN UNIV PROVIDENCE RI DEPT OF COMPUTER SCIENCE J J BOWE APR 86 CS-86-10 1/1

UNCLASSIFIED F/G 9/2 NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

(1)

BROWN UNIVERSITY

Department

of

Computer Science

1

Mail System Interface Issues With a UNIX-Macintosh Implementation

by

John Joseph Bowe

Technical Report No. CS-86-10
April 1986

Sc.M. Project

Submitted in partial fulfillment of the requirements for the Degree of Master of Science in the Department of Computer Science at Brown University.

[illegible]
Thomas W Doeppner
Advisor

Mail System Interface Issues With a UNIX-Macintosh Implementation

John J. Bowe

Brown University Computer Science

April 1986

Abstract

An electronic mail system is a communication tool employed by users of computer systems. Messages are typed by one user, stored on a computer, and read by any number of users. A mail program is simply an interface between a human and a store of mail messages. *Therefore, there are two possibly independent interfaces to* consider: one is the interface between the person and the mail program, and the other that between the mail program and the actual stored messages themselves. This project addresses these issues. A prototype front end was implemented on a Macintosh with the guidelines of user friendliness used by many Macintosh programs. An interface to the department's mail server, a VAX-11/750 running UNIX 4.2bsd, which was implemented in the summer of 1985, was written to provide an efficient exchange of mail messages between the Macintosh and UNIX host.

INTRODUCTION

An electronic mail system is a set of tools shared for communication among users of computer systems. Users need not be using the same machine or even the same kind of machine to relay messages. Mail messages are typed by one user on a computer read by other users on another computer. What happens to the message between the users can vary from being complex to very simple. The simplest case would be having a mail program read keyboard input and store the message directly in the destination mailbox file. This is sufficient for relaying a simple message to another user of the same computer. Some possible paths that a message can take include: being formatted with a text processing program such as *nroff* or *fmt*, being sent through an accounting program to charge the user for the message service, being transferred across the country over telephone lines to another computer, or being moved by a high-speed local area network, such as Ethernet, within an organization to a specialized mail handling computer. In most cases the user is unaware of what is happening to the message; the destination address of the message is typed and sent with the same confidence and casualness as a telephone number is dialed.

In its simplest view a mail program is just an interface between a human and a store of mail messages. Interfaces to mail messages vary as much as the computer systems themselves on which the messages are stored. There can be a multitude of possible features, even within the same operating environment. Some of these include: being able to organize and store messages in the file system, having a choice of which text editor to use, and having the ability to easily forward pieces of messages to other users. The implementor of the user-interface program must choose a practical and useful subset of features possible in the intended environment.

Another consideration is the target audience of users. A UNIX† hacker would probably not be upset by a cryptic and powerful interface while a naive or occasional user would need an interface where all options are clearly given and catastrophic operations, such as deleting all messages ever saved, are impossible to do unintentionally. An ideal interface would be one that was "fast" to use, that is commands require little user effort, one that has powerful features such as text manipulation, and one where commands and options are obvious.

The following discussions explore three areas. The first will discuss aspects of mail systems in general, in particular what features are useful and when they might be used. The second summarizes the UNIX 4.2BSD mail system and how it fulfills some of these features. It will be assumed that the reader is familiar with the UNIX mail system. (Note: The UNIX 4.2BSD mail program, often referred to as the UCB mail program, will usually be referred to simply as the UNIX mail system in this paper.) The final area of this paper will cover the Macintosh implementation of a mail interface to the UNIX mail system.

INTERFACE FUNCTIONALITY

Below is a detailed discussion of the important functions of a mail interface. Of course what is important depends on such things as user abilities, required functionality, and personal taste. Many important aspects are drawn from the existing ways that people compose and read memos and letters on paper at their desks. This is because people already have methods and systems for dealing with messages at their offices and a computer can enhance their message handling capabilities without forcing them to change their work habits.

Displaying messages

The most important function to a user is the ability to examine messages. There are many aspects of message displaying that a user may wish to use. First of all, there should be some indication of from whom the messages came, and the user should have a choice of which messages to read, if any. There should not be a requirement to read them in any particular order. This idea is similar to thumbing through a stack of (paper) mail, reading the return addresses, and deciding which to read now, which to read later, and which to discard without reading. A useful display might be a list of message senders with the date, subject, and length of each message. The user would then choose what he wants to do with each message.

Long messages sometimes are inconvenient to access on a computer. Many times the person reading the message will want to randomly look at different parts, usually referring back to a passage from the beginning once at the end or skipping backwards a page (or screen). He should not be forced to scan only sequentially from top to bottom, but should be provided with the same random access convenience as a multi-page paper letter. Incorporating several functions to easily access parts of a message borders on the functionality of a text editor. Allowing the user to enter an actual system text editor, perhaps the one of his choice, would permit fast access to any part of the message and give the added power and convenience of that text editor. The additional features of the text editor, such as string searching, can be very helpful for the user.

Very often a user will want to see several messages simultaneously on a display. Being able to display several messages at once can be difficult on a standard terminal. At best an *emacs*-like program can be devised to divide the terminal screen into two areas for independent viewing. The ideal situation would be to have separate windows of a workstation with a message in each one. Each window should be totally independent of the others and should provide full functionality. This feature becomes a necessity when one wants to compare several messages. An added nicity would be to allow the user to size the windows at will to show the desired portions of each message. This could be implemented on a window-oriented workstation such as a Sun, Microvax, or IBM RT-PC.

Another useful feature is to allow easy printing of an individual message. Although this can be accomplished in several steps, saving the message and then running a printing program, it could be part of the mail program itself. This seems to move away from the concept of a "paperless office", which may verge on defeating the purpose of electronic mail, but it will be desirable at times.

Message composition

The simplest case of sending mail is simply giving the destination user to the mail program and then typing the message from scratch. Much of the time this is sufficient. However, there are many occasions where the sender of the message would like to have more information and tools available on the same screen as the message being composed.

A nice feature is to have the ability to send a message while reading another. This enables one to reference a message as another is typed. For example, one could answer questions from someone else and be able to refer to the questions as the answers are typed. This is a very common mode of operation in an office; a person answers a letter with the original message in front of him or her almost without exception. Multiple windows or a split screen, as mentioned above, would be needed for this.

An extension to this is to use a more general interface, allowing incorporation of arbitrary text files and perhaps pieces of text files into mail messages. A workstation with multiple windows and mouse-selectable text make this concept very easy to use, even for a novice. This saves a lot of retyping and time when one wants to forward information to others. As a further extension, messages need not be confined to text. As computers become more widely used, people will want to exchange other forms of information, such as pictures, sound, or video. The format of this information will have to be carefully considered when this is implemented, but that issue will not be discussed here.

At times one may wish to compose several mail messages at once, perhaps to different users. A practical use for this would be to easily ensure that the facts of the messages to two different people are consistent. Again, a window-oriented workstation would be ideal.

Many people like to save messages they send for future reference. Offices have files filled with carbon copies. As convenient option the user interface would have some provision for this, whether automatic or explicitly invoked by the composer.

Saving and organizing messages

Most people reading mail, whether on a computer or from the postman, will want to save some of the messages and discard others. A typical person at home may separate mail into various piles: bills, business reminders, advertisements, or love letters. Old or uninteresting mail will be thrown out or kept in a shoe box in a closet. This idea extends to sorting mail on a computer. The categories may, in fact, be exactly the same as those of the home example. The various "piles" must somehow be incorporated into the file system of the computer. In an office a file is a place to keep a small pile of messages. Also in an office messages may perhaps be put in a huge disorganized drawer filled with hundreds of messages. Thus the concept is easy to understand and use. The point is that a user can organize messages as carefully or as loosely as desired, whether on a computer or in a filing cabinet. The mail program must, of course, interface with the file system and should not overburden the user with the details of the file system itself. It may be convenient to have the mail program keep messages in an area separate from other files on the computer, perhaps in a subdirectory, if the file system is tree oriented. This may help to user organize messages, but may be too strict; he may want messages scattered about in some other logical organization.

Modeless operation

There are advantages and disadvantages to using modes of operation. I define a "mode" as a logical section of a program where only a subset of all commands, usually closely related, may be called. The advantage is mostly for the implementor of the mail program; it is usually easier to program modally. When in a particular mode the user must complete an operation before switching to another mode. For example, a program may be in "read mail" mode where all commands are related to reading and storing messages, but to send a message, the user must leave that part of the program and go to "send mail" mode. As mentioned above, a person may often want to send a message while reading one without losing the state of the original act of reading. An interface using an ordinary terminal is inherently modal; the user can only see and do one thing at a time. People tend to think non-modally. If the mail interface program is non-modal, the user will not be required to adapt his work habits. Instead, the mail program should be used as freely as any other office tool, like a telephone or typewriter. A truly modeless interface would permit a user to both perform any task at any time, and allow several tasks to be done in parallel. For example, someone could be reading two messages at a multi-window workstation, both messages being on the screen in windows.

In another window, he could be typing a message to someone else. At any time he should be allowed to do any of the following: scroll up and down either incoming message, type some more of the message being sent, send that message, start typing a new message to someone else, read any other new messages, or close any of the windows.

Notification of new messages

An obvious necessity of a mail system is notification of the presence of mail. In many homes, one can tell when new mail arrives when the postman drops the letters through the mail slot. One can still see the new mail even if he is just getting around to reading the previous day's mail. However, if a person lives in the country, he may have to walk out to the end of the driveway to check the mailbox. Clearly it is much more convenient to have the mail drop inside one's front door. This notification should work as asynchronously on a computer system as well.

A typical computer user may be logged into the computer all day at work. A significant portion of the day may be spent running a mail program. The user would of course like to know when new mail has arrived, but he should not be inconvenienced needlessly by having to end the mail program, check for new mail, and start it up again to read the new message. This is essentially going out to the end of the driveway to get the new mail. So there are two components to new mail notification: one is the actual notification, and the other is fetching the new message. The notification should be automatic and the fetching should be as easy as possible.

Workstations vs. terminals

Most of the nicer features mentioned above work much better on a window and mouse oriented workstation or are even impossible without one. Most user-computer interaction, however, is still done through plain terminals. Therefore, much of the ideal functionality is impossible to implement. More people are able to use a mail system from ordinary terminals, whether in the office or at home with a modem.

Simplicity and power

As with many computer programs, there is a balance between power and simplicity. A powerful program usually implies complexity. The ultimate goal of a mail program that everyone would use is to provide as much power as possible while still retaining simplicity so the features will not be beyond the grasp of average and novice users. This is a very difficult problem for the designer of any computer system that will be used by a wide variety of people.

One approach to this interface problem is to have a simple base of options, each of which is flexible and easy to understand. All functions could be obvious to novices, while more experienced users could use extensions to the base functions to perform more advanced operations. The program could be flexible enough to allow several ways of accomplishing the desired tasks. Although some power may be sacrificed, usability may be increased through such simplicity and flexibility. For instance, there can be several ways to forward a portion of a message to someone else. One way, perhaps, would be to save the message in a file and use a text editor to keep only the part to be sent, then send that file. Another way would be to directly edit the message with the mail program itself, sending only the desired part.

Flexibility itself of the interface is an important issue. The person should be able to do what he or she wants and when. It is important to try not to force the user to do anything in any particular order. For example, people would probably be annoyed if they were forced to first read incoming messages before they could send any. Also, once an action is started, the user should not be forced to complete it. People will often start typing a message and decide in mid-sentence that they no longer wish to send it. There should be an easy and fast way to abort most operations. This is how people behave so an interface should be as adaptable as possible to people's behavior. The designer of the program should strive to make it as flexible and comfortable as a person's desk.

Providing several ways to perform tasks may add confusion to novice users, but may please a seasoned user. Capable users will choose the method of completing the message handling task which best suits their particular task requirements and personal taste. To solve the problem of confusion, the trickier operations could be made not quite as apparent as the one "standard" method, thus guiding them toward that one method. When they become more experienced, users may wish for fancier operations and naturally try other methods.

No office tool should be designed with the intent of making a targetted user change his or her work habits drasticly. Instead, the tool should fit somewhat into established ways of doing things. This enables anyone to use the tool since there is a very small learning curve. This is the most important idea that the implementor of a computerized message system should keep in mind.

THE UNIX 4.2BSD MAIL PROGRAM

In general many programs have more options than the average user will ever use. The UNIX 4.2bsd mail program is one such program; there are a multitude of options in this powerful program. In conjunction with the UNIX operating system, the user can usually accomplish almost any desired task. However, many more complicated tasks require a fair knowledge of UNIX and the mail program itself.

Since UNIX is a terminal oriented operating system, the desire to perform several interactive tasks simultaneously is difficult to satisfy. On a normal terminal, the UNIX 4.2 job control facility is the closest the user will come to running two interactive tasks at the same time. He or she can easily bounce back and forth between jobs, perhaps two processes reading different mail messages. A danger to this scheme, however, is that these are two independent processes, each assuming that it has sole access to the mailbox. The inconvenience of this approach is that both jobs are not visible on the screen at the same time. A multiple window workstation would ease this problem, however, since the user could see both jobs at once. However, if the mail program is running independently (separate processes) in two windows, both reading the user's mailbox, the same problem arises as in above. Both processes believe they have exclusive access to the mailbox. The user must be very careful with the disposition of his messages.

In a typical mail reading session, a user will invoke the program, which will show from whom the messages came with the time, date, and the subject of the message. The user can read messages as desired, keep some in the mailbox, delete some, and easily reply to any. These nicely agree with the desirable features discussed above.

The UNIX mail program has excellent interaction with the file system. Messages can be written anywhere in the file system (providing the user has proper access). Saved messages can easily be read from files by the program, edited, and saved again. The program interfaces not only with the file system, but other programs such as *vi*, *csh* or any program that uses standard input and output.

Most of the more common commands are obvious. For instance, "p" will print a message, "d" will delete it, and "q" will quit. Some, however, may lead to confusion. For example, there are two different commands to save a message to a file – one will save all the header lines while the other will not. This kind of thing may lead to confusion for a novice and a fairly experienced user as well. A nice feature is the on-line help that is available for every command. Although all the possible options are not laid out in front of the user, there is information available interactively, rather than off-line.

THE MACINTOSH

The Apple Macintosh, introduced more than two years ago, has given many people in both academia and business access to window-oriented and mouse-oriented computing at a fraction of the cost of higher-power workstations. The Macintosh interface has undoubtedly influenced more recent interface designs. Although this interface style was not started by Apple, the widespread use and availability the Macintosh has made the idea of window-mouse computing and the Macintosh synonymous to many.

Naturally, a Macintosh does not have the same power as a more expensive workstation. Such limitations include: memory, screen size, and disk storage, both in speed and capacity. A Macintosh is not only single-user, but also single-task. A mail system that covers the scope of the UNIX mail system would be impossible on such a machine. However, the speed of the graphics, and the interface tools within the Macintosh can provide an elegant user interface to a mail system. This is the human-computer interface mentioned in the introduction. The other interface to consider is that between the mail program, i.e. the Macintosh, and the store of mail messages. This can be accomplished through a slave program run on the mail serving host with which the Macintosh communicates through the serial port. A significant effort of this project was spent on these interfaces.

The Macintosh interface tries to imitate office processes in general to the point of emulating a desk top. This familiarity is comforting to computer novices, while usually not a hindrance to experts. There is a drawback to this, however. The user may continue to think strictly in terms of current office concepts rather than extending his conception of an office.

MACINTOSH MAIL IMPLEMENTATION

A Macintosh interface to the UNIX mail system was designed and implemented to explore and demonstrate some of these ideas. This was written to *conform* to *the* Macintosh *implementation guidelines as* discussed in *Inside Macintosh*. A technical description of the details is given in the appendix. The following discussion assumes that the reader has some familiarity with the Macintosh.

There are four menus in this application: **Mail**, **Edit**, **Connect**, and **Settings**. As is usually the case with Macintosh applications, the first menu, the **Mail** menu, often labeled the **File** menu, contains the open and close entries for windows and files and contains the option to quit the program. The second menu is almost always called the **Edit** menu and is described in more detail below.

All windows support vertical scrolling, with the standard Macintosh scroll bar, but not horizontal. The reason for this is that UNIX mail messages are usually sent on 80 column terminals. Implementation was also easier.

As windows are created on the screen, they are staggered slightly horizontally and vertically until they reach the right or bottom of the screen, respectively. When an screen border is reached, the new windows are created at the left or top edge, depending on which border was reached.

From Window

The **From** window gives the user a list of from whom his mail has come. The menu option is disabled until the user makes his presence known to the host via the **Connect** menu. When selected, the Macintosh creates a window and asks the host from whom the messages (if any) came. The **From** selection is disabled while the **From** window is on the display; it makes sense to only have one of these windows open at any time. This is a "read-only" window. It may be picked at with the mouse, but will not be affected by typed keys or attempts at pasting. When a point of the window is picked, the entire line is selected and highlighted. This is the message that is to be read. When a message is selected and the **From** window is active, the **Read** item will be enabled in the **Mail** menu. When the close-box is picked, the window is removed and the **From** item is once again enabled. When the **From** window is removed from the screen, the **Read** selection is disabled in **Mail** menu. Thus the user has full control over which messages are to be read.

Read Window

A **Read** window displays a mail message from host and is invoked by choosing **Read** from the **Mail** menu. The message displayed is the one selected in the **From** window. The entire window is dedicated to the one message. Thus, each individual message is independent visually as well as logically. Any number of these windows may be displayed at one time, with a limit being the memory capacity of the machine. This window is now a simple text window and its contents can be manipulated as such. When active, it responds to keyboard input, text selection with the mouse, and **Edit** menu commands such as **cut, copy**, and **paste**. **Undo** is not implemented.

The user has several choices upon closing this window with the close box. A dialog box with several *button items* prompts for a response. The text can be either saved to file or discarded. Saved files are of type TEXT. TEXT is the standard application non-specific file type. It can be read by MacWrite. As with all dialog boxes in this program, there is a *Cancel* button in case the user decides not to close the window.

Compose Window

A **Compose** window is used to create a messages to be sent. It behaves like ordinary text window and is, in fact, handled in exactly the same manner as a **Read** window above. This type of window may be created in several ways. Most obvious is by the **Compose** item of the **Mail** menu. Note that this item is always enabled; the user need not be connected to a host to type text into a window. The **Callup File** will also invoke a **Compose** window; this is discussed below. If the user chooses **Reply** to a **Read** window, a **Compose** window will be created for the user to type a response to a message.

The choices upon closing this window are similar to those of a **Read** window. A dialog box prompts for a response. The text can be saved to file, discarded, or sent to the recipient user by passing the message to the connected host. If the user has not connected himself via the **Connect** menu, the send option has no effect. Note that a message can be composed and saved to a file without being sent. This is useful if one wants to compose several messages off-line and send them all later.

Edit Menu

The **Edit** menu strictly follows the Macintosh user ınterface guidelines. There are two reasons for this. First, since edit functions are common to a great number of applications, the interface should remain constant across applications. Second, and more importantly, is that many desk accessories have edit capabilities, most commonly **Copy** and **Paste**. Desk accessories do not have access to information about an application's menus, such as the name of each item. They can, however, get events from the **Edit** menu. To assure the items are in the correct locations, applications programmers should include all the items of the **Edit** menu in the standard order even if the specific application does not support them.

File Manipulation and Off-line Operation

The feature of being able to compose messages and save them to files is useful for composing messages when a connection to a host is unavailable or impractical. Such a case may be when one is working at home and logging in to the mail host by a modem is expensive. Supporting the common Macintosh file of type TEXT greatly expands the flexibility of information exchange among users. The **Callup File** item will read files from MacWrite that have been saves as "text only", Word files, MacTerm files, statistics files from many games, or files from many other applications. For more flexibility, one can copy text from several documents all on the screen at once and paste it into a message to be saved or sent. MacWrite can not do this. The combination of **Compose** and **Call Up** is essentially an editor for plain text.

Settings

The items in the **Settings** menu in general should rarely need attention. These set the various configuration parameters of the mail communication sessions. If one accesses his mail always from the same location, the settings will never change. The following is a short description of each of the **Settings** options. All dialogs below have a *Cancel* option if the user sees that information is correct or does not wish to change it.

The **Connection** item allows selection of the device to which Macintosh is connected. This is done with a dialog box using *radio butto*. The possible connections are directly to a host, through a modem, or directly to Localnet-20† t-box. The connection is remembered when the program ends. The **Login** item in the **Connect** menu uses this information to determine how to connect to the mail serving host.

The **Baud** item gives the option of three baud rates for the Macintosh connection. These rates are 9600, 1200, and 300 bps. As in the **Connection** menu, radio buttons are used to make the selection.

The **T-port** is the Localnet-20 port (t-box) to which the host is connected. This should be set if the connection is directly to a t-box or through a modem, which will use Localnet-20 at the modem pool to connect to the host. This is selected through an *Editable text* dialog.

The **Phone #** (obviously) is the phone number used to connect to the host. This is also selected through an *Editable text* dialog.

The **Login** name is the user's account name n the UNIX host. This is selected through an *Editable text* dialog. The user must have an account on a UNIX host to read his mail.

The **Password** item allows selection of the password for the user denoted in **Login**. Like the other text settings, it is set with an *Editable text* dialog. However, for protection, the existing password is not shown when this item is selected. It is echoed as it is typed for verification, but will never be displayed again.

Check New Mail enables automatic checking for new mail from the host. Will only work if there is a mail checking program such as *hclock* running on the host. The program assumes the receipt of a bell (ASCII 07) means new mail. This is checked in the idle loop of the program, so normal communication with the host will not be disrupted. When selected, the menu item will have a check mark next to it. When selected a second time, this feature will be disabled and the check mark will disappear.

Connect

The **Connect** menu is used to initiate or terminate a mail transaction session with the host computer. The items were designed to provide as much flexibility as possible. **Login** initiates the login procedure with the host. Depending on what was selected in the **Connection** item in the **Settings** menu, the mail program will expect different responses on the serial line.

Logout logs the user off the UNIX host. This need only be done if the he is completely finished with the host. If he plans to use a terminal emulation program to proceed with other work, he should use **Quit**. Using **Logout** does not end the mail program. The user can still create messages, write them to the disk, edit files, etc. without leaving the program. After logging out, **Login** or **Already In** must be used to re-establish a mail session with the host.

Already In is used if the user is already logged in to the host. This will be true in several instances. The first is when the user has already logged in to the host with a terminal emulator and has decided to use the Macintosh mail front end. The other case is when the user has already used the mail program, **Quit** from it to do some local Macintosh work and wishes to re-enter the mail interface.

Quit terminates the Macintosh mail program, but leaves the user logged in to the host. This allows the user to work on other Macintosh applications and to easily return to the mail program or leaves him logged in if he wants to use a terminal emulator.

MAIL-CHECKING DESK ACCESSORY

A very convenient feature of a mail interface is its ability to notify the user that he or she has new mail. This is easy if there is a program in the interface that is dedicated to this task. However, when a person is using a Macintosh, he will undoubtedly be doing other work (other Macintosh applications) some of the time. The purpose of a Macintosh desk accessory is to provide a small service while running any application. Furthermore, a desk accessory must be independent of running applications to assure complete compatibility with all applications. A desk accessory, **MailCheck**, was written to do this.

This is activated like any other desk accessory, through the apple menu. When new mail arrives, the user is notified in two ways, visually and audibly. For each new message that arrives, the Macintosh produces a beep. If this is the first message since the accessory was activated or since the user acknowledged a new message, a small window is created telling the user about the new mail. This window is removed when the mouse is pressed anywhere on it. However, if another window is subsequently created on top of the notification window by an application, the notification window is hidden, partially or fully. Thus, it behaves like any ordinary window. The purpose of this scheme is to remind the user of the new mail even if he chooses to not acknowledge it immediately and to proceed with an application.

Of course, for this to work properly, the host must run some supporting software. The program *hclock*, used at Brown on the VAXes, performs this task nicely. Normally, *hclock* runs in the background and puts text in the status line of a terminal every minute. The name comes from the Heathkit Heath-19 terminal, which supports a status line. Other terminal types are supported, including VT100. Depending on options, *hclock* will put the time, date, and load average in the status line. When new mail arrives, the word "mail" appears in the status line and the bell is sounded. Writing to the status line is accomplished with escape sequences. Running *hclock* with the *-C* option is the most efficient for the desk accessory, sending one character (bell) only when new mail arrives; nothing else is written to the terminal.

This accessory makes two assumptions. First is that the host is connected through the modem serial port in the back of the Macintosh. Second is that the user has logged in to the host with some sort of host-Mac program, such as *MacTerm*, *Red Rider*, or the mail program discussed above, and has started *hclock*. This sets up the baud rate for the serial port; the default baud rate for a Macintosh at power-up is usually 9600 bps.

SUMMARY

This proved to be a very interesting experiment in user-Macintosh and Macintosh-host interfaces. The window and mouse oriented interface is indeed pleasant to use and has features that enhance the usability of the standard UNIX mail system. The Macintosh may perhaps be thought of as an intelligent front-end interface processor to the host. This idea could be expanded to other applications where the Macintosh would handle most user interface chores, leaving tasks requiring other resources to the host processor.

Conversion to a Centralized Mail Service

A Summary

Appendix A

John J Bowe

INTRODUCTION

When the trend in computing has been heading toward decentralization and workstations, why implement a central, department-wide mail server? The strongest reason for this is to enable a single mail service to be available to the entire community of computers. Maintaining the server in a well-known location in the local network domain allows assured access by other hosts. Thus equal mail service is provided throughout the community.

The mail program distributed with Berkeley 4.2 UNIX is designed to allow each host to run a totally independent mail system. The department had been using this configuration until the centralized mail server was installed in August 1985. Usually, a user arranged to have his or her mail forwarded to his primarily used host. This is is not very inconvenient if the number of hosts is small. However, since there are a fair number of workstaions are now in use with more planned, a user may no longer have a "primarily used" host.

Another consideration was offloading the burden of mail handling from the department's primary two hosts, VAX 11/780s. *Sendmail*, the back-end delivery program of Berkeley mail, uses a large amount of CPU time. This problem is further loads the host during peak usage hours. Having a host dedicated to such chores would free CPU cycles on the other hosts for other uses.

There are, however, a few disadvantages to this scheme. Most importantly is fault tolerance. If the computer on which all the department's mail resides suddenly becomes unavailable, all existing mail is inaccessible. With the system distributed by Berkeley, only the mail of the users who primarily use the down host is unavailable. Another disadvantage is that there is now a much greater communication load on the serving host. Every access to a person's mailbox requires inter-process and inter-host communication. This also may take time if the server host is very busy, which may annoy the user. Under normal system load, however, this is not noticeable.

SERVICE vs A SERVER

A goal of this project was to provide a mail *service*, rather than a *server*. A service implies a movable entity, rather than a statically assigned, machine-dependent implementation. Thus, ideally, the service could be transparently moved from host to host. Clients should not have to be concerned with the details of discovering where that service is; the information should be readily available. This is, in fact, how the implementation was done. Each client host has a file telling which host on the local network is the mail server at any particular time. If the server host is scheduled to be taken out of operation for a while, all existing mailboxes could be migrated to another host and the service information on the clients could be adjusted. If the mail serving host crashes unexpectedly, the disk drive containing the mail files could perhaps be moved to another host to act as the mail server.

It is hoped that the idea of having this "service" as an entity available to clients would extend to other services. Possibilities include: news, printing, text processing, and perhaps CPU time itself. As of yet there is no design for this "service server." The interface to the mail service was designed so that the process of finding where the service currently is would be easily changed to fit this more general scheme.

SERVICES IMPLEMENTED

The purpose of this project was to experiment with server-client interaction, not write a mail interface. The goal of the implementation was to keep the appearance of the existing Berkeley UNIX mail program, but to move the resource consuming parts, mostly the *sendmail* program to the server. Thus, the Berkeley program was divided into a front and a back end. All interactions are dependent on TCP/IP and UDP/IP, which UNIX 4.2 (and 4.3) sockets use. Thus, any host which can use this interprocess communication scheme can join the heterogeneous community of mail clients.

There are two basic services provided: sending mail and receiving mail. Both use TCP/IP to assure reliable communication of data. Let us start with sending, since it is simpler. When a client wants to send mail, the front end must rendezvous with the server whose function is to deliver the message to the final destination. With the aid of some smaller destination-dependent delivery programs, the Berkeley program *sendmail* performs this task completely. In fact, the division between client and server was made where the Berkeley mail program invokes *sendmail*. The server simply gets the message to be sent from the client and passes it to *sendmail*.

Since the mail is no longer stored on the local (client) host, another server must exist to fetch a person's mailbox. This server-client division was made where the mailbox is accessed. There are several complications in reading mail. If the user always read his or her mail and then discarded it or saved it to a file on the client host, the server would simply fetch the mailbox and delete it. However, it is often the case that users just want to read their mail, leaving it in their system mailbox or read only some of the messages, deleting them, and leaving the others in the system mailbox. Also, the Berkeley program keeps track of which messages in the mailbox were read, which are old, but unread, and which are new. This requires further interaction between client and server. To complicate matters more, new mail may arrive for the user while the present mail is being read, so the system mailbox will actually contain more messages than the client thinks is there.

When the person is finished reading some number of messages, some of those will be marked for deletion and some for saving. This information is passed to the server to adjust the mailbox. Many times all messages will be marked for deletion in which case the mailbox is removed. This was fairly simple to implement since, fortunately, the Berkeley software had a module which did just this on a local host given the name of the mailbox file and a structure telling which messages to save and which to delete. This structure is built and adjusted by the front end program as the user reads the messages. The structure is therefore passed to the server, which then adjusts the person's system mailbox accordingly by essentially the same module used by the Berkeley program.

OTHER SERVERS AND INTERACTIONS

There are other peripheral servers running which do not interact directly with the user or the front-end mail program. One performs the task of notifying users that they have new mail. Before this mail system was implemented users were notified of new mail by the locally written program *hclock*. This periodically checked the last modified date and time of the person's mailbox (on that host) and sent notification to the terminal. A scheme was devised where a process on the mail serving host would instead send a datagram, via UDP/IP, to the *hclock* process on the local host when the user got a new message. This way no matter where the user is, he or she will be notified of new messages.

In order to assure that the mail serving host is still operating properly, a daemon runs on each client host to listen for datagrams periodically sent by the server. The same daemon performs the analogous function of sending the datagrams on the server. If a client detects that the datagrams are not being sent, it assumes that the server had failed and adjusts the information for the client programs (the mail front end). If another host becomes the mail server, datagrams are sent from the new server to that same daemon, which makes proper note of the new situation.

HOW HAS IT WORKED? A SHORT RETROSPECT

From the user point of view, the mail interface has not changed. Users can indeed read and send mail from any host on the local Ethernet. It is much more convenient to type *mail user* rather than *mail user@host*. Which host a user usually uses is no longer a concern. Fetching a mailbox is not noticeably slower.

An occasional annoyance occurs when the usual mail serving host is down for maintenance. The down time is not long enough to warrant moving the mail service to another host. So for that short time, existing mail is inaccessible. The client hosts continue to run in "local mode", which is exactly the Berkeley distribution implementation.

A serious concern was the data structure representation among the community of homogeneous hosts. In the first phase of implementation, where all the hosts were VAXes, this was not a problem. Other hosts, such as Suns and Apollos, use a different byte order for integers and fill structures differently in memory. This problem was solved by assuring that the structures passed from each host were equivalent. A much better solution is to use some sort of external data representation, such as Sun's XDR in their RPC implementation. This would add a small amount of complexity, but would ensure portability.

FUTURE DIRECTIONS

There are several paths to pursue to improve the present scheme. One would be to simplify the control scheme of which determining and maintaining on which host the service resides. A general "service server" would an elegant way. In general, there are too many servers and daemons running on the server host. A smaller number which performed more functions would be more manageable. Another would be to do the inter-process communication through remote procedure calls, rather than strict client-server rendezvous. This would make the code more readable and more easily portable to hosts that use the same RPC scheme.

Technical Implementation Notes

Appendix B

John J Bowe

INTRODUCTION

There were several major goals in this implementation of a Macintosh interface to UNIX mail. The first was to create a usable software product which people could use to read and send electronic mail. Another was to experiment with Macintosh applications in general. There is a very steep learning curve in unlocking the Macintosh toolbox and understanding how the components of an application work together.

Two independent pieces of software were written. Ideally, the entire mail program would be a desk accessory, which could be used at any time, no matter what application was running. However, an average size for a desk accessory is about 3K of code; Apple claims that a limit to a desk accessory is 8K [Appl85]. This is the major reason that two independent parts of the mail interface were written. The desk accessory simply checks the serial line of the Macintosh for a signal of new mail from the host and asynchronously notifies the person using the Mac when new mail arrives. To access the mail system on the host, the user would then invoke a separate mail interface program.

The longer of the two programs by far, is the front end mail interface. This is invoked as any other Macintosh application. With this one can compose messages, read them from a UNIX host, send them, and save them in Macintosh files in a window-oriented, friendly environment. To compose and create files, the Mac need not be connected to a host. The communication with the host is transparent to the user.

DEVELOPMENT ENVIRONMENTS

A different development environment was used for each of the two programs. At the time of the start of implementation there was only one choice within the department: the SUMacC (Stanford University Macintosh C) compiler and *rmaker*, the resource compiler. Programs are written in *C* on a UNIX host and compiled and linked with the SUMacC package (*cc68* and *ld68*); the SUMacC package was written at Stanford University. The code module is combined with resource information by *rmaker* into a *resource file*, which is transferred to a Macintosh running MacTerm by *macput* on the UNIX host. MacTerm automatically detects when a file is being transferred to it. The protocol used for the transfer is XMODEM. The advantage to this is that the compilation is quite fast on a large host. Also, facilities on the host such as the text editors, *make*, and *grep* are very convenient. It takes a lot of time, however, to transfer a large program to the Macintosh.

The desk accessory was written with Manx Aztec C. This runs totally on a Macintosh. There were several reasons for using this. The first is that it is easier to write a desk accessory in this environment. A program is provided to easily install one into the system file of the Macintosh diskette. I also wanted to experiment with another development environment. The slowness of the Macintosh disk access was not a burden since the desk accessory code itself is small, and, in fact, resides in a single file. Further development tools are necessary if more users wish to write their own Macintosh applications.

WINDOW DISPLAYS

All window functions are performed using the standard Macintosh user interface guidelines outlined in *Inside Macintosh* [Appl85].

There are three main windows, not counting dialogs. A *from* window to show from whom one's mail has come, a *read* window to display a mail message, and a *compose* window in which one composes a message. All three are created by the same procedure, but handled slightly differently when manipulated by the user.

The *read* and *compose* windows behave exactly the same until closed. Text can be typed, selected, cut, copied, or pasted. When in these windows, the cursor is the standard I-beam to denote this. Upon closing the user is prompted for what he wants done with the window by a simple dialog box. In both cases the options are to be saved to a file, destroyed, or no operation. The *compose* window has the option to send the message.

The *from* window contents can not be changed. The cursor remains an arrow. When the mouse is clicked, an entire line is selected and the Read option of the Mail menu is enabled. The line selection is done by looking forward and backward for carriage return characters from the point selected in the text and explicitly setting the selection range in the TextEdit *record* for that window. From the array of linestarts in the TextEdit record, the program knows which line was selected, thus knows which message the user wants to read.

HOST COMMUNICATIONS

There are several major reasons to communicate with the host. The first is to log in or log out. The user's login ID and password are kept as string resources in the application itself. The standard UNIX login procedure is assumed and the program recognizes when the host is prompting for the parts of the login procedure.

A necessary task is to obtain a list of from whom messages have come. The program *fetch* was written to fetch messages from a user's mailbox on the UNIX host. *Fetch -f* behaves similarly to UNIX *from* command, but uses less than half the CPU time. The Macintosh puts this information directly into a window for the user.

To speed retrieval of messages from the host, *fetch*, unlike *mail*, fetches a message from the user's mailbox quickly with no further command interaction. *Fetch* is compatible with both the standard Berkeley mail system and the Brown-enhanced mail program. The command is issued to the host through the serial port and the mail message is returned. The message asked for should never not be there since the Macintosh had just previously asked the host for a list of the available messages. A user could fool the Macintosh program by deleting some messages at a terminal. This is no problem, however, since *fetch* will simply return nothing, including no error message.

The communication process of sending mail is similar to reading. The Macintosh invokes *sendmail*, the back-end delivery program, on the UNIX host and simply sends the message typed into a *compose* window through the serial line. *Sendmail* is called with the -t flag to get the recipient from the header of the message rather than the command line.

DESK ACCESSORY

A Macintosh desk accessory is a specialized form of a driver. A driver has entry points depending on what function is to be performed. Functions include: open, close, status, prime (usually read and write), and control. It is loaded into the system file as a resource and is of type DRVR (driver).

The mail checking desk accessory simply checks the serial line for information about new mail. Thus, there must be two entry points. One, obviously, is open. The other is to do the actual checking of the serial line. This is the control entry point. In this desk accessory it is invoked every five seconds. To specify the time, there is a field of a table at the beginning of the code of the desk accessory that specifies how often (in system clicks, 1/60ths of a second) the accessory is given control. Other information specified in this table is which events (mouse, keyboard, etc) will be handled, the menu ID of the menu that the accessory will use. if any, and a list of entry points the the functions listed above. The actual checking is done by seeing if there are any characters available at the serial port and if there are read them. If one is a bell (07 hex) then then the desk accessory notifies the user of new mail. It is assumed that there is a program running on the UNIX host that will send a bell when new mail arrives. *Hclock* with the *-C* option will do just this.

REFERENCES

[Appl85] Apple Computer, *Inside Macintosh*, 1985.

[Cher85] Chernicoff, Stephen, *Macintosh Revealed,* Volumes I and II, Hayden Book Company, 1985.

[Shoe83] Schoens, Kurt, "Mail Reference Manual", Version 2.18, UNIX 4.2BSD source tape: /usr/doc/Mail, 1983.

[Smit85] Smith, Michael G., "Ther New Mail System Overview", Brown University Department of Computer Science, MA-6-85, August 1985.

[Sun85] Sun Microsystems, "Remote Procedure Call Programming Guide", Release 2.0, 1985.

[Leff84] Leffler, Samuel J., Fabry, Robert S., Joy, William N., "A 4.2BSD Interprocess Communication Primer", Department of Electrical Engineering and Computer Science, University of California at Berkeley, October 1984.

END

DTIC

7-86