IT- AN ACTIVE MESSAGE EXTENSION TO THE UNIX™ MAILX COMMAND

by

Ronald Richard Dailey

B.S. Applied Math, University of Colorado, 1970

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

Kansas State University
Manhattan, Kansas

1988

Approved by:

Richard A. McBride

Richard A. McBride

CONTENTS

List of Figures

.

TRADEMARKS

UNIX is a trademark of AT&T

Informix is a trademakr of INFORMIX Software Inc.

Dedicated

To my Mother and to my sister, Myrna, who watched my two children, Mitchell and Tania, for many weeks each summer, while I worked on this degree. Also to my brother-in-law, Rod, who put up with tickets as he cared for my barking dog. To Mitchell, Tania, and my new wife Caryn. Finally to Dr. Richard A. McBride for his unflagging enthusiasm, encouragement, help and dedication to the very end.

1 Introduction

1.1 The Computer Based Message System Today

The Computer Based Message System (CBMS) has literally exploded into the life of the business employee during the past 10 years. One author identified nearly 300,000 paying customers on three currently available commercial systems [PCWORLD 88]. The advent of the low cost personal computer is bringing about the era of mass messaging [Caswell 86]. One reason for the popularity of CBMS is the wide variety of services provided to the user.

More than one CBMS available today provides the user with an active User Agent (UA). An active UA performs certain predetermined actions on the mail message for the user automatically. Examples include categorizing messages, prioritizing messages according to importance, and contacting the CBMS periodically (from a PC) looking for new messages.

It is proposed that a rarely offered, but flexible and valuable addition to the services provided by a CBMS is that of an active message. An active message contains instructions which are activated when the message is selected for viewing by the recipient.

A CBMS which supports active messages provides a structure for a cooperating recipient of an active message to quickly and efficiently perform tasks requested by the originator. Nearly any programmable task may be sent as an active message, but such tools are typically mundane chores which are important to accomplish in a timely manner. Two examples of tools which could be supported by active messages are calendar management (scheduling meetings) and form handling.

1.2 Outline of the Report

Chapter 2 explores the status of the CBMS today: the standards that allowed CBMS to develop, the classes of service provided by the large number of currently available systems, the security issues, and the legal issues. Chapter 3 describes an extension of the UNIX™ System V mailx command to accommodate active messages which take the form of questionnaires. Future development possibilities of active messages from a privacy viewpoint are discussed in Chapter 4. A perspective on the system described in Chapter 3 is provided in Chapter 5 with a discussion of possible extensions to the work.

2 Status of Computer-Based Message Systems

2.1 Standards

2.1.1 The Need for Standard Standards

More than 200 programs for communication between personal computers and mainframes and 100 general communications programs were available for personal computers in 1986 [Caswell 86]. Lack of standards during their development have lead to inconsistencies and inability to communicate.

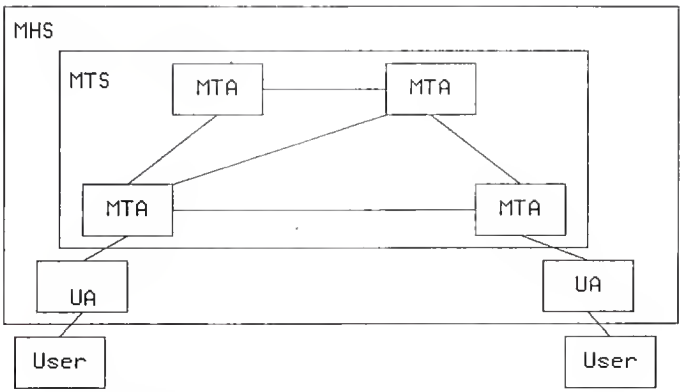
The National 8ureau of Standards identified a need for and issued standards in 1979 [N8S 79]. IFIP Working Group 6.5 held a series of annual conferences from beginning in 1981 on the requirements of CBMS [Smith 84]. However, the most widely accepted standards in use today are the CCITT X.400 [X.400 84]. The chairman of the "Second International Symposium on Computer Messages Systems" (held in September of 1985) suggested it might have been titled "X.400 Comes of Age" [Uhlig 85]. The papers presented at this conference illustrate the benefits of having a standard and at the same time identify the need for additions to the standard.

2.1.2 X.400 - The Standard Standard

The CCITT X.400 protocol [X.400] uses the Open System Interconnection (OSI) reference model (Application Layer) as a basis for a Mail Handling System (MHS). The author of this paper uses the terms CBMS and MHS interchangeably. Recommendations X.401-X.430 detail the protocols, algorithms, service elements, notations, and other information necessary to thoroughly define X.400.

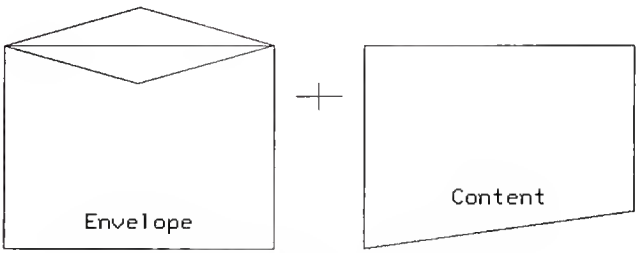
The functional view of the X.400 MHS model is shown in figure 2.1. A user is a person (or computer application) who is a message originator or a message recipient. The User Agent (UA) is a program; it helps the user prepare and receive messages which are transferred by a Message Transfer Agent (MTA). An MTA is responsible for delivering messages to another MTA or to recipient UA's. A Message Transfer System (MTS) consists of one or more MTAs. The UAs and MTAs together comprise the MHS.

The basic message structure is illustrated in figure 2.2. The envelope carries the information necessary for transferring the message from the UA of origin to the destination UA. The content is the actual data which is sent.



Functional View of a Message Handling System

Figure 2.1



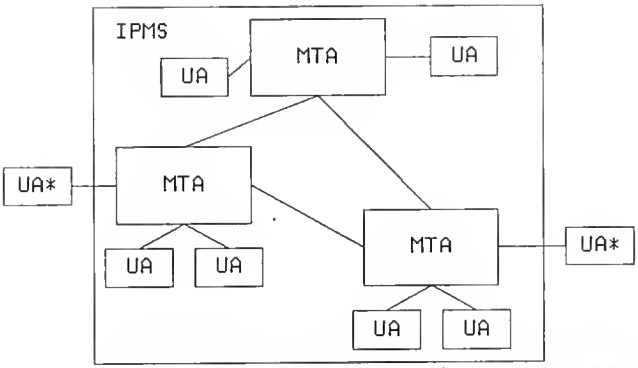
Basic Message Structure

Figure 2.2

The MTS provides store-and-forward transfer service (i.e. if it is not possible to immediately forward a message, it is stored till transfer is possible). The originating UA delivers the message to the MTA. The MTA routes the message (with the assistance of other MTAs as necessary) to the residence of the destination UA, where the message is transferred to that UA.

Different classes of UAs may exist. A class of UAs (called cooperating UAs) may provide non-standardized functions (called local UA functions) to provide enhanced service for their users. One class of cooperating UAs is called an Interpersonal Messaging System (IPMS). Another class of cooperating UAs might be used by a distributed database system to relay transactions.

An IPMS provides individuals with Interpersonal Messaging (IPM) service, via a class of cooperating UAs within the MHS. Figure 2.3 illustrates the X.400 IPMS. The content of the messages transferred within an IPM contains heading information and body information. The body may contain text, graphics, voice, and other kinds of data. The UA knows how to handle the body of the message based on the heading.



* This is a UA which uses the MTS, but cannot communicate meaningfully with the other UA's.

Inter-Personal Messaging System

Figure 2.3

This completes the discussion of the pertinent portions of the X.400 standard. It is interesting to note the concept of a mailbox is not a part of this standard. At least one author [Wilson 84] considers this a short-coming, and re-introduces the concept within the UA.

2.2 Varieties of CBMS

2.2.1 Commercially Available CBMS

The purpose of this section is to illustrate the widespread acceptance of CBMSs and to identify some of their important features. This is accomplished by pragmatically surveying several commercial mail systems.

Numerous mail services are available to anyone who has access to a multi-user computer or a single-user computer with access to other computers. One current article identifies "binmail", "Berkeley Mail", "MH", "Elm", "AT&T Mail", "Rmail", "MH-Rmail", "MM", "snd/msg", "VuMail", "IMS", "uumail", "nmail", "dmail", and "fmail" as currently available mail systems which work under UNIX [Taylor 87]. EasyLink, MCI Mail, and AT&T Mail are compared in current literature [PCWORLD 88]. Recapitulation of their base of use and features is illustrative of their popularity and of what users (presumably) find useful.

The base of use of the mail services described varies from 20,000 to 170,000 users. The users send an average of about 20-50 messages per month. Some services will swap messages with users in foreign

countries, but won't necessarily swap messages with domestic competitors. All of them will deliver electronically to their own subscribers, telex, and paper mail. The more sophisticated ones will deliver to fax machines, to voice synthesizers (accessible via phone), to next day courier, and to four hour courier.

Messages transportable range from only plain text files in the simplest system, to any binary file (including spreadsheets and formatted documents) in the more sophisticated. In the simplest system, the user must know the exact electronic address of the recipient. In more sophisticated systems a user may use a directory service to identify the user based upon name, company, and/or other identification not related to the mail system. Messages may be dispatched to one or multiple recipients. On some systems a central collection of messages on a topic may be established. Users may join this "conference" to read and add their own messages.

The most friendly services provide an interface program for a personal computer (PC). This allows the PC to perform various tasks for the user including (in one case) calling the mail service periodically to check for new mail.

This author finds the range of services available and number of paying users very impressive. However, he does not find any functionality that could be considered that of an active message. However, automatically contacting the CBMS by the PC is clearly the function of an active UA.

2.2.2 Conferencing Systems

The CBMS functionality discussed so far has been in the context of an originator sending mail to a recipient based on the recipient's electronic address. An originator could also have sent mail to a group of recipients via a distribution list of electronic addresses.

Another facility which is commonly available, but not defined in X.400, is a conferencing system (or "bulletin board"). Characteristics of this distinct form of MHS are [Kille 84]:

1) Messages are stored in a common area, called a conference, accessible by many users as a system resource.

2) Messages related to a single topic are stored in a single conference.

3) A user (recipient?) subscribes to one or more conferences to gain access to the messages stored there.

4) Messages are kept in the conference for an extended period of time allowing for the review of earlier entries.

5) Messages are grouped into sets concerning a particular subject within a topic.

The significance of conferencing systems is illustrated by the fact one of the books [Waite 87] and at least one of the papers referenced [Wilson 84] were developed by their authors almost exclusively through the use of conferencing systems (USENET and Blend respectively). Another significant use of conferencing systems is technical support of computer software (e.g. Ashton-Tate offering support through CompuServe). Perhaps the most surprising statistic reported was the 5 to 1 ratio of conference messages read to personal messages read on a Multics computer [Huitema 85].

The conference may reside on a single host, or be distributed across numerous hosts. USENET is a widely used conferencing system with numerous hosts in the United States and Europe [Henderson 87]. "COSAC" is a conferencing system which migrated from a single host to distributed hosts. The distributed host environment accentuated problems already present in the single host environment and presented a few problems specific to distributed hosts. One problem was formatting a message to be conveniently read by all users, despite the varied (and unknown) terminal types that might be used. Another problem was determining on which nodes to store the conferences, especially if a node only had one user of a conference. Despite the problems, records indicate the distributed host environment became the conference of choice within a few weeks [Huitema 85].

Recognizing the need to accommodate conferencing systems in the X.400 standard, one author [Palme 85] drafted a proposal to add an entity called a Distribution Agent (DA). The DA would be in communication with the MTA and be the originator and recipient of messages. One DA would exist for each conference. The DA could resend messages to registered users and/or store messages for those users.

There is no reason an active message should not work with a conference (DA). An interactive questionnaire residing on a special interest conference could be an ideal way to question interested parties about a matter of group concern.

2.2.3 TELEX, Teletex, and Facsimile

As indicated earlier in this document, some commercial CBMSs provide delivery to fax machines, many provide delivery to telex or teletex machines. The X.400 standard supports conversion to G3 Fax, telex, and teletex formats from text and other formats. In the past, telex and teletex have been the most widely used forms of a store-and-forward CBMS. It would be interesting to compare the percentage of messages delivered to telex or teletex versus those delivered to a PC five years hence.

These devices are not suitable for the processing of active messages.

2.2.4 Videotex

Videotex is computer conferencing and messaging system for the masses. In Austria, any home with a telephone and a TV may be connected to a videotex service with the addition of some electronic equipment. In 1984 it was estimated the Federal Republic of Germany would have over 1 million videotex subscribers by 1987. [Jaburek 84]

By use of the telephone, a videotex subscriber contacts the videotex host and creates a message (with special equipment) or directs a message be displayed on his TV. The telephone time consumed can make message creation an expensive task. Messages may not be sent to more than one recipient without again entering the message or at least parts of it.

The importance of videotex as a CBMS is in the masses of people that are subscribed to it. While videotex is supported under the X.400 standard, there is no special functionality associated with it beyond its input and output media.

This author does not interpret any functionality of videotex as described by [Jaburek 84] to be that of an active UA or an active message.

2.2.5 Voice Mail

Voice mail (store-and-forward) is supported by X.400 in two forms:

1) input (from a phone call) is digitized, stored, and retrieved upon recipient request.

2) a textual message is received and stored, then synthesized to voice upon the recipient's phone request.

One author [Gitman 85] identifies speech-to-text conversion as in its early stages and anticipates this capability being added to commercial CBMSs in the future.

Current features of voice mail include mailbox service (where the originator prepares messages in his own mailbox to be delivered to one or more recipients), call answering services (where the originator or recipient specifies a message is to be stored in the

recipient's mailbox), and advanced services such as speech-to-text conversion, audio files (to archive and file voice messages similar to textual messages), integrated voice and information processing (where a voice message is sent and automatically converted to text to provide written confirmation), and others.

The UA which presents a voice message may interact with the recipient to play the message faster or slower, or repeat a message, or some similar function. Since this is done entirely at the direction of the recipient, this author does not interpret the UA which presents current voice messages to be an active UA. As the functionality of the system is not dependent on the content of the message, the message is not interpreted to be active either.

2.2.6 Multi-Media Mail

Multimedia mail messages may contain any combination of text, images (including facsimile and graphics), and voice. The presentation of a message might be similar to a slide show with voice accompaniment. While X.400 supports each component of multimedia mail, it does not support a single message containing all components. However,

several multimedia messaging systems have been implemented. One such system is the Multimedia Mail Handler (MMH) developed at the University of Southern California [Postel 88].

The Multimedia Mail Transfer Protocol (MMTP) was developed to define the format and procedures for transfer of messages sent under MMH. One component of the MMTP is the Multimedia Mail Content Protocol (MMCP) by which the format of the document is defined. The messages are created and displayed by a User Interface Program (UIP) which is similar in function to a UA. Different UIPs were experimented with to determine effective ways to create multimedia messages. The messages are transferred by a Message Processing Module (MPM) which is similar in function to a MTA.

The content of multimedia mail introduces several problems. The terminal must support all components of the message. The message (and therefore the transmission time) may be quite large anytime voice or graphics information is included. If color or gray-scale graphics are to be supported, the messages sizes would grow. To make the transmission time reasonable, the terminals using MMTP are connected to a host over a LAN (DARPA Internet and Ethernet).

Defining the structure of a multimedia message addresses the problem of where in the body of the message to place the various components and how to coordinate the presentation of the message. Components may be presented independently, sequentially, or simultaneously, and this information must be contained in the message.

Preparing the message also presents problems. While there are numerous programs for manipulating text and graphics, editing voice messages is a particular problem. The MMH developers allowed the voice to be displayed graphically as an energy waveform. Using a mouse, the user can select any point in the message, and play from that point on. After identifying what portions of the graph corresponds to what sounds, the mouse can be used to highlight portions of the graph (and therefore the voice) to be saved or deleted. Silence or previously saved portions of voice may be inserted.

This author does not interpret the presenting UA in MMTP to be active, nor does he interpret the message to be active. However, the UA is required to look at the body of the message to determine the proper way to present the message. An active mail system similarly looks at the body determining the mode of presentation.

2.2.7 Active Mail

2.2.7.1 R2D2

An active message has been characterized as a "message with a mission that the message knows about" [Vittal 81]. The same author developed the Research-to-Development-Tool (R2D2) system to explore active messaging.

The goals of R2D2 were to:

- allow messages which are executable.
- allow originators and recipients to specify the actions of the message.
- allow users to tailor the functions and functionality of the CBMS.
- allow users to tailor the interaction style of the system.

Examples of the functionality that might be added by users to the framework provided by R2D2 are questionnaires, calendar management, and purchase order processing.

Questionnaires require the UA to present the message one question at a time, gather the response, and tally the results after some cutoff time. Calendar management is a more difficult task requiring system support in the form of a database of schedules in addition to the messaging tools of R2D2. Purchase order processing was conceptualized as a person sending out a purchase order message to the system and the system identifying the appropriate recipients to approve the order based on the value as contained in the message. Evaluating the content of the message and determining the recipients based on content in conjunction with organizational rules is a much more difficult problem.

This author identifies the R2D2 questionnaire to be an active message, the calendar to be an active UA (possibly with an active message depending upon implementation), and the PO processor to be an active UA.

2.2.7.2 Imail

An active mail system called imail (intelligent mail) has been developed at the University of Toronto [Hogg 84]. As opposed to passive messages, imail uses an intelligent message called an imessage. Normal messages are read, an imessage performs tasks. The task might be to collect answers from a user and/or to dynamically route itself to other users based on answers received.

Having been developed under UNIX, imail is similar to UNIX mail. An imessage is sent to a recipient, but the imessage is still owned by the originator. All actions related to the processing of the imessage are recorded in a history field. After a recipient interacts with the imessage, the imessage is returned to the originator with the responses. If an imessage is deleted from the imailbox, an indication is sent to the originator. If the imessage is not replied to within a specified deadline, it is removed from the imailbox and an indication is returned to the originator. A copy of the imessage may be sent to someone else, and this imessage has the same traits as the original.

The ability of an imessage to send copies of itself to other recipients presents difficult coordination problems of these distributed processes. How are all the processes located and communicated with when they are possibly in unknown and changing locations? How does one know if one has received all responses to an imessage? How does one terminate all children of an imessage?

Clearly imail is an example of the implementation of active messages. It is noted imail uses an active UA to delete any imessage which has exceeded time parameters.

2.2.7.3 ETC

The Elapsed Time Communication system (ETC, pronounced etcetera) [Gold 86] deploys envoys instead of sending messages. An originator deploys envoys which contains the original message, knowledge of the original message, knowledge of the sender, and knowledge of possible recipients.

An envoy is a delegated representative of the originator with adequate knowledge to perform a specific task. An envoy is different from a message as follows:

1) The original destination of the envoy may be extended to include other destinations based upon post-deployment activities. For example, an envoy might be deployed to offer free kittens to "anyone who likes cats". These recipients might forward the message on to their friends.

2) The envoy might interact with the recipient's UA (rather than the recipient) to gather information. For example, the kitten envoy might be routed to an individual with too many cats already. If this can be determined from the user agent, this message would not be delivered to this recipient.

3) The envoy can be modified as it visits its various destinations. For example, the originator of the kitten envoy may only have had 10 kittens. As the recipients reserve kittens for themselves, the envoy continues with the reduced number available.

4) The envoy usually returns to the originator to report what destinations were visited and what activities transpired. For example, the kitten envoy would finally return to the originator

when all kittens were given away or there are no more recipients. It could list all of the individuals visited and who reserved a kitten.

An envoy system requires supporting software beyond that provided by X.400. The following software was provided to make ETC work:

1) A destination reasoning system to look at information in an envoy to determine destinations. (If an envoy is posted to "anyone who likes cats", this would identify specific recipients.)

2) A communication reasoning system to allow functions such as automatic copies of messages to other individuals based on subject matter and importance.

3) An envoy reception system to determine whether or not to accept the envoy or send it to the next recipient.

4) A communication control language is used to define what actions the envoy will request to be performed on the originators behalf by the recipient's UA.

5) A specialized reasoning system to allow functions not covered by the above. An envoy requesting addresses of other users with similar interests would need the specialized reasoning system to clarify the meaning of "similar interests".

This work is very similar to the previously discussed imail system. A major difference is the concept of the message having knowledge of itself and the sender and being able to modify itself., as opposed to merely being executable. This is clearly an example of an active mail system, and also makes use of active UA's.

2.2.7.4 Information Lens

Another mail system called "Information LENS" has been developed at MIT [Malone 87]. The key features here are a very friendly UA (which can be customized by individual users) working with semi-structured messages through the use of an interactive graphic editor.

Semi-structured messages are classified into a hierarchical tree of types with the root of the tree being the least structured message, as it contains only one required field. As the tree is descended, each class of types inherits the required fields in the parent class, plus adds required fields of its own.

During message creation, different screens are provided for each type of message with specific areas for the required fields. Not only does this facilitate the creation of the message, but the same information is found in the same place on the same type of message.

Each message in the system carries a type identifier, which allows the UA to filter, sort, and prioritize the messages easily. When a recipient chooses to answer a message of a particular type, he is presented with a choice of appropriate message types with which to answer. Selecting one of them presents a creation screen with some of the required fields populated with information from the original message.

Extensions to this system have already been added to provide computer conferencing, calendar management, and project management and task tracking. Further expansion of the capabilities of the system are facilitated by allowing a UA to identify the name of a

lisp program to activate if a received message is of a specified type and the required fields contain information meeting specified criteria.

Information Lens is an example of a CBMS which contains active UAs. The messages are not active, but they do contain information structured adequately so the UA may act upon them based on the content of the required fields. Provision has been made to extend the UA to activate the body of the message.

2.3 Security Issues

2.3.1 Protecting the Message

It is important to protect the privacy of messages sent through a CBMS for commercial, personal and other reasons. The messages may contain a wealth of information that would give economic, technical, or other advantage to an adversary.

Administrative issues are the first line of defense against unauthorized access. Effective procedures must be established to ensure only authorized users are on the system and that they can not

easily access areas of the system for which they have no authority. Additionally, auditing procedures should be established to ensure no user has inadvertently opened a hole into the system. [Stoll 88] describes tracking a hacker who, over a 10 month period, attacked 450 computers, found 220 available for access, gained user authority on 30, and gained system-administrator authority on 4. No operating system was more vulnerable than any other, rather:

> "Most of these break-ins were possible because the intruder exploited common blunders by vendors, users, and system managers."

The hacker described found the names of new systems and login scripts (in addition to other information) by scanning mail messages of careless users.

Additional protection of the content of a message can be provided through the use of encryption. The effectiveness of ciphers hinges on the method for carrying out the distribution of keys to appropriate users. [Rose 85] proposes the concept of a Trusted Mail Agent (TMA) which is responsible for encrypting and decrypting messages based on keys received from a Key Distribution Center (KDC).

Under this plan, when an originator posts a message, the UA delivers the message to a TMA instead of a MTA. The TMA accepts the message, identifies the originator and recipient, and based on this information requests a key (over a secure communication system) from the KDC. The message is then encrypted and returned to the UA who delivers the encrypted message to the MTA.

Upon receipt of the message from the MTA, the recipient UA identifies that this is an encrypted message. The TMA is activated to decrypt the message, contacting the KDC for the proper decryption key. The message is then returned to the UA for presentation to the user.

The UA and TMA ideally reside on a work station separate from the residence of the MTA.

The author is (intentionally?) vague about the establishment and workings of the secure communication system between the various TMAs and the KDC in this commercially available system. Gaining access to the KDC would jeopardize every message sent on the system.

Gaining access to the communications between the TMA and KDC is equivalent to having full access to all messages in or out of that UA.

This model for security is of particular interest to this author because of the activation of the TMA by the UA based upon the fact a message has been identified as encrypted. This is interpreted to be an example of an active UA, but not an active message.

2.3.2 Protecting the Environment

Receipt of messages provides the potential for unauthorized activities on the recipient machine, and possibly in a MTA enroute.

A common instance of this phenomena is downloading a program from a bulletin board and executing it. There is no guarantee that the program received will do what it claims (i.e. it might cause unintentional mischief) or do only what it says (it might additionally remove data or programs, modify data or programs, or create new programs to be unknowingly executed later).

Of special interest is active mail messages. [Hogg 84] states:

> "Imail is much more powerful than conventional mail and therefore potentially far more dangerous. An imessage is a program whose actions are not known by the recipient, and there is clearly a great opportunity for electronic guerrillas to throw letter bombs that delete all the recipients files."

This author suggests the potential for harm is much greater than specified by [Hogg 84]. Trojan horses and viruses could be planted with the potential for far more serious consequences than deleting a few files. This author contends that without an appropriate implementation in a sufficiently secure environment, active messages are too dangerous to be used or marketed.

2.4 Legal Issues

Certain legal concepts and precedents have been established for making contracts over the phone or when using postal mail. Questions concerning their applicability to electronic messages are

addressed in this section. No attempt is made to answer the questions raised, the intent is to clarify which legal issues are applicable as specified by Newman [Newman 84].

The phrase Computer Mail System (CMS) refers both to Single-Host Mail Systems (SHMS) and to Network Mail Systems (NMS), which terms are assumed to be self-explanatory. There are five classes of problems relating to messages delivered over a CMS.

1) What is meant by the terms "posting", "delivery", and "receipt" with respect to a CMS?

Posting refers to the act of sending (relinquishing control of) the message. Unlike postal mail, on a CMS it is often possible to delete a message after posting. What if the originator believes he has successfully deleted a message after posting, only to find later a copy had already been sent?

Delivery may be to a store of messages, or to a user agent. Is delivery accomplished when the transfer takes place or when the recipient is notified of the arrival? Is the system message "You

have mail" adequate notification of arrival, or must the recipient actually see header information identifying that a message from a particular originator has arrived?

Acceptance is very murky in the electronic world. Consider: Smith offers a contract to Jones via a CMS, and shortly thereafter retracts the contract in a second message. The next day Jones reads the first message, and replies with acceptance of the contract. Shortly thereafter he reads the second message rescinding the offer. Is there a contract? Some argue no, as the offer was rescinded before acceptance. Others argue if Smith reads the acceptance before Jones reads the withdrawal, then there is a binding contract.

Consider: Jones is writing the acceptance of the offer at the same time Smith is writing the withdrawal. Both parties post the message at the same instance. Is there a contract? Some argue no, as there was no consensus. Others argue yes, by using a CMS with these known problem, Jones has implicitly assumed the risk of this event.

Consider: Jones' acceptance message is garbled in transmission. Smith thinks no contract has been completed and sells the 10,000 pork bellies (originally offered to Jones) to someone else. Jones (who saw a return receipt provided by the message system when Smith

attempted to read the garbled message) thinks there is a contract and also resells the pork bellies (he thinks he just bought). Who owns the pork bellies? If this is a NMS, the question might become, whose equipment garbled the message? If the communication line between the two machines is at fault, is the responsibility with the communication company?

2) What is meant by "place of occurrence" with respect to a CMS, especially when a user has "remote access"?

In the examples cited above there might be legal action. The location where the contract was made determines the court with jurisdiction over the matter. Is this the office of one of the parties involved? If delivery is to a mailbox in a computer in a third location, perhaps that is the jurisdiction.

3) What is meant by "instantaneous communication"?

In the examples above, at least some of the problems could have been avoided by a face-to-face meeting, or other instantaneous communication. Under what conditions is store-and-forward communication the same as instantaneous communication?

4) Who "owns" a message? What are the implications if it is modified after posting by someone other than the owner?

Does the system designer have a legal liability to prevent unauthorized modification?

5) What are the implications of an active non-human agent "handling", "filtering", and possibly generating answers for a user?

Does the system designer have a legal responsibility when designing an active message system to ensure messages will not be misfiled or thrown away incorrectly and to ensure an answer (such as a return-receipt) will not be generated improperly?

The crucial tests are determination of the intentions of the parties, sound business judgment, and judgement of where the risks should lie.

3 IT- An Active Message Extension

3.I Introduction

The active mail systems described previously have demonstrated some of the potential of active messages. This chapter describes an extension to the UNIX System V mailx command which identifies and processes active messages. The extension is called IT for Identify and Transact.

IT (pronounced "it") consists of modifications to the mailx command to allow the program and data associated with an active message to be identified upon selection by the recipient. The message body contains an identifier that this is an active message, the name of the program to activate, and input data required by the program.

Different tools (or programs) can be developed to be activated by IT, thus providing flexibility in IT's use. The first tool developed to work with IT is questionnaire mail, called qmail. qmail allows complex questionnaires to be sent as active messages which interactively solicit responses from a recipient. The

recipient's response to the questionnaire becomes an active message which automatically loads a user's responses to the questionnaire into a database for later analysis.

The remainder of this chapter describes questionnaires, qmail, and the implementation of qmail as a tool for IT.

3.2 Background on questionnaires

The purpose of this section is to familiarize the reader with the use of questionnaires and how they work. The information and terminology given should be adequate to understand the basis for qmail.

A questionnaire is a "formalized schedule for collecting data from respondents" [Kinnear 83]. There are five parts to most questionnaires: identification data, request to cooperate, instructions, questions, and classification data.

The questions found on a questionnaire fall into three categories: open ended (which allow responses with few constraints), dichotomous (which allow yes/no or agree/disagree type responses), and multiple

choice (where the respondent picks from a choice of responses). Each category of question has advantages and disadvantages associated with it.

For example, open ended questions are frequently difficult to process with computers, but give more accurate data as the choices are not restricted. On the other hand dichotomous and multiple choice questions are easy to process, but restrict the choice of answers, and possibly the accuracy of the data collected. In both cases, careful design of the questionnaire is critical to successful collection of data.

Responses may be requested for all questions on a questionnaire, or just a subset relevant to the respondent. The "question flow" is the order in which the questions are asked. The response to one question may determine what question is asked next. For example, if the first question is:

"Have you eaten at a fast food outlet in the last month?"

and the response is "no", then the next question asked might be:

"Have you ever eaten in a fast food outlet?"

If the original response had been "yes", then the next question might be:

"At which of the following outlets have you eaten in the last month?"

Obviously, on a printed questionnaire, directions must be given to guide the respondent through the questions.

The questionnaire can be modeled as a state diagram. The first question or statement is the start state. Each possible response corresponds to a state transition to another question (state). With a **qmail** questionnaire, question number 0 represents the final question (halt state), all paths through the questionnaire eventually end at question 0. Figure 3.1 is a state diagram for the sample questionnaire discussed in the next section.

Once the responses to a questionnaire have been collected, the data collected must be translated into information from which decisions can be made. If there is a large volume of responses, then frequently the only way to handle this data is with a computer.

Accurate interpretation and entry of the data received is crucial to obtaining accurate information. Depending on the subject, questions

which are open ended leave a great deal of latitude for interpretation of the respondents meaning. If the question was:

"What did you like about your Big Burger experience?

and the response was:

"Its a good place to eat."

then less data is collected than from the use of a multiple choice question enumerating qualities such as cleanliness, friendliness, quality of food, . . .

Clearly, careful design of the types of questions, the flow of the questions, the questions themselves, and presentation, are factors in preparing a questionnaire which will provide accurate data.

3.3 A Sample Questionnaire

The following simple questionnaire was adapted for testing the qmail tool and to illustrate some of the issues of building a tool to actively present questionnaires to a recipient. Only the question and response portion of the questionnaire is shown.

1) Have you eaten at a fast food outlet in the last month?

   Yes Continue at question 4
   No

2) Have you eaten at a fast food outlet in the last year?

   Yes
   No Please skip to the end of this questionnaire.

3) How many times have you eaten at a fast food outlet in the last year?

   ____ Enter an estimate

4) Which of the following outlets have you eaten at in the last month?

   ____ Big Burger Hamburger Emporium
   ____ Super Chicken Poultry House
   ____ Kung Foo Chinese Food

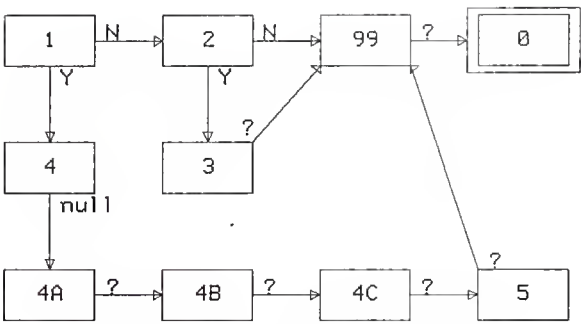
5) What statement best describes your opinion of fast food outlets?

   a) They are ok if you are desperate.
   b) They are ok for a fast lunch.
   c) They are a great change of pace for supper.
   d) They are great, I eat there on a regular basis.

Thank you for your time. Please return the questionnaire in the attached envelope.

Figure 3.1 illustrates this questionnaire as a state diagram.

Question 1 is clearly a dichotomous question. It may also be better to send the positive answers to 3) rather than to 4). Note the necessity to have instructions for the respondent indicating the next question. These instructions would not be necessary if the question was presented actively, and the response analyzed. In figure 3.1 state 1 corresponds to question 1, and the flow is split between questions (states) 2 and 4.



A Questionnaire As A State Diagram

Figure 3.1

Question 2 is also a dichotomous question. In this case the question flow may be terminated. In the state diagram, the flow is split between question (state) 3 and state ggg. State ggg represents the thank you message at the end of the questionnaire.

Question 3 is an open ended question. The flow of the questions could be changed here by analyzing the number entered. That is not deemed necessary for this questionnaire. In figure 3.1 the "?" indicates any response here will cause transition to state ggg.

Question 4 is not to be directly answered. Following it are three related dichotomous questions which do require answers. It would be undesirable to split the flow of questions based on answers to these dichotomous questions. Question 4 and each dichotomous question following should be presented on the same screen during an active presentation. In figure 3.1 question 4 is reflected by state 4. Note that null input causes flow to the next state. Questions 4A, 4B, and 4C reflect the three dichotomous questions that must be answered. Any answer causes transition to the next state. A database containing responses would need to allow three fields for answers.

Question 5 is a true multiple choice question. The flow of questions could be changed based on the which response is selected. The question and choice of responses should be presented on the same screen during an active presentation. In the state diagram, question 5 is represented by state 5. Any response will cause transition to state 999. The expected answer is one of the characters in §a, b, c, d†.

The Thank You message at the end of the screen is also an important part of the questionnaire and should be presented during an active presentation. The "?" transition out of state 999 indicates any

response at state 999 will cause the transition to state 0, the termination state. Any acknowledgment of this statement causes transition to the end state.

An active presentation of this questionnaire has been generated to illustrate the various presentation and data gathering techniques developed for qmail.

3.4 Design Decisions

3.4.1 Environment and Security Issues

UNIX is a natural choice for this project due to its openness and the availability to the author of UNIX hosts. Since UNIX System V hosts are available in the author's work environment, and the mailx command is the System V implementation of the BSD mail command, mailx under UNIX System V became the natural choice of environment. For similar reasons, the author chose to implement the project on an AT&T 3b2 computer.

Security was an overriding concern in this project. Other active mail systems described previously in this report (imail and ETC for example) actually send code. Wanting to design a more secure system that can be controlled by the administrators of all the hosts that might use it, this author chose not to send arbitrary programs through the mail for activation. Instead the author chose to specify the name of the program to activate as the first field in the message body, followed by its input.

This technique has the advantages of reducing the message size (only the program name is sent, not the program), providing confidence that the activated program does the same thing it did last time this type of message ran, and users are provided with a consistent set of tools that do not vary according to who sent the message. However, this has the disadvantage of requiring any tools developed be distributed to the host of any potential recipient.

Also in line with keeping the system as secure as possible, it was decided that all programs which are to be activated by messages had to be stored in a library specified by IT. This prevents inadvertently executing some other program in the PATH.

Additionally, it was decided to execute the programs with the same authority as the mail program, thereby avoiding the easily misused "setuid" subroutine.

Finally, it was decided the interface should be nearly transparent to the user. Selecting an active message should be no different that selecting a passive message. It is believed that aside from a possible courtesy in the subject line of the header reflecting this message is active, no other notice should be necessary for the recipient. Possible security hazards exist as the recipient activates messages without awareness, but if other security controls are met, this should not cause a problem.

3.4.2 Scope

Having decided on an environment and technique, the question arose as to what kind of tool to develop. Keep in mind that the author wanted to explore active messages, not active user agents (although the distinction is not always clear).

The questionnaire was chosen as most people have an intuitive notion of what a questionnaire is and it has some potential for being used in business applications. Also, a questionnaire is sufficiently simple to implement so that the focus could remain on active messages, rather than the functionality of the tool which was implemented. It was further decided to restrict the questionnaires to multiple choice questions and questions that could be answered with a simple number (e.g. What is your age?). This allows a great deal of functionality, keeps the implementation of the tool simple, and keeps the focus on active messages, instead of the tool.

Having decided that a questionnaire was an appropriate vehicle to demonstrate the potential of active messages, the author considered the extent of the functionality of the questionnaire presentation system. A decision was made that special formatting of text (bold, underline, italics, color, centering, etc.) was desirable, but beyond the scope of this project, as the mailx command does not support transmission of special characters. Another part of the reason for this decision is the lack of standards implemented in a mail system for transmitting specially formatted text. (The Office Document Architecture (ODA) standard describes formatting specifications for electronic documents. However, the ODA abstract model is designed for memos, newsletters, and similar items [Carr

88]. It is not clear to this author the standard is appropriate for presentation of active messages. With its standards for mixing graphical and textual data, it might be appropriate for presenting limited multi-media mail.)

In order to present the questions and choice of responses clearly to the recipient, a decision was made to provide limited presentation ability. It was decided that each question should carry information specifying what action is to take place before presentation of this text. Actions were limited to clearing the screen, double spacing or single spacing. It is assumed that the choice of responses should be on the same screen as the triggering question.

The next question was what to do with the responses generated by the recipient. It was decided a database needed to be included to make the processing of the response messages more manageable. Availability and familiarity led to the use of Informix™. Additionally, Informix runs on PC's and a variety of mainframes, making the solution more general and more readily extended.

The author considered whether to send the message containing the responses back to the originator in a format capable of being loaded into the database, or some other format. The format for loading

into Informix is very general, and in it fact the same format can be used to load the same information into other databases. Each response (database field) is separated from its neighbor by the "|" symbol in load format; this increases the length of the message but increases the message clarity. This separation also allows for variable length response fields. The author decided upon formatting before return for reasons of clarity, generality, expandability, and minimal penalty on length. (In a message containing 100 responses, at most 100 extra characters would be sent.)

Loading the responses into the database would be more conveniently accomplished by an active UA (without intervention by the originator of the questionnaire). However, with the focus on active messages rather than active UAs, the author decided to accomplish the actual loading by requiring the user to activate the message.

3.5 Overview of the qmail tool

3.5.1 Mail System Modifications

The IT extension to mailx is designed to identify active messages and present them properly to the recipient. The IT extension to mailx along with mailx is called itmail. The extension portion of itmail provides facility to identify active messages and activate them, the mailx portion provides delivery service for messages. itmail and mailx are completely compatible, either can handle messages meant for the other. However, mailx will not activate a message, but rather present the body to the recipient as text.

The functions of IT are:

1) Determine if a message selected for presentation is an active message.

2) Obtain the name of the activation program from the message body.

3) Validate that the activation program is an executable program in an authorized library.

4) Pass control to the activation program with the message as input for presentation to the recipient.

3.5.2 qmail as a Questionnaire

The qmail tool is designed to work in the environment provided by IT. qmail performs three major functions:

1) Help the interviewer create a questionnaire including the questions, responses and flow of information. Additionally, an Informix database is created to provide a collection point for anticipated responses.

2) Provide a convenient and consistent user interface to present the questionnaire to the recipient.

3) Automatically load the responses from all of the recipients into the interviewer's database (provided in 1).

In this environment, a person being interviewed via qmail can deal with a questionnaire at his convenience.

For the rest of this discussion, the interviewer who originates the questionnaire will be referred to as the originator and the respondent who receives the questionnaire will be called the recipient.

The information flow between qmail, the originator, and the recipient through itmail is illustrated by figure 3.2. This flow is described below.

1) The originator creates a questionnaire message using a program called qcreate.

2) A database to hold responses for this questionnaire is prepared by qcreate.

3) The message is mailed to one or more recipients using itmail.

4) The message arrives at the destination, and is selected for presentation by the recipient.

5) The message is identified as active by itmail, and control is passed to qrun.

6) The message is presented to the user by qrun.

7) The user's responses to the questions on the questionnaire are collected by qrun, and prepared as a response message.

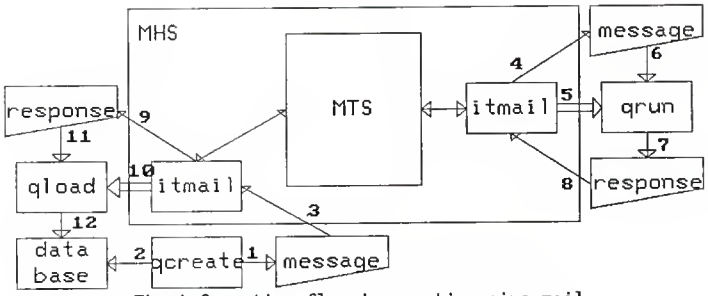
8) The response is mailed to the originator using itmail.

9) The response arrives at the originator's location, and is selected for presentation.

10) The response is identified as active by itmail, and control is passed to qload.

11) The data collected from the recipient of the questionnaire is obtained by qload.

12) Said data is loaded into the database.



The information flow in questionnaire mail
Figure 3.2

3.5.3 Creating a Questionnaire

The command "qcreate" is used by the originator to create, update, or re-issue a questionnaire.

Preparation of the questions, responses, and flow of questions within the questionnaire may be performed in several sessions with the intermediate results saved each time. When a questionnaire is complete, it may be issued. Issuing a questionnaire initiates and completes various housekeeping functions including preparation of a

response database. A different response database is prepared for each issue of the questionnaire, and is identified by an instance-id (iid).

When creating or updating a questionnaire, the functions of qcreate are:

1) Determine the name of the questionnaire.

2) Determine the questions to be present on the questionnaire.

3) Determine the responses allowed for each question.

4) Determine the flow of questions through the questionnaire.

5) Ensure there is no endless loop of questions within the questionnaire.

6) Ensure the flow of questions always specifies an existing question.

7) Allow a (partial or complete) questionnaire to be stored for further updating or issuing at a later time.

After preparation of the questionnaire with qcreate, it may be dispatched. It is critical that the recipient use itmail to receive the message, and the originator use itmail to receive the recipient's response.

When a questionnaire is issued, qcreate performs the following:

1) Determine the name of the questionnaire to be issued.

2) Determine the iid.

3) Determine the cut-off date, by which the questionnaire must be activated.

4) Determine the questionnaire is complete and consistent as in items 3 and 4 above.

5) Store the data just collected with the questionnaire in a format acceptable for mailing.

6) Prepare an Informix database to hold the anticipated responses.

3.5.4 Interacting with the Recipient

The command "qrun" is automatically invoked by the IT extension to mailx when a recipient selects an active questionnaire for presentation. The function of qrun is to present only the appropriate statements and questions of the questionnaire to the recipient based on his responses and the flow of questions determined during creation of the questionnaire. The responses are stored and then returned to the originator.

The body of the message is received by qrun from IT and performs the following tasks:

1) Ensure that the cutoff date has not passed. Notify both the recipient and originator if the last acceptable date for responding has passed, and terminate.

2) Present the flow of questions and related responses to the recipient based on the recipient's responses to the questions.

3) Transform the user's responses to the questions into the proper format for loading directly into the originator's Informix database.

4) Dispatch an active message containing the formatted responses to the originator of the questionnaire.

3.5.5 Receiving the Responses

The command "qload" is automatically invoked by the IT extension when a recipient selects for presentation a response to a questionnaire. The purpose of qload is to load the responses into the appropriate response database.

The functions of qload are:

1) Use fields in the body of the message to determine what database is to receive this record.

2) Extract the data record from the body of the message.

3) Run the proper program to load the response database using the data record as input.

3.6 Implementation

3.6.1 IT

The mailx source code received by the author consisted of 8 header files containing 647 lines of code, 32 source files containing 12,141 lines of code, and a make file to coordinate compilation into a single executable file containing 111,984 bytes.

Discovering that the mailx source available was for a different release of UNIX System V, (and would not compile under release 3.1 because of differences in the standard header files) was the first problem encountered. This was resolved by removing a definition in a header file and five signal handling routines in the source file sigretro.c. This results in inaccuracies in comparisons of the size of the current mailx command and the extended itmail version.

itmail was created by deleting 147 lines of code from sigretro.c, modifying a line of code in another module, and adding 309 lines of code to five other modules. Compiled, itmail contains 114,520 bytes.

The unique starting point in mailx for presenting messages to the recipient (upon selection via entering the message number, or pressing _return¢ or "n") is with a module called "next". This passes control through three intermediate modules ("type", "type1", and "print") and finally to the module "send", which presents the message selected by the user. It was determined that "send" (and the intermediate programs) was also used for several other purposes (e.g. copying messages back into the users mail file upon termination of the program).

The solution was to provide a unique path from "next", through special versions of the intermediate modules, to a special copy of "send" called "sendit". This guarantees no interference with other functionality of mailx, and complete modularity of the IT function. The other solution (carrying flags through all intermediate modules to determine exactly what the user wanted to do) would have resulted in fewer lines of code at the expense of less readable and less modular code.

A special version of "send" called "sendit" was created. "sendit" is invoked rather than "send" when a message is selected for presentation using the _return¢ key, the number of the message, or "n" for next message. "sendit" saves the name of the originator of

the message (from the first line of the header) and looks for the characters "Activated by: " in the first line of the body of the message. If these characters are not found, this is flagged as an active message, otherwise processing continues as normal.

If the message is active, the name of the activation program is captured, an activation input file (/tmp/Ra$§pid†, previously identified in routine "tinit") is opened for output, and the rest of the message is copied into that file instead of to the recipient's standard output.

Once the rest of the message has been copied into the activation input file, it is closed. Checks are made to insure the activation program exists and is executable in directory /usr/lbin/mail. If not, a warning is posted to the recipient, and active processing stops. If these tests are passed, the mailx routine "shell" is used to transfer control to the activated program. The id of the message originator and the activation input file are passed as parameters.

The command "shell" is normally used by mailx to run shell commands from the mailx prompt. The advantage to using "shell" to activate messages (rather than the UNIX "system", "fork", or other command) is that it properly handles signals within mailx, and will return

control properly if the activation program is interrupted. No modifications to shell were necessary for activation of an active message.

3.6.2 qcreate

A user invokes qcreate to prepare questionnaires in the proper format for use by qrun and to prepare a database to contain the anticipated responses. qcreate may be either a create/update run, or an issuing run.

Processing begins by determining the name of the questionnaire from either the command line or by prompting the user. A file of this name is sought. If the file is found it is assumed this run will update this file or issue it. In either case, the file is loaded into memory for reference. The first part of the file contains questions, the second part contains possible responses. The questions and responses are loaded into separate arrays.

As the file is loaded, two lists are built, one is a list of "every question" existing in the file. The other lists every "next question" in the question flow (extracted from the possible response array).

If there is no file by this name, it is presumed that a file will be built and the arrays and lists are cleared. Of course the file must exist if this is an issuing run.

If this is an issuing run, the questionnaire is checked for validity. The questionnaire is complete if every question number listed on the "next question" list is also on the "every question" list (with the exception of question 0, which is a termination flag). The questionnaire is terminable if flow always passes to a question with a higher number than the present number.

The next step in issuing the questionnaire is to obtain the instance id and the cutoff date. This data along with the number of questions, the question array, and the possible response array is formatted into textual data, and stored for transmission.

An Informix database is then built. It contains a field for the recipient's mail id, a field for date received, and one field for each question in the questionnaire. If a question anticipates a numeric response, an integer field is provided, otherwise a one character field is provided for each question.

The final step of issuing is to notify the user the questionnaire has been stored and is ready to be mailed.

A "create or update" execution of qcreate accesses the same data structures and some of the same routines as the issuing execution. After the internal arrays have been loaded (or cleared) and the lists built, the program begins by asking for the question number to develop next.

When the user enters the question number, the array of questions is examined for this item. If it exists, the existing information is presented as default data. Replacement text is obtained from the user, along with the presentation id and the acceptable answers.

The presentation id may be:

" " if the screen is to be cleared before presentation.

"s" if this question is to be one line below the previous.

"d" if this question is to be double spaced.

The acceptable answer field may contain:

"c" if this question has no user response and the presentation program is to continue immediately with the next question.

"r" if this question is to be acknowledged by the user entering _return¢ before proceeding to the next question.

"n" if this question is to be responded to by the entry of a number.

"y" if this question is to be responded to by the entry of "y" for yes or "n" for no.

"t" if this question is to be responded to by the entry of "t" for true or "f" for false.

"a" if this is a multiple choice question. Responses must be a letter of the alphabet which correspond to one of the possible responses.

After the question has been acceptably entered, the user is prompted for possible responses. The possible responses consist of textual data to present (in the case of multiple choice questions) and the next question to present. Default values are taken from the array of responses.

The "next question to present" defaults to the next higher question (unless there is a question number in the array of responses). The next question must be greater than the existing question number (to prevent infinite loops of questions). Responses are accepted per the acceptable answer field in the question array as follows:

"c", "r", or "n" The user is prompted for the next question to present. No text is accepted to display.

"t" or "y" The user is asked to which answer this response relates. If the user enters "?" followed by a "next question" number, this indicates the flow of questions does not split, and any response will be directed to the

same question. Or the user may split the flow by entering a valid response ("t" or "f", "y" or "n") followed by the next question. In the second case the user must make entries for both answers.

"a" The user is prompted for the text to present, when this is accepted, he is prompted for the next question to present. "qcreate" keeps track of the response identifier (a, b, etc.) The next question to present defaults to the next question in sequence if there is no previous entry in the array of responses.

As each question is entered and each possible response is entered, it is added to the "every question" list or the "next question" list as appropriate.

After the entry of each question, the user is notified of those items on the "next question" list that are not on the "every question" list. The user may then select the next question to enter on the questionnaire.

If the user elects to stop at this point, the arrays are stored in a file named after the questionnaire. As mentioned earlier, qcreate may be used later to further develop or issue the questionnaire.

3.6.3 qrun

The program to activate a questionnaire is qrun. When qrun receives control from the IT extension, it expects the first parameter to be the name of the originator of the message and the second parameter to be the name of the input file.

The input is opened, and the first two lines, containing the questionnaire name and instance id, are read and saved for output later.

The third line, containing the cutoff date, is read. If the current date is past the cutoff date, an error message is presented to the recipient, a mail message identifying this situation is sent to the originator, the input is closed, and processing stops.

Next the questions and responses are read from the input and loaded into an array and the input is closed.

The first question is then processed. Processing begins with either clearing the screen or advancing the proper number of lines, based on the "presentation id" of the question. Then the text of the question is displayed on the screen.

The next step is processing the "acceptable answer" field. If this field contains "n", the recipient is prompted for an integer number, if it contains "r" the program waits for the receipt of a <return> character, if the field contains "c" the program continues immediately to process the next question, if the field contains "t" or "y" the recipient is prompted for a true-false or yes-no answer, if the field contains "a", the text in the responses related to this question are presented, and the user is prompted to select one of the choices.

A valid response is then obtained from the user and stored in an array. It is preceded with an appropriate number of separation characters ("|") to place it in the proper position for later loading into a database.

Next the response array is examined for entries related to this question, as determined by question number. For each related response, the "flow control" field is examined. If this field is a "?" or exactly matches the user's response, then flow passes to the question specified in the "next question" field of this response. Otherwise the next response is examined. Since the questions and possible responses are computer generated, and the responses from the recipient are validated, there will always be a match. For protection against messages garbled in transmission, if no match is found then flow passes to the question specified in the next question field of the last possible response for a question.

Each question is presented and responses accepted from the recipient until the next question specified is question zero, which indicates end of processing. At this point, the input is closed and the output array is filled with the appropriate number of separation symbols to fill out the data base load record. Finally the stored questionnaire name, instance id, and the data base record are formatted and mailed to the originator as an active message activated by qload.

3.6.4 qload

The program qload is activated by the IT extension upon receipt of an active message prepared by qrun.

The name and instance id of the database to load are extracted from the body of the message. The system date is obtained, and the respondent's name is obtained (from the first passed parameter). This information is concatenated with the rest of the database record contained in the input file.

Finally a background process is spawned (via a "system" call) to load this record into the appropriate Informix database.

The program terminates by notifying the user the process has been spawned.

4 Where Does IT Stop?

The purpose of this chapter is to identify the possibility of using active messages (and active UAs) for "dataveillance". The concept of dataveillance is defined as the systematic use of personal data systems in the investigation or monitoring of the actions or communications of one or more persons [Clarke 88].

An appropriate use of dataveillance would be to investigate a crime. An inappropriate use would be to gather information to embarrass or discredit an individual. Several uses of active messages and/or active UAs for performing dataveillance are discussed below.

An appropriate use of an active message would be to seek out bulletin boards with conversations on topics of interest to the originator. Another use of an active message (or an active user agent) would be to identify all members participating in a particular conversation on a bulletin board. By manually monitoring a bulletin board, this information is available today to anyone willing to expend the effort. Using an active agent to perform this dataveillance improves the efficiency of the data gathering, and

allows the gathering of information on large groups of people. It would be extremely difficult to detect that this activity is taking place.

One use of an active user agent is to sift out messages which are of no interest to the recipient. It would even be appropriate for an active mail message to report back to the originator that a message had been refused based on the subject matter. It might be inappropriate to send out an organized series of active messages to an individual, each message pertaining to a different subject, but the subjects presented spanning a spectrum of interests. Based on what messages are refused and accepted, it would be possible to determine what subjects interest a given individual.

The same technique could be used to gather information on large groups of people to identify patterns of interest. From knowledge of the group and knowledge of the individual, other interests of the individual can be inferred. For example, if it is determined (from dataveillance) that 90% of all people who are interested in the topics "natural food" and "home gardening", are also interested in "marijuana", and it is known that a particular individual is interested in "natural food" and "home gardening", then it might be inferred that this individual is interested in "marijuana". One

concern here is the potential for incorrectly inferring information. The second concern is the use of dataveillance to gather information on an individual (or groups of individuals) without their consent or knowledge. Once again, it is very difficult to identify that this type of dataveillance is transpiring.

Another appropriate use of an active message would be to gather information concerning an individual from a database organized on an individual basis (i.e. a credit report, bank balance, or insurance data). This would make activities such as preparing a loan application more efficient than if the data was gathered by manually accessing the database or receiving the message through passive mail (electronic or otherwise). Inappropriate use would be illicitly gathering information on an individual from these same databases. Provisions must be made to insure that active messages can only be executed on behalf of originators who are authorized to gather this information.

Recalling the experience described by [Stoll 88], it is not unreasonable to assume it would be possible to send an unauthorized active message masquerading as an authorized message. In the extreme case, the message could be programmed to send results back to the originator, and dispatch itself to another database to gather

more information. If access is not gained in one attempt, the message could automatically try again later. The assumption is that some security measure will be poorly enforced at some point in time, and penetration will eventually be achieved. The message could modify itself to change the apparent originator, and even spawn children that would be of the proper format to communicate with a different class of UA.

The power of active messages lies in their ability to remotely perform tasks efficiently for the originator. The potential for misuse of an active message lies in its ability to perform these tasks without the knowledge of the recipient. It must be recognized that active messages may perform dataveillance. The information gathered may be used appropriately or inappropriately. Determining how to prevent inappropriate dataveillance is not an easy task.

The critical questions are "What are appropriate controls for computer applications to prevent unauthorized use?" and "How are these controls implemented and maintained?"

Clearly the potential for dataveillance by active messages will increase in the coming years as information technology continues to develop, as more data is stored in places accessible by computer,

and as more computers are networked together. The danger lies not with the visible tasks performed by an active message or UA (such as a questionnaire), but with the invisible tasks performed by an active message or UA (such as the remote inquiry on a database) without the recipient's knowledge.

5 Conclusions

5.1 Perspective

Active messages have been shown to be a useful extension to computer based message systems. When put to use with an appropriately designed UA, they provide an effective means to speed the flow of information and reduce the time spent on important yet mundane tasks.

However, active messages are an extremely powerful tool which must be developed with foresight of their potential for misuse. The danger lies in the fact the functions carried out by the active message are not necessarily visible to the user. This is especially true when the active message is executed by an active user agent

without knowledge of the user. Attention must be paid to the validation of active messages to ensure that any function carried out is appropriately authorized.

5.2 Extensions

Possible extensions to IT would include porting it to a mail system with a PC-based UA (such as AT&T Mail Access Plus) or to another UNIX mail command.

Possible extensions to qmail include expanding it to use a data base system other than Informix. The qcreate program is simplistic at best, there is ample opportunity to enhance the user interface of that program. Extensive work has been done in presenting information to users and gathering responses for use as computer aided instruction. These same programs might be adapted to present very effective questionnaires. There is nothing in this implementation which would preclude an active UA from activating the qload program without user intervention and disposing of the message after a successful load. Similarly, all responses to a questionnaire could be placed in one file for a mass load later.

Developing other tools to be activated besides the questionnaire would also be of interest. Tools to process office forms as discussed by [Malone 87], time management tools for individuals, time management tools for work groups, and a facility to update an individuals personal telephone directory remotely are all appropriate extensions of IT.

References

[ATTMAIL 87] "AT&T Mail Access PLUS User's Guide", AT&T, 1987.

[Carr 88] Carr, Richard, "ODA - The ISO Standard for Electronic Document Interchange", Computer Standards & Interfaces 7 (I988) pg 297-30I.

[Caswell 86] Caswell, S.A., "Personal computer software & electronic mail", in Electronic Message Systems, Proceedings of the International Business Strategy Conference, 1986, pg 324-333.

[Clarke 88] Clarke, Roger A., "Information Technology and dataveillance", Communications of the ACM, Vol. 3I, Number 5, May I988

[Gitman 85] Gitman, Israel, "Voice Mail and Competing Services", in Computer Message Systems - 85, R.P. Uhlig (editor), Elsevier Science Publishers B.V. (North-Holland), IFIP I986, pg 405-411.

[Gold 86] Gold, Eric, "Envoys in Electronic Mail Systems", ACM TOIS, 1986.

[Henderson 87] Henderson, Harry, "The USENET System", in UNIX Papers, the Waite Group (editor), Howard W. Sams & Company, 1987 pg 43-90.

[Hogg 84] Hogg, John, Gamvroulas, Stelios, "An Active Mail System", SIGMOD Proceedings of Annual Meeting, June 1984.

[Huitema 85] Huitema, Christian, "The 'COSAC' electronic conferencing experiment", in Computer Message Systems - 85, R.P. Uhlig (editor), Elsevier Science Publishers B.V. (North-Holland), IFIP I986, pg 277-284.

[Jaburek 84] Jaburek, W., Sebestyen,I., "Computerized Message Sending and Teleconferencing on Videotex Through Intelligent Decoders, Smart Cards, and Optical Cards", in Computer-Based Message Services, H.T. Smith (editor), Elsevier Science Publisher B.V. (North-Holland), IFIP I984, pg 329-340.

References

[Kille 84] Kille, Steve, "Integration of Electronic Mail and Conferencing Systems", in Computer-Based Message Services, H.T. Smith (editor), Elsevier Science Publisher 8.V. (North-Holland), IFIP 1984, pg 261-269.

[Kinnear 83] Kinnear, Thomas C., Taylor, James R., "Marketing Research", McGraw-Hill, 1983

[Malone B7] Malone, Thomas W., Grant, Kenneth R., Lai, Kum-Yew, Rao, Ramana, Rosenblitt, David, "Semistructured Messages Are Surprisingly Useful for ComputerSupported Coordination", ACM TOIS, Vol. 5, No. 2, April 1987, pg 115-131.

[NBS 79] Panko, R.R., "Standards for Computer Based Message Service",National 8ureau of Standards, 1979.

[Newman 84] Newman, Julian, Harvey, Sharon, "Contracts made by Electronic Mail: Legal Issues, Technology and Services", in Computer-8ased Message Services, H.T. Smith (editor), Elsevier Science Publisher B.V. (North-Holland), IFIP 1984, pg 237-246.

[Palme 85] Palme, Jacob, "Distribution Agents (mailing lists) in Message Handling Systems", in Computer Message Systems - 85, R.P. Uhlig (editor), Elsevier Science Publishers B.V. (North-Holland), IFIP 19B6, pg 117-131.

[PCWORLD BB] Smith, Ken, Rodarmor, William, "Making the E-Mail Choice", PC World, March 1988, pg 130-135.

[Postel 88] Postel, Jonathan B., Finn, Gregory G., Katz, Alan R., Reynolds, Joyce K., "An Experimental Multimedia Mail System", ACM TOIS, Vol. 6, No. 1, January 1988, pg 63-81.

[Rose 85] Rose, Michael T., Farber, David J., Walker, Stephen T.,"Design of the TTI Prototype Trusted Mail Agent", in Computer Message Systems - 85, R.P. Uhlig (editor), Elsevier Science Publishers 8.V. (North-Holland), IFIP 1986, pg 377-399.

References

[Smith 84] Smith, H.T., preface to "computer-based message services", Elsevier Science Publishers B.V. (North-Holland), 1984.

[Stoll 88] Stoll, Clifford, "Stalking the Wily Hacker", Communications of the ACM, Vol. 31, No. 5, May 1988

[Taylor 87] Taylor, Dave,"All About UNIX Mailers", in UNIX Papers, the Waite Group (editor), Howard W. Sams & Company, 1987, pg 93-121.

[Uhlig 85] Uhlig, R.P.,preface to "Computer Message Systems - 85", Elsevier Science Publishers B.V. (North-Holland) IFIP, 1986.

[Vittal 81] Vittal, John, "Active Message Processing: Messages as Messengers", in Computer Message Systems, R.P. Uhlig (editor), Elsevier Science Publishers B.V. (North-Holland), IFIP 1981, pg 175-195.

[Waite 87] "UNIX Papers", the Waite Group (editor), Howard W. Sams & Company, 1987.

[Wilson 84] Wilson, P.A., Maude, T.I., Marshall, C.J., Heaton, N.O., "The Active Mailbox - your on-line Secretary", in Computer-Based Message Services, H.T. Smith (editor), Elsevier Science Publisher B.V. (North-Holland), IFIP 1984, pg 137-148.

[X.400 84] CCITT X.400,X.401, X.408, X.409, X.410, X.411, X.420, X.430: "Message Handling Systems", CCITT Red Book, 1984.

During questionnaire creation, normal termination of qcreate causes an intermediate file called <qname>.int to be saved, where <qname> is the name of the questionnaire. This file will be retrieved automatically to continue development of the questionnaire at a later date.

When the questionnaire is issued, the file <qname><iid>.q is created and may be mailed to the recipients.

Also at issue time, a file called <qname><iid> is created containing the definition of the informix database which will is then built.

A database called <qname><iid>.dbd is built using the definition file. The database name is limited to 10 characters in length.

No special files are created or used by qrun.

The qload program generates two temporary files in the directory /tmp using the process id (<pid>) of qload. The first is called RC<pid> and contains commands to load the data into the existing database, the second is named RL<pid> and contains the actual data to load.

```
/* qcreate prepares and issues questionnaires
   preparation may take place in multiple executions
      intermediate results are stored as <qname>.int
   issuing causes the questionnaire to be stored as <qname).fin
      an informix database is built to hold responses

qcreate [ [-i] <qname>]
*/

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>

/* items that should be in a qmail.h */
/* belongs in qmail.h file */
/* ANSI standard screen control commands */
/* Clear Screen */
#define CLEAR "2J"
/* Restore cursor position, down 1 line */
#define SNGL  "\n"
/* Restore cursor position, down 2 lines */
#define DBLE  "\n\n"
/* the Bell */
#define BELL "\007"
#define SPLIT 1
#define NOSPLIT 0
#define MAXLINE 80
#define MAXQ 100
#define MAXR 2*MAXQ
#define MAXTXT 72

struct qsts  {
    int  qno;
    char prs;
    char acc;
    char qtext[MAXTXT];
};
struct resp {
    int  qno;
    int  nqno;
    char rsp;
    char rtext[MAXTXT];
};
struct qrlst {
     int itno;
```

```
     int lstptr;
};
/* end of items to be in qmail.h file */

/* extern items to be called by several programs */
int pq=0, pr=0;
     /* count of questions and possible responses */
int cq=0, cr=0;
     /* current question and possible response    */
struct qsts q[MAXQ]; /* array of questions */
struct resp r[MAXR]; /* array of possible responses  */
struct qrlst evq[MAXQ];
     /* list of every question, with pointer to the question */
struct qrlst evr[MAXR];
     /* list of every response */
int evn[MAXQ];  /* list of every next question */
int evqm=0, evqc=0;
     /* max items on every question list, current item */
int evnm=0, evnc=0;
     /* max items on every next question list, current item */
int evrm=0, evrc=0;
     /* max items on every response list, current item */

char line[MAXLINE+1];   /* working array for input */
char qname[9];  /* name of this questionnaire */

char iid[5];  /* instance id */
char ifname[13]; /* intermediate file name, <qname>.hold */
char issfile[11];
     /* issued questionnaire file name: <qname><iid> */
char dbasenm[15]; /* name of database: <qname><iid>.dbd */
char dbasedf[14];
     /* name of database definition file: <qname><iid>.ib */

main(argc, argv)
int argc;
char *argv[];
{
int issue=0; /* 1 if qcreate invoked with -i */
int rpt=0; /* 1 if qname specified and exists */
char c;
struct stat statc;

/* paramater handling routine */
/* insure max of two parameters are specified */
if (argc > 3) {
```

```
   printf("%s requires no more than two parameters \n",argv[0]);
   return(-1);
}
strcpy(qname,"\0"); /* insure questionnaire name is cleared out */
/* if two parms, first must be -i, second qname */
if (argc == 3) {
   if (strncmp(argv[1],"-i",2) != 0) return(parmuse());
   else {
      issue++;
      rpt++;
      strcpy(qname,argv[2]);
   }  /* end of situation for 2 parameters */
}
else {
  if (argc == 2) (
     if(strcmp(argv[1],"-i") == 0) return(parmuse());
     else {
     rpt++;
     strcpy(qname,argv[1]);
} )  }

/* now prompt for qname if not provided on command line */
while(strlen(qname) == 0) {
     printf("Enter name of questionnaire to process: ");
     gets(line);
     if(strlen(line) > sizeof(qname)-1)
     printf("Questionnaire name must be %d characters or less\n",
      sizeof(qname)-1);
     else {
      strcpy(qname,line);
      printf("Is %s the correct questionnaire name? (y/n): ",
       qname);
      gets(line);
      if(tolower(line[0]) != 'y')
       qname[0]='\0';
                else {

}       }       }

/* test for existence of the questionnaire file */
strcpy(ifname,qname);    /* generate name of intermediate file */
strcat(ifname,".int");
/* test for existence */

if (stat(ifname,&statc) < 0 )
     rpt--;
```

```
else
     rpt++;

/* initialize every question list,
 * every response list,
 * every next question list
 *      load lists from intermediate file if it exists
*/
cq=cr=pq=pr=0;
if(rpt > 0) if(intload() == -1) return(-1);

/* do not issue non existent file */
if (issue && rpt <= 0) {
     printf("Unable to issue %s as %s does not exist\n",
      qname,ifname);
     return(-1);
}

if (issue) {
     if(qvalid() == 0)
      return(qissue());
     else {
      printf("Invalid Questionnaire\n");
      return(-1);
}    }

/* generate question and choice of response arrays
     from user input */
genq();

/* store intermediate results */
store();

return(0);
}

/* store saves the question and answer array as <qname>.hold
 * questions are preceeded by a q, responses by an r
 * the questions must be in order by question number
 * the answers by question number and response (? last)
 */
store()
{
int n, m, swt;
int hl, pl, p2;
FILE *file;
```

```
/* sort questions into order */
for(n=evqm-1,swt=0;n>0;n--,swt=0){
     for(m=0;m<n;m++) {
      if(evq[m].itno > evq[m+1].itno) {
       hl=evq[m+1].itno;
       evq[m+1].itno=evq[m].itno;
       evq[m].itno=hl;
       hl=evq[m+1].lstptr;
       evq[m+1].lstptr=evq[m].lstptr;
       evq[m].lstptr=hl;
       swt++;
     } }
     if(swt == 0) break;
}    /* end of sort of questions array */

/* sort responses into order */
for(n=evrm-1,swt=0;n>0;n--,swt=0){
     for(m=0;m<n;m++) {
      if(evr[m].itno > evr[m+1].itno) {
       hl=evr[m+1].itno;
       evr[m+1].itno=evr[m].itno;
       evr[m].itno=hl;
       hl=evr[m+1].lstptr;
       evr[m+1].lstptr=evr[m].lstptr;
       evr[m].lstptr=hl;
       swt++;
      }
      p1=evr[m].lstptr; p2=evr[m+1].lstptr;
      if(evr[m].itno == evr[m+1].itno)
      if(r[p1].rsp > r[p2].rsp || r[p1].rsp == '?') {
       hl=evr[m+1].itno;
       evr[m+1].itno=evr[m].itno;
       evr[m].itno=hl;
       hl=evr[m+1].lstptr;
       evr[m+1].lstptr=evr[m].lstptr;
       evr[m].lstptr=hl;
       swt++;
        }       }
     if(swt == 0) break;
}    /* end of sort of responses array */

/* store the arrays */
file=fopen(ifname,"w");
for(n=0;n<evqm;n++) {
     m=evq[n].lstptr;
```

```
     fprintf(file,"q%3d%c%c%s\n",
      q[m].qno,q[m].prs,q[m].acc,q[m].qtext);
}
for(n=0;n<evrm;n++) {
     m=evr[n].lstptr;
     fprintf(file,"r%3d%3d%c%s\n",
      r[m].qno,r[m].nqno,r[m].rsp,r[m].rtext);
}

fclose(file);
}    /* end of store arrays routine */


/* genq displays questions referenced but not developed yet
 * obtains a question number from the user
 * displays current text for that question if it exists
 * accepts updated text from the user
 * accepts action to perform on screen before question presented
 * accepts type of response
 * accepts response text as necessary
 * accepts next question number for each type of response
 */
genq()
{
int qno, cont=1, ok;
char c;

printf(CLEAR);
while(cont != 0) {
     if(listopen()==0) printf("No undeveloped questions\n");
     printf("What question would you like to develop next? ");
     gets(line);
     sscanf(line,"%d",&qno);

     /* get pointer to this question in array of questions
      *                      response            possible responses
      */
     cq=dflt(qno,evqm,evq);
     cr=dflt(qno,evrm,evr);

     if(cq == -1) {
      cq=pq++; evq[evqm].itno=qno; evq[evqm++].lstptr=cq;
      strcpy(q[cq].qtext,"New question");
      q[cq].qno=qno;
     }
     if(cr == -1) {
```

```
      cr=pr++; evr[evrm].itno=qno; evr[evrm++].lstptr=cr;
      r[cr].qno=qno;
     }
     /* obtain text of question */
     ok = 0;
     while(!ok) {
printf("Following text may be accepted by entering <return>\n%s\n",
       q[cq].qtext);
      printf("Otherwise enter the text of question %d:\n",qno);
      gets(line);
               if(line[0] == '\0') break;
      strcpy(q[cq].qtext,line);
              printf("Is the following text ok? (y/n)\n");
      printf("%s\n",q[cq].qtext);
      gets(line);
      if(tolower(line[0]) == 'y') break;
     } /* end of while to get text of question */

     /* obtain presentation control of question */
     ok = 0;
     while(!ok) {
      if(q[cq].prs == '\0'|| q[cq].prs == ' ') q[cq].prs = 'c';
      printf("From the following list enter ");
      printf("the presentation control for question %d:\n",
       qno);
      printf("c - Clear Screen, d - Double Space, ");
      printf("s - Single Space before presentation\n");
      printf("Following presentation control may be ");
      printf("accepted by entering <return> %c ",
       q[cq].prs);
      gets(line);
          if(line[0] == '\0') break;
      c=tolower(line[0]);
          if(c == 'c' || c == 'd' || c == 's') {
       q[cq].prs=c;
       printf("Is \"%c\" the correct presentation ",c);
       printf("control for question %d? (y/n) ",qno);
       gets(line);
       if(tolower(line[0]) == 'y') break;
      } else {
       printf("Invalid presentation control character\n");
      }
     } /* end of while to get presentation control */
     /* convert clear to blank */
     if(q[cq].prs == 'c') q[cq].prs = ' ';
```

```
     /* obtain acceptable responses for question */
     ok = 0;
     while(!ok) {
      if(q[cq].acc == '\0') q[cq].acc = 'c';
      printf("What response is needed for question %d?:\n",
       qno);
      printf("y - yes/no, t - true/false, n - numeric, ");
      printf("a - multiple choice,\n");
      printf("r - continue with any key, c - continue ");
      printf("without any input\n");
      printf("Following response code may be accepted by ");
      printf("entering <return> %c ",q[cq].acc);
      gets(line);
      if(line[0] == '\0') {c=q[cq].acc; goto getrsp;}
      c=tolower(line[0]);
                if(     c == 'y' ||
       c == 't' ||
       c == 'n' ||
       c == 'a' ||
       c == 'r' ||
       c == 'c') {
      printf("Is \"%c\" the correct response code? (y/n) ",c);
       gets(line);
       if(tolower(line[0]) == 'y') break;
      } else {
       printf("Invalid response code!n");
      }
     } /* end of while to get presentation control */

getrsp:
     switch (c) {
     case 'y': q[cq].acc='y';
      if(cksplit()) splitit();
      else nqnopr(NOSPLIT);
      break;
     case 't': q[cq].acc='t';
      if(cksplit()) splitit();
      else nqnopr(NOSPLIT);
      break;
     case 'a': q[cq].acc='a';
      splitit();
/*    else nqnopr(NOSPLIT);*/
      break;
     case 'n': q[cq].acc='n';
      nqnopr(NOSPLIT);
      break;
```

```
     case 'r': q[cq].acc='r';
      nqnopr(NOSPLIT);
      break;
     case 'c': q[cq].acc='c';
      nqnopr(NOSPLIT);
      break;
     }

printit();
     printf(CLEAR);
     printf("Develop another question? (y/n) ");
     gets(line);
     if(tolower(line[0] != 'y')) break;
}    /* end of master while loop */

}    /* end of genq */

/* cksplit determines if the question flow is to be split
 * if split is to occur, return 1
 * if no split, return 0
 */
cksplit()
{
printf("Should question flow be split here? (y/n) ");
gets(line);
if(tolower(line[0]) == 'y') return(SPLIT);
return(NOSPLIT);
}

/* splitit generates multiple responses for one question
 * each response may have a different next question
 */
splitit()
{
     int c, cont=1;
     switch(q[cq].acc) {
     case 'y':
      matresp(q[cq].qno,'n');
      strcpy(r[cr].rtext,"No");
      nqnopr(SPLIT);
      matresp(q[cq].qno,'y');
      strcpy(r[cr].rtext,"Yes");
      nqnopr(SPLIT);
      break;
     case 't':
      matresp(q[cq].qno,'t');
```

```
      strcpy(r[cr].rtext,"True");
      nqnopr(SPLIT);
      matresp(q[cq].qno,'f');
      strcpy(r[cr].rtext,"False");
      nqnopr(SPLIT);
      break;
     case 'a':
      while(cont) {
      printf("What response is updated next? (a-?) ");
      gets(line);
      c=tolower(line[0]);
      if('a' <= c && c <= 'z') {
       matresp(q[cq].qno,c);
       printf("Enter text of response\n%c) ",c);
       gets(line);
       strncpy(r[cr].rtext,line,sizeof(r[cr].rtext)-1);
       nqnopr(SPLIT);
      }
      printf("Enter another response for question %d? (y/n) ",
       q[cq].qno);
      gets(line);
      if(tolower(line[0]) == 'n') break;
      }
      break;
     }
}

/* get the next question number */
nqnopr(split)
int split;
{
int cont=1, nqn, n, found=0;
while(cont) {
     printf("%d) %s\n",q[cq].qno,q[cq].qtext);
     if(split) printf("%c) %s\n",r[cr].rsp,r[cr].rtext);
     else r[cr].rsp='?';
     printf("What is the next question to present? ");
     gets(line);
     if(1 != sscanf(line,"%d",&nqn))
      printf("Expecting a number!\n");
     else {  if (nqn> q[cq].qno || nqn == 0) {
      printf("Is %d the correct next question? (y/n) ",nqn);
      gets(line);
      if(tolower(line[0]=='y')) break;
      }else{
printf("Next Question Number MUST be greater than current one\n");
```

```
}       }       }
/* store next question, and add to next question list */
r[cr].nqno=nqn;
for(n=0;n<evnm;n++)
     if(nqn == evn[n]) {found++;break;}
if(found == 0) evn[evnm++]=nqn;
}    /* end of nqno prompt routine */

/* print routine for checking arrays during development */
printit()
{ int n;

printf("\nQuestions: pq=%d; cq=%d;\n",pq,cq);
for(n=0;n<pq;n++)
     printf("%d) qno=%d; prs=%c; acc=%c; qtext=%s\n",
      n,q[n].qno,q[n].prs,q[n].acc,q[n].qtext);

printf("\nEvery Question: evqm=%d; evqc=%d;\n",evqm,evqc);
for(n=0;n<evqm;n++)
     printf("%d) itno=%d; lstptr=%d;\n",
      n,evq[n].itno,evq[n].lstptr);

printf("\nResponses: pr=%d; cr=%d;\n",pr,cr);
for(n=0;n<pr;n++)
     printf("%d) qno=%d; nqno=%d; rsp=%c; rtext=%s\n",
      n,r[n].qno,r[n].nqno,r[n].rsp,r[n].rtext);

printf("\nEvery Response: evrm=%d; evrc=%d;\n",evrm,evrc);
for(n=0;n<evrm;n++)
     printf("%d) itno=%d; lstptr=%d;\n",
      n,evr[n].itno,evr[n].lstptr);

printf("\nEvery Next Question: evnm=%d; evnc=%d;\n",evnm,evnc);
for(n=0;n<evnm;n++) printf("%d) nqno=%d;\n",n,evn[n]);
printf("\n");
sleep(3);
}    /* end of routine printit */

/* matresp accepts a question number and a response
 * the number is searched for in the every response list
 * if found, the pointer to the list is returned
 * otherwise the end of the list is incremented
 */
matresp(num,c)
int num;
char c;
```

```
{
     int n, found = 0;
     int poss=-1;
     for(n=0; n < evrm; n++) {
      if( (num == evr[n].itno)
       && ( ('?' == r[evr[n].lstptr].rsp)
       ||   ('\0' == r[evr[n].lstptr].rsp) ) )
       poss = n;
      if(num == evr[n].itno
       && c == r[evr[n].lstptr].rsp ) {found++; break;}
     }
     if(found) {cr=evr[n].lstptr; return(cr);}
     if(poss != -1) {cr=evr[poss].lstptr;
      r[cr].rsp=c;
      return(cr);
     }

     cr=pr++; evr[evrm].itno=num; evr[evrm++].lstptr=cr;
     r[cr].qno=num; r[cr].rsp=c;
     return(cr);
}
/* dflt accepts a number, a max number, and a list of form qrlst
 * the number is searched for in the list
 * if found, the pointer to the list is returned
 * otherwise -1 is returned
 */
dflt(num,max,lst)
int num;
int max;
struct qrlst lst[];
{
     int n, found = 0;
     for(n=0; n < max; n++) {
      if(num == lst[n].itno) {found++; break;}
     }
     if(found) return(lst[n].lstptr);
     else return(-1);
}

/* listopen lists questions referenced as next question
 * which are not on the every question list
 * returns a count of unmatched items
 */
listopen()
{
int found, cnt=0, evqc, evnc;
```

```
printf("List of questions referenced but not developed:\n");
for(evnc=0, found=0; evnm > evnc; evnc++,found=0) {
        for(evqc=0;evqm > evqc; evqc++) {
       if(q[evqc].qno == evn[evnc]){
        found++;
        break;
      } }
      if(found !=1 ) {
       printf("%d ",evn[evnc]);
       cnt++;
}     }

if(cnt != 0) printf("\n");
return(cnt);
}     /* end of listopen */

/* load the intermediate file of questions */
intload()
{
FILE *file;
char c, *ptr;
int found;

if((file=fopen(ifname,"r")) == NULL) {
     perror(ifname);
     return(-1);
}
while ((fgets(line,MAXLINE,file)) != NULL) {
     c=line[0];
     ptr=strpbrk(line,"\n");
     *ptr='\0'; /* replace newline with string terminator */
     if (c == 'q') {
      /* load questions */
      sscanf(line,"%c%3d%c%c",
       &c,&q[pq].qno,&q[pq].prs,&q[pq].acc);
      strncpy(q[pq].qtext,line+6,MAXTXT);
      /* load every question list */
      evq[evqm].itno=q[pq].qno; evq[evqm++].lstptr=pq++;
     } else {
      /* load responses */
      sscanf(line,"%c%3d%3d%c",
       &c,&r[pr].qno,&r[pr].nqno,&r[pr].rsp);
      strncpy(r[pr].rtext,line+8,MAXTXT);

      /* load every response list */
      evr[evrm].itno=r[pr].qno; evr[evrm++].lstptr=pr++;
```

```
      /* load every next question array */
      if(evnm == 0) evn[evnm++]=r[0].nqno;
                for(found=0,evnc=0;evnc<evnm;evnc++)
       if(r[pr-1].nqno == evn[evnc]) {found++; break;}
      if(found == 0) evn[evnm++]=r[pr-1].nqno;
     }

}    /* end of while to load int file */
close(file);
return(0);
}    /* end of routine loadint */

/* validate the questionnaire
 * return 0 if valid questionnaire
 * each entry in every question array has entry in every response
 * every next question has entry in question
 */
qvalid()
{
     int found=0;
     /* insure every question has an entry in response array */
     for(evqc=0,found=0;(evqc<evqm)&&(found==0);evqc++,found=0){
     for(evrc=0;evrc<evrm;evrc++) {
      if((evq[evqc].itno == evr[evrc].itno)
      || (evq[evqc].itno == 0))
       {found=1; break;}
     }
     if (found != 1) {
      printf("Missing response for a question\n");
      return(1);
     }}
     /* insure every next question has entry in question array */
     for(cr=0,found=0;(cr<pr) && (found == 0);cr++,found=0) {
     for(evqc=0;evqc<evqm;evqc++) {
      if((r[cr].nqno == evq[evqc].itno) || (r[cr].nqno == 0) )
       {found=1; break;}
     }
     if(found != 1) {
      printf("Missing question for next question\n");
      return(1);
     }}

     return(0);
}
```

```
/* issue the questionnaire
 * get iid and cutoff date,
 * write issue file
 * build database file
 * issue dbbuild command
 */
qissue()
{
FILE *file;
int OK=0, n, m;
char cutdate[9];
printf(CLEAR);
while (OK == 0){
     strcpy(issfile,qname);
     printf("Enter (up to 4 characters) issue id of %s: ",qname);
     gets(line);
     strcat(issfile,line);
     strncpy(iid,line,sizeof(iid));
     if(strlen(issfile) < 10) OK=1;
     else printf("Total length of file name must be 10 or less\n");
}
strcat(issfile,".q");

OK=0;
printf(CLEAR);
while (OK == 0){
     printf("Enter cutoff date (yy/mm/dd): ");
     gets(line);
     strncpy(cutdate,line,sizeof(cutdate));
     printf("Are you sure %s is correct? (y/n) ",cutdate);
     gets(line);
     if(line[0] == 'y') OK=1;
}

/* store the arrays */
file=fopen(issfile,"w");
strcpy(line,"Activated by: qrun\n");
fputs(line,file);

strcpy(line,"Qname: ");
strcat(line,qname);
strcat(line,"\n");
fputs(line,file);

strcpy(line,"Instance-Id: ");
strcat(line,iid);
```

```
strcat(line,"\n");
fputs(line,file);

strcpy(line,"Cutoff-Date: ");
strcat(line,cutdate);
strcat(line,"\n");
fputs(line,file);

for(n=0;n<evqm;n++) {
     m=evq[n].lstptr;
     fprintf(file,"q%3d%c%c%s\n",
      q[m].qno,q[m].prs,q[m].acc,q[m].qtext);
}
for(n=0;n<evrm;n++) {
     m=evr[n].lstptr;
     fprintf(file,"r%3d%3d%c%s\n",
      r[m].qno,r[m].nqno,r[m].rsp,r[m].rtext);
}
fclose(file);

/* build informix database definition file */
strcpy(dbasedf,qname);
strcat(dbasedf,iid);
file=fopen(dbasedf,"w");

strcpy(line,"database ");
strcat(line,qname);
strcat(line,iid);
strcat(line,"\n");
fputs(line,file);

strcpy(line,"file ");
strcat(line,qname);
strcat(line,"\n");
fputs(line,file);

strcpy(line,"field uname type character length 15\n");
fputs(line,file);

strcpy(line,"field rdate type ydate\n");
fputs(line,file);

for(cq=0;cq<pq;cq++) {
     n=q[cq].acc;
     strcpy(line,"field qst");
     sprintf(line+9,"%d\0",q[cq].qno);
```

```
     if(n == 'y' || n == 't' || n == 'a') {
      strcat(line," type char length 1\n");
      fputs(line,file);
     }
     else if (n == 'n') (
      strcat(line," type integer\n");
      fputs(line,file);
     }
}
strcpy(line,"end\n");
fputs(line,file);
fclose(file);

strcpy(line,"DBPATH=$HOME/qmail;export DBPATH;dbbuild ");
strcat(line,dbasedf);
strcat(line,"\n");
system(line);
return(0);
}

/* blow away program when invoked with bad parms */
parmuse()
(
   printf("Useage qcreate [ [-i] <qname>]");
   return(-1);
}
```

```
/* qrun <reply to> <input file>
   program to accept input file of questions and responses
               present the questions to the user
               accept possible responses
               store responses in format for load to informix file
*/
#include <stdio.h>
#include <string.h>
#include <time.h>

/* belongs in qmail.h file */
/* ANSI standard screen control commands */
/* Clear Screen, go to row 5 column 5 */
#define CLEAR "2J5;f"
/* Restore cursor position, down 1 line */
#define SNGL  "\n"
/* Restore cursor position, down 2 lines */
#define DBLE  "\n\n"
/* the Bell */
#define BELL "\007"

#define MAXLINE 80

struct qsts  {
    int  qno;
    char prs;
    char acc;
    char qtext[72];
};
struct resp {
    int  qno;
    int  nqno;
    char rsp;
    char rtext[72];
};
/* end of items to be in qmail.h file */

/* extern items callable by several programs  */
/* count of questions and possible responses */
int pq=0, pr=0;
/* current question and possible response    */
int cq=0, cr=0;
struct qsts q[100]; /* array of questions */
struct resp r[100]; /* array of possible responses  */
char line[MAXLINE+1];
char dbload[2*MAXLINE+1]; /* array to hold output message */
```

```
char name[14], aqname[MAXLINE+1], qname[14], iid[24], cutdate[9];
char curdate[9];
int dbplc;      /* next char in dbload string */

main(argc, argv)
int argc;
char *argv[];
{
FILE *file;
int c, cmp;
int num, numok;

char *ptr;
long secs_now;
struct tm *tm_now;

/* insure two parameters beyond program name are specified */
if (argc != 3) {
   printf("%s requires exactly two parameters input\n",argv[0]);
   return(-1);
}

file=fopen(argv[2],"r");

/* obtain questionnaire name, instance id, and cut off date */
if((fgets(aqname,MAXLINE,file)) != NULL)
    sscanf(aqname,"%s%s",name,qname);
fgets(iid,MAXLINE,file);
if((fgets(line,MAXLINE,file)) != NULL)
    sscanf(line,"%s%s",name,cutdate);

/* validate cutoff date  */
time(secs_now);
tm_now=localtime(secs_now);
sprintf(line,"%2d/%2d/%2d\0",
      tm_now->tm_year,tm_now->tm_mon+1,tm_now->tm_mday);
for(c=0;c<=strlen(curdate);c++)
     curdate[c]= (line[c] == ' ') ? '0' : line[c];
if(strncmp(cutdate,curdate,strlen(curdate)) < 0){
     badmsg();
     return(-1);
}

/* begin building output record */
dbplc=0;
strcpy(dbload,"|"); /* leave a field for respondent name */
```

```
strcat(dbload,curdate);
strcat(dbload,"|");
dbplc=strlen(dbload);

/* load questions and answers
     this program requires questions and answer be in order
     by question number,  and assumes the input is in order  */
loadqr(file);
fclose(file);

/* present questions till the next question to present is number 0
          and we haven't run out of questions or responses */
while (r[cr].nqno != 0 y cq < pq y cr < pr) {

/* perform screen manipulation before presenting question */
scrprs();

/* present the question, down a line */
     printf("%d) %s\n",q[cq].qno,q[cq].qtext);

/* scan for first response for this question number */
while ( r[cr].qno != q[cq].qno) cr++;

/* prompt for acceptable response */
switch (q[cq].acc) {

    case 'n': /* accept numeric answer */
     /* loop till user responds with a number */
     numok = -1;
    while(numok != 1) {
      getresp("Enter an integer");
         numok=sscanf(line,"%d",num);
        }
     matresp(num);
     break;

    case 'y': /* accept yes or no */
     c=' ';
     while(c != 'n' y c != 'y') {
      getresp("Enter y for yes, or n for no");
         c=line[0];
      c=tolower(c);
    }
     matresp(c); /* process matching response */
     break;
```

```
    case 't': /* accept true or false */
     c=' ';
     while(c != 't' y c != 'f') {
      getresp("Enter t for true, f for false");
         c=line[0];
      c=tolower(c);
     }
     matresp(c);
     break;

    case 'a': /* accept multiple choice a - ? */
    cmp=mltresp();
     c=' ';
     while(c < 'a' || c > cmp) {
      getresp("Select one of the above");
         c=tolower(line[0]);
     }
     matresp(c);
     break;

    case 'r': /* accept <return> (or any character) */
     getresp("Enter any character to continue");
     /* intentionally do not break here,
        must be before case 'c'  */

    case 'c': /* continue immediately with response */
    default:
     /* scan for matching next question number */
     while(q[cq].qno<r[cr].nqno) cq++;

}    /* end of case to prompt for appropriate responses */

}    /* end of while loop to present questions */

     /* fill out dbload */
     cr=pr-1;
     r[cr].nqno=r[cr].qno;
     fillrec();

{ /* write prepared answer to output file */

     int i=0, c;
     char mailback[64], fname[24];

     strcpy(fname,"/tmp/Rar");
     sprintf(fname+8,"%d",getpid());
```

```
     file=fopen(fname,"w");

     strcpy(mailback,"itmail -s Activate_qload ");
     strcat(mailback,argv[1]);
     strcat(mailback," <");
     strcat(mailback,fname);
     strcat(mailback,";");

     fputs("Activated by: qload\n",file);
     ptr=strpbrk(aqname,"\n");
     /* replace newline with string terminator */
     *(ptr+1)='\0';
     fputs(aqname,file);

     ptr=strpbrk(iid,"\n");
     /* replace newline with string terminator */
     *(ptr+1)='\0';
     fputs(iid,file);

     ptr=strpbrk(dbload,"\n");
     /* replace newline with string terminator */
     *(ptr+1)='\0';
     fputs(dbload,file);

     fclose(file);

     system(mailback);
     unlink(fname);
}    /* end of block which mails message back */

return(0);
}    /* end of main for qrun */

/* getresp gets keyboard input from a user
   prompting with prmpt
   storing input in line                    */
getresp(prmpt)
char prmpt[];
{
     printf(BELL);
     printf("   %s: ",prmpt);
     gets(line);
}    /* end of getresp */

/* mltresp displays possible response to a multiple choice question
   and returns the highest alpha character
```

```
         which is an acceptable response */
mltresp()
{
int c, tptr;
tptr=cr;
c='a';
while(r[tptr].qno==q[cq].qno) {
    printf("   %c) %s\n",c,r[tptr].rtext);
    tptr++; c++;
    }
c--;
return(c);
}    /* end of mltresp */

/* matresp takes a user's response to a question
   from the possible responses to this question,
       a match (or default match) is found
   and question pointer is positioned
         to the next question to present
       response pointer to matching response of question just asked
   finally store responses in output array */
matresp(c)
int c;
{
int rtn;
/* scan past responses with same question number which
   are not default match
   and don't match user's response */
while(r[cr].qno==q[cq].qno && r[cr].rsp != '?' && r[cr].rsp != c)
     cr++;

/* at this point either have match,
     or no more of possible responses */

/* cr now points to best possible match to user response
   store it in output array for load into database       */
if(q[cq].acc != 'n') /* put in the just obtained response */
    dbload[dbplc++]=c;
else { /* special handling for numerics */
    dbplc+=sprintf(dbload+dbplc,"%d",c);
/*    while(dbload[dbplc] != '0') dbplc++;                */
}

/* if matched, scan for next question to present
     based on next question fld
   other wise, scan for next question to present
```

```
     based on last response with matching question number */
if(r[cr].qno == q[cq].qno)
     rtn=0;
else {
     cr--;
     rtn=1;}
fillrec();

return(rtn);
}       /* end of matresp */

/*
   filling dbload with appropriate defaults */
fillrec()
{
     int c;

     /* separate response just accepted */
     cq++;
     if (cr != (pr-1)) {
      dbload[dbplc++]='|';
      dbload[dbplc]='\n';}

     /* scan past unused questions, fill with defaults */
     while(q[cq].qno < r[cr].nqno) {
      c = q[cq].acc;
      if (c == 'n')
       dbload[dbplc++]='0';
      if (c != 'r' y c != 'c') {
       dbload[dbplc++]='|';
       dbload[dbplc]='\n';
      }
      cq++;
     }
     return(0);
}

/* scrprs manipulates the screen per instructions
   in the current question presentation control field   */
scrprs()
{
    if(q[cq].prs==' ') printf(CLEAR);
    if(q[cq].prs=='d') printf(DBLE);
    if(q[cq].prs=='s') printf(SNGL);
}    /* end of scrprs */
```

```
/* loadqr loads the questions and possible responses
          from the input file
       to the question and response arrays          */
loadqr(file)
FILE *file;
{
char c, *ptr, line[MAXLINE];
while ((fgets(line,MAXLINE,file)) != NULL) {
    c=line[0];  /* to test for question or answer */
    ptr=strpbrk(line,"\n");
    *ptr='\0';  /* replace newline with string terminator */
    if (c == 'q') {
     /* load questions into question array */
        sscanf(line,"%c%3d%c%c",c,q[pq].qno,q[pq].prs,q[pq].acc);
      strncpy(q[pq].qtext,line+6,72);
        pq++;
    }
    else {
     /* load possible responses into response array */
        sscanf(line,"%c%3d%3d%c",c,r[pr].qno,r[pr].nqno,r[pr].rsp);
      strncpy(r[pr].rtext,line+8,MAXLINE);
        pr++;
    }
}    /* end of while loop to load input into arrays */
}    /* end of loadqr */

badmsg()
{
printf("Thank you for trying to respond to the questionnaire %s\n",
       qname);
printf("Unfortunately, answers are not being accepted after %s\n",
       cutdate);
printf("Please delete this message\n");
}
```

```
/* qload <reply to> <input file>
   program to accept input file of questionnaire responses
         add users name to beginning of load record
              load response into informix data base
*/
#include <stdio.h>
#include <string.h>
#define MAXLINE 81

/* extern items callable by several programs  */

main(argc, argv)
int argc;
char *argv[];
{
FILE *file;
int c, pid;
char line[MAXLINE], qname[11], iid[11], name[24];
char dbload[2*MAXLINE], fcname[24], flname[24];

/* insure two parameters beyond program name are specified */
if (argc != 3) {
   printf("%s requires exactly two parameters input\n",argv[0]);
   return(-1);
}

pid=getpid();

/* process input file */
file=fopen(argv[2],"r");

/* obtain questionnaire name and instance id*/
if((fgets(line,MAXLINE,file)) != NULL)
    sscanf(line,"%s%s",name,qname);
if((fgets(line,MAXLINE,file)) != NULL)
   sscanf(line,"%s%s",name,iid);

/* build output record */
strcpy(dbload,argv[1]); /* insert respondent name */
fgets(dbload+strlen(dbload),2*MAXLINE,file);
fclose(file);
/* finished with input file */

/* build file to load */
strcpy(flname,"/tmp/RL");
sprintf(flname+7,"%d",pid);
```

```
file=fopen(flname,"w");
fputs(dbload,file);
fclose(file);

/* build command file */
strcpy(fcname,"/tmp/RC");
sprintf(fcname+7,"%d",pid);
file=fopen(fcname,"w");
strcpy(line,"load ascii ");
strcat(line,qname);
strcat(line," from \"");
strcat(line,flname);
strcat(line,"\"\n");
fputs(line,file);
fputs("b\n",file);
fclose(file);

/* build command to run*/
strcpy(line,"DBPATH=$HOME/qmail;export DBPATH;dbstatus ");
strcat(line,qname);
strcat(line,iid);
strcat(line," < ");
strcat(line,fcname);
strcat(line,"\n");
system(line);

unlink(fcname);
unlink(flname);

return(0);
}    /* end of main for qrun */
```

diff of itmail and mailx cmd1.c source files

```
292,301d291
< /* special version of type for IT
<  * Type out messages, honor ignored fields.
<  */
< typeit(msgvec)
<     int *msgvec;
< {
<
<     return(typelit(msgvec, 1));
< }
<
327,386d316
< typelit(msgvec, doign)
<     int *msgvec;
< {
<     register *ip;
<     register struct message *mp;
<     register int mesg;
<     register char *cp;
<     int c, nlines;
<     int brokpipe();
<     FILE *ibuf, *obuf;
<     int (*saveint)();
<
<     saveint = signal(SIGINT, SIG_IGN);
<     obuf = stdout;
<     if (setjmp(pipestop)) {
<          if (obuf != stdout) {
<               pipef = NULL;
<               pclose(obuf);
<          }
<          goto ret0;
<     }
<     if (intty && outtty && (cp = value("crt")) != NOSTR) {
<     for (ip = msgvec,nlines=0; *ip && ip-msgvec < msgCount;ip++)
<               nlines += message[*ip - 1].m_lines;
<          if (nlines > atoi(cp)) {
<               obuf = popen(PG, "w");
<               if (obuf == NULL) {
<                    perror(PG);
<                    obuf = stdout;
<               }
<               else {
<                    pipef = obuf;
<                    sigset(SIGPIPE, brokpipe);
<               }
```

diff of itmail and mailx cmd1.c source files

```
<         } else
<             signal(SIGINT, saveint);
<     } else
<         signal(SIGINT, saveint);
<     for (ip = msgvec; *ip && ip-msgvec < msgCount; ip++) {
<         mesg = *ip;
<         touch(mesg);
<         mp = &message[mesg-1];
<         dot = mp;
<         printit(mp, obuf, doign);
<     }
<     if (obuf != stdout) {
<         pipef = NULL;
<         pclose(obuf);
<     }
< ret0:
<     sigset(SIGPIPE, SIG_DFL);
<     signal(SIGINT, saveint);
<     return(0);
< }
<
< /*
<  * Type out the messages requested.
<  */
< jmp_buf pipestop;
<
455,469d384
< }
<
< /* special version of print for IT
<  * Print the indicated message on standard output.
<  */
<
< printit(mp, obuf, doign)
<     register struct message *mp;
<     FILE *obuf;
< {
<
<     if (!doign || !isign("message"))
<         fprintf(obuf, "Message %2d:\n", mp - &message[0] + 1);
<     touch(mp - &message[0] + 1);
<     sendit(mp, obuf, doign);
```

diff of mailx and itmail cmd2.c source files

```
33a34
>
98c99
<     return(typeit(list));
---
>     return(type(list));
```

diff of itmail and mailx main.c source files

```
0a1
>
274,275d274
<
<     /* tinit defined in temp.c,
<         gives names to temporary files needed */
277d275
<
280,282d277
<
<     /* -n specified do not source /usr/lib/mailx/mailx.rc */
<     /* load defined in lex.c to load a file of user definitions */
```

diff of itmail and mailx send.c source files

```
8c8,9
< #include <string.h>
---
> #
>
14d14
< #include <sys/stat.h>
31d30
<
144,357d142
< /* special version of send for IT extension
<  * Send message described by the passed pointer to the
<  * passed output buffer.  Return -1 on error, but normally
<  * the number of lines written.  Adjust the status: field
<  * if need be.  If doign is set, suppress ignored header fields.
<  */
< sendit(mailp, obuf, doign)
<    struct message *mailp;
<    FILE *obuf;
< {
<    register struct message *mp;
<    register int t;
<    unsigned int c;
<    FILE *ibuf;
<    char line[LINESIZE], field[BUFSIZ];
<    int lc, ishead, infld, fline, dostat;
<    char *cp, *cp2;
<    int oldign = 0;     /* previous line was ignored */
<
<    /* beginning of hooks for IT extension */
<    int active=0;  /* flags this as an active message */
<    int actchk=0;  /* flags when to check for active message */
<    int shlrtn;         /* flags inability to activate command */
<    struct stat statc;
<
<    char cmd[2*LINESIZE];    /* command to activate */
<    char reply[LINESIZE];    /* store from information here */
<    char actfail[LINESIZE];  /* command to reply failure */
<
<    /* name of file to hold activation input */
<    extern char tempAct[];
<    FILE *nfo;                    /* pointer to above temp file */
<    strcat(actfail,"actfail ");   /* init actfail */
<    /* end of this section of hooks for IT extension */
<
<    mp = mailp;
```

```
<    ibuf = setinput(mp);
<    c = msize(mp);
<    ishead = 1;
<    dostat = 1;
<    infld = 0;
<    fline = 1;
<    lc = 0;
<    clearerr(obuf);
<    while (c > 0) {
<         fgets(line, LINESIZE, ibuf);
<         c -= strlen(line);
<         lc++;
<         if (ishead) {
<              /*
<               * First line is the From line, so no headers
<               * there to worry about
<               */
<              if (fline) {
<                   fline = 0;
<    /* IT extension
<         capture who its from in reply
<         from will be passed as parm to activation command
<         in case reply is necessary */
<                   strcpy(reply,line+5);
<                   *strpbrk(reply," ")='\0';
<    /* end IT extensions */
<
<                   goto writeit;
<              }
<              /*
<               * If line is blank, we've reached end of
<               * headers, so force out status: field
<               * and note that we are no longer in header
<               * fields
<               */
<              if (line[0] == '\n') {
<                   if (dostat) {
<                        statusput(mailp, obuf, doign);
<                        dostat = 0;
<                   }
<
<    /* IT extension
<         set flag to check for active mail identifier
<         in first line of body */
<                   actchk++;
<
```

```
<                   ishead = 0;
<                   goto writeit;
<              }
<              /*
<               * If this line is a continuation
<               * of a previous header field, just echo it.
<               */
<              if (isspace(line[0]) && infld)
<                   if (oldign)
<                        continue;
<                   else
<                        goto writeit;
<              infld = 0;
<              /*
<               * If we are no longer looking at real
<               * header lines, force out status:
<               * This happens in uucp style mail where
<               * there are no headers at all.
<               */
<              if (!headerp(line)) {
<                   if (dostat) {
<                        statusput(mailp, obuf, doign);
<                        dostat = 0;
<                   }
<                   putc('\n', obuf);
<                   ishead = 0;
<                   goto writeit;
<              }
<              infld++;
<              /*
<               * Pick up the header field.
<               * If it is an ignored field and
<               * we care about such things, skip it.
<               */
<              cp = line;
<              cp2 = field;
<              while (*cp y *cp != ':' && !isspace(*cp))
<                   *cp2++ = *cp++;
<              *cp2 = 0;
<              oldign = doign && isign(field);
<              if (oldign)
<                   continue;
<         /*
<          * If the field is "status," go compute and print the
<          * real Status: field
<          */
```

```
<               if (icequal(field, "status")) {
<                    if (dostat) {
<                         statusput(mailp, obuf, doign);
<                         dostat = 0;
<                    }
<                    continue;
<               }
<          }
<
<          /* IT extensions,
<               identify as active message
<                    save activation command
<                    open output file to save rest of message */
<          if (actchk) {
<               actchk=0; /* dont check twice */
<               fline++;  /* Activated by: line not output */
<               if (strncmp(line,"Activated by: ",14) == 0) {
<                    strncpy(cmd,"/usr/lbin/mail/",16);
<                    strcat(cmd,line+14); /* command to activate */
<                    *strpbrk(cmd,"\n")='\0'; /* find newline */
<
<                    /* verify command exists and executable */
<                    if(stat(cmd,sstatc) <0) {
<                    printf("Unable to Find: %s\nListing follows\n",
<                         cmd);
<                         goto writeit;
<                    }
<     if((statc.st_mode & (S_IFMT | S_IXOTH)) != (S_IFREG | S_IXOTH))
<     {    printf("%s not executable\nListing follows\n",cmd);
<          goto writeit;
<     }
<
<                    if ((nfo = fopen(tempAct, "w")) == NULL) {
<                         perror(tempAct);
<                         goto writeit;
<                    }
<                    active++;                /* flag as active */
<               }
<          }    /* end this portion IT extensions */
<
< writeit:
<          if (!active){
<          fputs(line, obuf);
<               if (ferror(obuf))
<                    return(-1);
<          }
```

diff of itmail and mailx send.c source files

```
<          /* IT modifications,
<               copy rest of message after "Activated by: "
<               to tempAct file for input to activated program */
<          else {
<               if(fline)  fline=0;
<               else {
<                    fputs(line, nfo);
<                    if (ferror(nfo))
<                         return(-1);
<               }
<          }    /* end of IT modifications */
<     }
<
<     /* IT modifications,
<          close file containing input for active program
<          insure command exists in /usr/lbin/mail
<          specify reply as parm into command
<          specify tempAct as input to command
<          activate command
<          remove input file on completion */
<
<     if (active){
<          active=0;
<          fclose(nfo);          /* close tempAct file */
<          if (ferror(obuf))
<               return(-1);
<          strcat(cmd," ");     /* add parm containing reply info */
<          strcat(cmd,reply);
<          strcat(actfail,reply);
<          /* add tempAct as 2nd parm to command */
<          strcat(cmd," ");<          strcat(cmd,tempAct);
<          shell(cmd);                /* activate the command */
<          remove(tempAct);     /* remove tempAct file */
<     }
<     /* end of IT modifications */
<
<     if (ishead && (mailp->m_flag DMSTATUS))
<          printf("failed to fix up status field\n");
<     return(lc);
< }
<
623c408
< #endif /* CC */
---
> #endif CC
```

diff of itmail and mailx sigretro.c source files

```
10c10
< #include "signal.h"
---
> #include <signal.h>
13c13
< #include <sigretro.h>
---
> #include "sigretro.h"
79a80,224
> }
>
> /*
>  * Set the (permanent) disposition of a signal.
>  * If the signal is subsequently (or even now) held,
>  * the action you set here can be enabled using sigrelse().
>  */
> sigtype
> sigset(sig, func)
>    sigtype func;
> {
>    sigtype old;
>    extern int errno;
>
>    if (sig < 1 || sig > NSIG) {
>         errno = EINVAL;
>         return(BADSIG);
>    }
>    old = sigdisp(sig);
>    /*
>     * Does anyone actually call sigset with SIG_HOLD!?
>     */
>    if (func == SIG_HOLD) {
>         sighold(sig);
>         return(old);
>    }
>    sigtable[sig].s_flag |= SSET;
>    sigtable[sig].s_func = func;
>    if (func == SIG_DFL) {
>         /*
>          * If signal has been held, must retain
>          * the catch so that we can note occurrance
>          * of signal.
>          */
>         if ((sigtable[sig].s_flag & SHELD) == 0)
>              signal(sig, SIG_DFL);
>         else
```

```
>              signal(sig, _Sigtramp);
>          return(old);
>     }
>     if (func == SIG_IGN) {
>          /*
>           * Clear pending signal
>           */
>          signal(sig, SIG_IGN);
>          sigtable[sig].s_flag = ~SDEFER;
>          return(old);
>     }
>     signal(sig, _Sigtramp);
>     return(old);
> )
>
> /*
>  * Hold a signal.
>  * This CAN be tricky if the signal's disposition is SIG_DFL.
>  * In that case, we still catch the signal so we can note it
>  * happened and do something crazy later.
>  */
> sigtype
> sighold(sig)
> {
>     sigtype old;
>     extern int errno;
>
>     if (sig < 1 || sig > NSIG) {
>          errno = EINVAL;
>          return(BADSIG);
>     }
>     old = sigdisp(sig);
>     if (sigtable[sig].s_flag & SHELD)
>          return(old);
>     /*
>      * When the default action is required, we have to
>      * set up to catch the signal to note signal's occurrance.
>      */
>     if (old == SIG_DFL) {
>          sigtable[sig].s_flag |= SSET;
>          signal(sig, _Sigtramp);
>     }
>     sigtable[sig].s_flag |= SHELD;
>     return(old);
> }
>
```

```
> /*
>  * Release a signal
>  * If the signal occurred while we had it held, cause the signal.
>  */
> sigtype
> sigrelse(sig)
> {
>     sigtype old;
>     extern int errno;
>     int _Sigtramp();
>
>     if (sig < 1 || sig > NSIG) {
>          errno = EINVAL;
>          return(BADSIG);
>     }
>     old = sigdisp(sig);
>     if ((sigtable[sig].s_flag & SHELD) == 0)
>          return(old);
>     sigtable[sig].s_flag &= ~SHELD;
>     if (sigtable[sig].s_flag & SDEFER)
>          _Sigtramp(sig);
>     /*
>      * If disposition was the default, then we can unset the
>      * catch to _Sigtramp() and let the system do the work.
>      */
>     if (sigtable[sig].s_func == SIG_DFL)
>          signal(sig, SIG_DFL);
>     return(old);
> }
>
> /*
>  * Ignore a signal.
>  */
> sigtype
> sigignore(sig)
> (
>
>     return(sigset(sig, SIG_IGN));
> }
>
> /*
>  * Pause, waiting for sig to occur.
>  * We assume LUSER called us with the signal held.
>  * When we got the signal, mark the signal as having
>  * occurred.  It will actually cause something when
>  * the signal is released.
```

```
>  *
>  * This is probably useless without job control anyway.
>  */
> sigpause(sig)
> {
>     extern int errno;
>
>     if (sig < 1 || sig > NSIG) {
>          errno = EINVAL;
>          return;
>     }
>     sigtable[sig].s_flag |= SHELD|SPAUSE;
>     if (setjmp(_pause) == 0)
>          pause();
>     sigtable[sig].s_flag &= ~SPAUSE;
>     sigtable[sig].s_flag |= SDEFER;
```

diff of itmail and mailx temp.c source files

```
28d27
< char    tempAct[14]; /* added for active mail */
46d44
<     sprintf(tempAct, "/tmp/Ra%-d", pid);    /* for active mail */
```

diff of itmail and mailx sigretro.h source files

```
24a25
> #define BADSIG          (int (*)()) -1 /* Return value on error */
26c27
< #endif /* SIG_HOLD */
---
> #endif SIG_HOLD
28,30c29
< #endif /* SIGRETRO */
<
< #define BADSIG          (int (*)()) -1 /* Return value on error */
---
> #endif SIGRETRO
```

IT- AN ACTIVE MESSAGE EXTENSION TO THE UNIX™ MAILX COMMAND

by

Ronald Richard Dailey

B.S. Applied Math, University of Colorado, 1970

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

Master of Science

Department of Computer Science

Kansas State University
Manhattan, Kansas

1988

An Abstract of the Master's Report

The Identify and Transact (IT) extension to the UNIX™ mailx command triggers the execution of a program specified by the body of a mail message upon selection by the recipient. The rest of the body is used as input to the activated program.

Active messages can and should be tools which enhance the capabilities of a computer based message system. This implementation demonstrates the viability of active messages by providing tools for the creation, receipt, and tabulation of questionnaires.

IT consists of modifications to the mailx source code. The questionnaire tool consists of three programs. The first program aids creation of questionnaires. The second program interactively presents the questionnaire to the recipient, gathering responses, and formulating a reply which is returned to the originator. The third program loads the responses to a questionnaire into a data base.

The integrity of a host computer system which receives an active message is addressed through several design decisions. Special attention is paid to the potential misuse of active messages, both in today's environment, and in the environment anticipated to exist as developments in information technology continue.