1983

Conversation-Based Mail

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

Larry J. Peterson

Report Number:
83-465

Conversation-Based Mail

Tilde Report CSD-TR-465

March 17, 1984

Douglas E. Comer
Larry L. Peterson

ABSTRACT

We describe a user interface to computer mail based on conversations among individuals. Conversations provide a higher level organization to messages than conventional memo-based mail systems. Messages in a memo-based system are treated independently, and presented to the user ordered by their arrival at the user's mail-box. Any relationship that may exist between memos must be explicitly observed and managed by the user. In contrast, a conversation-based system organizes mail before the user sees it by grouping messages into conversations. In addition, messages within a conversation are ordered based on the *context* in which they were written.

This project is supported in part by grants from the National Science Foundation (MCS-8219178), SUN Microsystems Incorporated, and Digital Equipment Corporation.

1. Introduction

Computer mail has gained acceptance as a viable medium for communication among individuals. It offers the advantages of written communication at electronic speeds, while remaining less expensive than telephone communication.

Conceptually, conventional electronic mail can be viewed in the following manner. A sender first composes a message with the assistance of a *User Interface* (*UI*). The user interface then submits the message to the *Message Transport System* (*MTS*) for delivery to the destination. The *MTS*, which may span one or more machines, deposits the message in the recipient's *mail-box* where it remains until accessed. The recipient retrieves the message with the assistance of a user interface. The sender identifies the recipient by specifying a *mail-box address*; the sender encloses a return address with the message, enabling the recipient to reply [HEND79, LEVI79, PICK79, SCHK81].

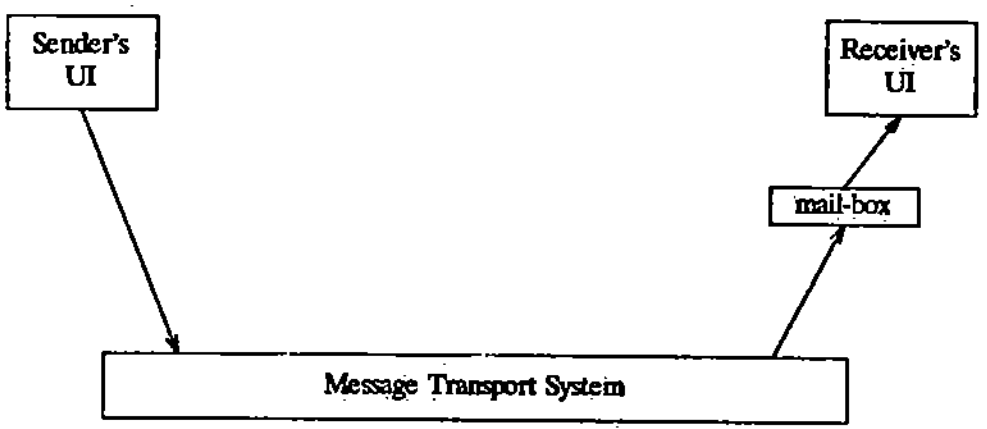

Figure 1.1
Logical View of Computer Mail

The message transport system is implemented as a collection of delivery subsystems. The users on a single computer system share a *local* mail delivery system. If computer systems are connected by packet-switched networks, the local delivery systems may cooperate to form a

network–wide delivery system. Because computer systems are often connected to more than one network, network delivery systems may overlap to form an *internet* mail system. Thus, while the users of the mail system view the *MTS* as a single logical entity, it actually consists of a confederation of interconnected delivery systems. Figure 12 shows an example of three computer systems, *X*, *Y*, and *Z*, all connected to more than one network, forming a transport system which spans networks *A*, *B*, and *C*.

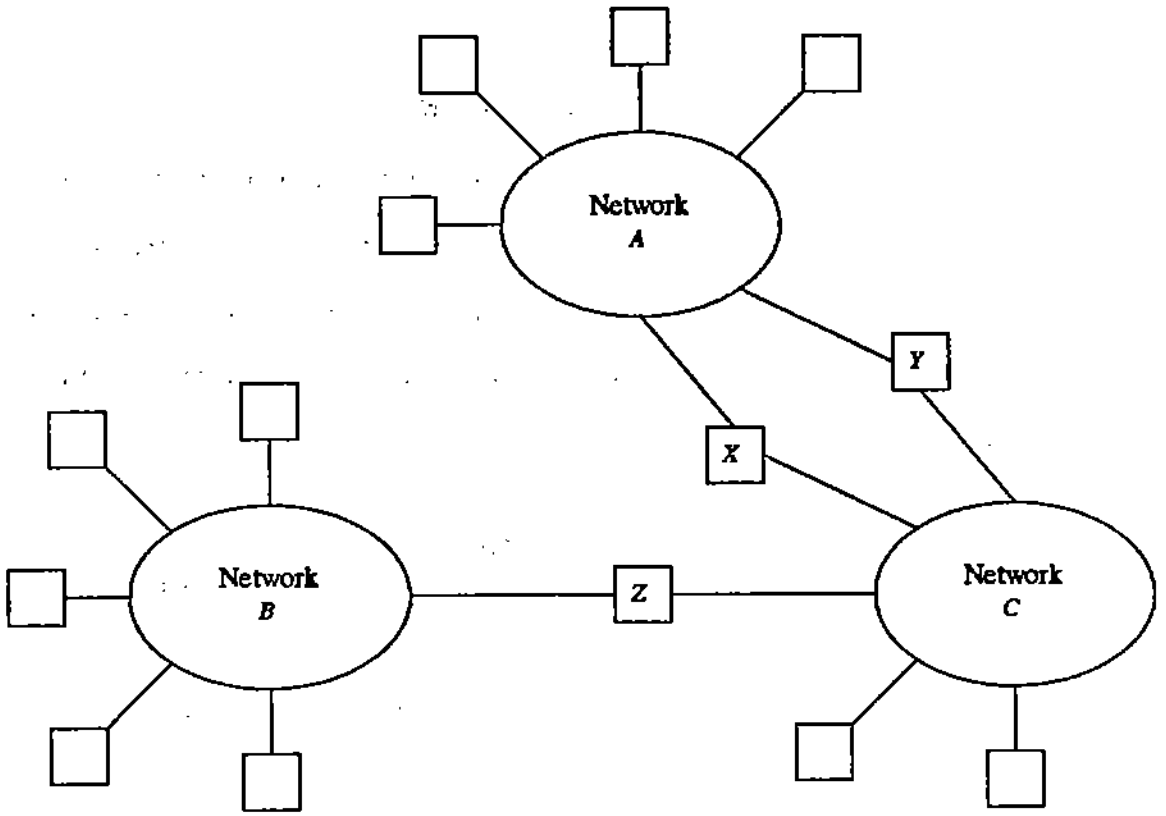

Figure 1.2
Physical view of the *MTS*

In order to deliver messages through the *MTS*, the component computer systems implement a hierarchy of *mailers* as illustrated in Figure 1.3. When the sender deposits a message for delivery, an *internet–mailer* [ALLM83, CROC79, POST79] accepts it, and either passes the message to a *local–mailer* [SHOE79] for final delivery to a mail-box, or invokes a suitable *network–mailer* [POST82, SCHM79] to deliver the message to a remote system.

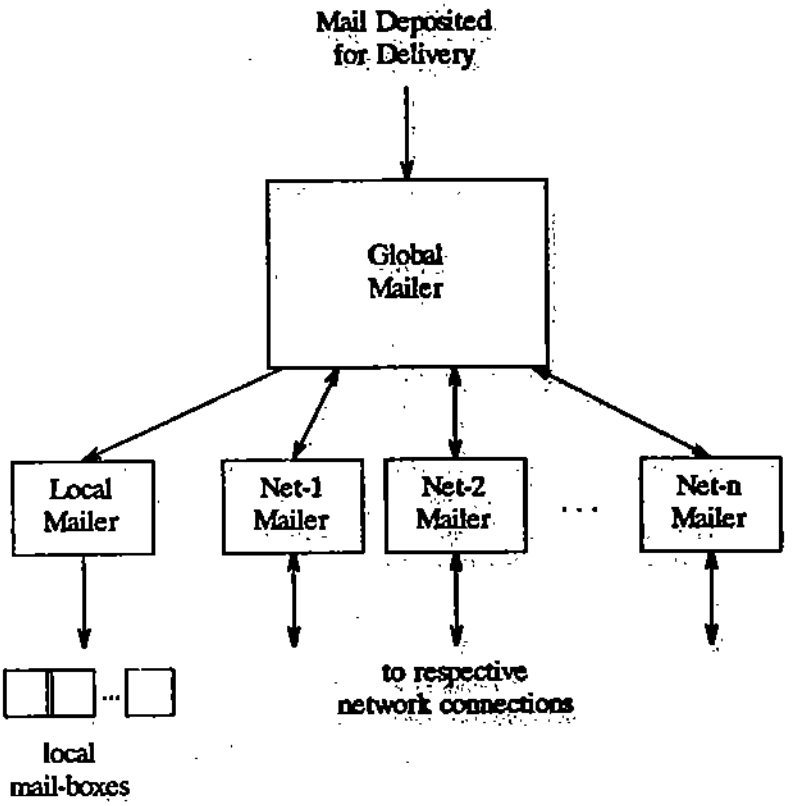

Figure 13
Mailers on One System

This process may be repeated at a number of intermediate computer systems, with the internet-mailer at each system selecting the next network to forward the message through. When the ultimate destination is reached, its local-mailer is invoked to complete the delivery.† The mailer at each step accomplishes its delivery task by interpreting the address of the recipient's mail-box [SHOC78, DENN81]. In the example in Figure 12, messages from a computer on network *A* must travel through the intermediate machines *X* or *Y*, and *Z* to reach a mail-box on a system in network *B*.

†In some cases, such as GRAPEVINE [BIRR82], the network-mailer is able to complete the delivery as mail-boxes are not bound to a single computer system.

We now turn our attention to the user interface. The transport system just described provides an enormous flow of information among individuals. It is the purpose of the user interface to assist the individual in the management of this information.

Many user interfaces currently exist, [BORD79, BROT81, CROC77, MYER76, SHOE79]. They provide varying degrees of sophistication for viewing, composing and archiving messages. They are, however, all based on *memo communication*, an idea founded in conventional office practices. Each memo that arrives for an individual is treated independently from all other memos. Memos are displayed in the order they arrive. Any relationships that may exist between memos must be explicitly observed and managed by the user; the user interface provides little help in gathering together related memos.

This paper describes a new approach to electronic mail that replaces the concept of independent memos with a higher level concept of **conversations** among a group of individuals. Conversations provide a uniform scheme for managing both new and old messages. We begin by listing a set of goals for a user interface in the next section. We then explain the attributes of conversation-based mail in section three, and present a formal description of the significant components in section four. In section five, we discuss the implementation of a conversation-based system in a distributed environment. Finally, in section six we comment on the design of a prototype conversation-based mail system.

2. Objectives

The purpose of a user interface is to aid in the organization of a user's electronic mail. It should present messages to the user in an easily digestible and comprehendable fashion. To this end, we list the specific goals we believe a user interface should meet.

- Messages should be grouped together into conversations when presented to the user. Conversations are more consistent with the way humans communicate.

- All forms of mail-like services such as bulletin boards [LIPT74] and news should be incorporated into a single, uniform system.

- The order in which groups of related messages are presented to the user should correspond to the importance of the messages. The user should be able to determine what mail is urgent so he can process it first.

- Mail should be archived so that it can be easily referenced. The user should be able to query the mail archive for all messages discussing a particular subject, from a certain person, etc.

- The history of a conversation should be maintained to allow new users to be included in a discussion.

- The context in which a message was written should be available. That is, there should be a mechanism for determining what messages the sender viewed before composing a given message. Furthermore, the user interface should use the message context to present messages in a meaningful order.

- Messages and conversations should be displayed in a clean and easily-readable format. Multiple windows should be used so that relevant information can be easily extracted, and messages should not be cluttered with unnecessary header information.

- Irrelevant and unnecessary messages should be eliminated so as to reduce the volume of mail with which the user must deal. For example, messages should have expiration dates so that out-of-date notices are automatically deleted. Also, it should be possible to replace a group of messages with a single summarizing message.

- The user interface should display names in a form acceptable to the user so that the participants in a conversation are easily identifiable.

3. System Attributes

This section focuses on the attributes of conversation-based mail by describing a set of operations for managing conversations and messages. The fundamental object of communication is the *conversation* rather than the memo. Instead of reading, composing, and filing individual memos, the user participates in a set of conversations.[†] Conversations provide a grouping to messages that is closer to natural forms of human communication.

[†]Whether a conversation restricts itself to a single topic or not is a decision made by the participants. It is envisioned, however, that conversations will be restricted to a single topic as users are free to have more than one conversation with the same participants at the same time.

We now present a high-level description of the commands available for manipulating conversations. In order to discuss these commands, we classify them into three *levels* or *modes* of operation:

(1) Operations for managing a collection of conversations;
(2) Operations for managing the contents of a single conversation;
(3) Operations for manipulating individual messages;

For each class, we first list a set of commands, and then give an informal semantic description of each. A formal model of the fundamental components is developed in the next section.

3.1. Operations on Sets of Conversations

The first group of commands are used to manage a set of conversations belonging to a given user.

- **LIST-CONV**
- **SELECT-CONV**(*conversation*)
- **MERGE-CONV**($conversation_1, \cdots, conversation_n$)
- **SPLIT-CONV**(*conversation*)
- **CREATE-CONV**(*topic, participants,* $message_{initial}$)
- **DELETE-CONV**(*conversation*)
- **QUERY**(*attribute, value*)
- **EXIT**

At this level, the user can list the known conversations, as well as select one for further consideration. Selecting a conversation causes the conversation level described in section 3.2 to be entered. In addition, new conversations can be created and old ones deleted. When a new conversation is created, the user must supply a topic for the conversation along with a list of participants. The message level described in section 3.3 is then entered to aid the user in the creation of the conversation's initial message.

In contrast to a memo-based system where messages are perceived to be either "new" or "old", the set of conversations that a user is involved in is partitioned into *foreground* and

background subsets. A conversation is in the foreground if a participant has acted on it in the past *n* days, where *n* is a parameter of the system. Foreground conversations are further partitioned into those that contain messages not yet seen by the user, and those in which the user is up-to-date.

In addition to the implicit partition of conversations into foreground and background subsets, an explicit mechanism is provided for generating a subset of conversations. All of a user's conversations are stored in a private database. The user may query this database for a set of conversations by specifying an attribute-value pair, where appropriate attributes are conversation *topic* and *participant*. One of the conversations produced by the query, and in general, a single conversation from any set of conversations, may be selected for further consideration by the user. The database can also be queried using attributes that cut across conversation boundaries, such as message *subject*, *date*, and *content*. In this case, a set of messages is produced rather than a set of conversations. (Note that a conversation has a *topic*, and each message in a conversation has a *subject*.)

Additional commands allow the user to control the direction of conversations. A conversation that has diverged into a number of subtopics may be divided, and two or more overlapping conversations can be merged together. Splitting a conversation means that two new identical conversations are created to replace the original conversation. Merging a set of conversations implies that the new conversation consists of the union of the messages in the original conversations.

Finally, basing mail on conversations leads to a scheme for incorporating all forms of mail-like activities into one format. For example, current system-wide bulletin boards can be replaced by a single conversation, with all users on the system as participants. Similarly, each news group that a user belongs to can be managed as a conversation. In the case of news, participants can be classified as either *active* or *passive*, where passive participants are not

allowed to contribute to a conversation.

3.2. Operations on a Single Conversation

The following group of commands provides a means to manipulate a single conversation.

- **LIST-MSGS**
- **LIST-PARTS**
- **SELECT-MSG**(*message*)
- **CONTEXT-MSG**(*message*)
- **SUBMIT-MSG**(*subject*, $message_{new}$)
- **ADD-PART**(*participant*)
- **REMOVE-PART**(*participant*)
- **SUPERSEDE-MSGS**($message_1$, $\cdots$, $message_n$, $message_{new}$)
- **EXIT**

The user can list the messages in the conversation, as well as add and delete both messages and participants. Additional mechanisms are provided for focusing on a single message.

Once the user selects a specific conversation for viewing, information about that conversation is displayed. This information includes a brief synopsis of the unread messages, with high-priority messages highlighted. The underlying structure of messages in a conversation provides a comprehensive history of conversations so that new users can be included. It also supplies current members with information concerning the *context* of a given message — how that message fits into an ongoing discussion. The next section presents a model for this underlying structure.

In addition, a mechanism for removing unwanted messages from a conversation is supported. The operation gives participants the ability to supersede a group of messages in a conversation with a single replacement message. Such a mechanism allows a participant to summarize a discussion, and exchange that summary for the individual messages which led to it. The superseded messages still exist, but are *hidden* and not automatically displayed with the conversation. This command allows participants in a conversation control over the volume of

irrelevant information that inevitably accumulates in computer mail.

Both the SUBMIT and the SUPERSEDE command imply that a new message is to be added to the conversation. Consequently, both cause the message level described in section 3.3 to be entered.

3.3. Operations on a Single Message

Finally, we discuss the operations that can be performed on individual messages. Once a conversation and message have been selected using the operations defined above, the user can display the contents of that message and form replies. The following is a list of commands that can be performed on a single message.

- **DISPLAY**
- **COMPOSE**
- **ATTACH**(*file*)
- **DETACH**(*file*)
- **SET-PARAM**(*parameter*, *value*)
- **EXIT**

Unlike conventional memo-based systems which display a large number of header lines, (even some dealing with message transport), conversation-based mail displays only relevant information such as the message author, subject, and date, in addition to the message body.

When submitting a message to a conversation, the participant generates the message by first composing the message body. The user may then set a number of parameters associated with that message. For example, a *time-fuse* parameter may be set. Normally, a message has an arbitrary lifetime. Defining a time-fuse, however, causes it to be removed from the conversation at a specified time. This keeps out-of-date messages from cluttering conversations.

The composer can also assign a high *priority* to the message so it will be marked as urgent when listed by the other conversation members. Finally, a *reply* parameter can be set to denote the message as a reply to a specific message in the conversation. In addition to setting

parameters, the user can attach files of information to a message, (e.g. programs, data, text-processing sources, etc.). Other members of the conversation then detach the files for further processing and execution.

Note that the degenerate case of conversation-based mail is the same as memo-based mail. For example, a user can start up a new conversation by simply sending an initial message. If replies are also sent in separate conversations, a set of independent conversations results. Such a set of conversations is equivalent to the set of independent memos generated by current memo-based user interfaces.

3.4. Access Permissions

So far, the discussion has assumed that all participants in a conversation have the authority to perform all of the operations. This cannot practically be allowed. Instead, a set of permission flags are associated with each conversation, which grant **read**, **write**, and **administrative** authority for that conversation.

All participants in a conversation are given read permission. If the write permission is also enabled, then the participant is considered active. If writing is disabled, the participant is passive. Having administrative permission implies the ability to add and delete participants, merge and split conversations, and supersede messages. The owner of a conversation, (i.e. the user that creates it), is initially given all access rights, as well as the right to extend permissions to other participants.

4. Conversation Model

In this section we develop a model for the fundamental object of conversation-based mail — the conversation. Specifically, we present a notation for the components of a conversation: participants and messages. In addition, we describe the underlying structure of a conversation.

The critical difference between the structure of conversation-based mail and current memo-based systems is that rather than ordering messages according to their arrival at the recipient's mail-box, messages are recorded based on the context in which they were written.

4.1. Basic Terminology

A conversation consists of a group of messages, denoted $M = \{m_i \mid i \geq 0\}$, shared by a set of participants, denoted $P = \{p_i \mid i \geq 0\}$. Each participant is able to view all messages associated with the conversation, and may add new messages. Thus, we take the view that messages are added to a specific conversation, not mailed to a group of recipients.

A conversation begins when a user defines the set of participants P and submits an initial message m_0. New members may be added to a conversation by having that list expanded to include them. Similarly, old members may be removed. Being added to a conversation means having access to the entire history of the conversation, (i.e. all of set M). Removal implies not being able to read any future messages submitted to the conversation. A conversation ends when all of the participants have been removed. In addition, participants are notified, and given the option of closing a conversation, when it has been inactive for a period of time.

Each participant in a conversation is known by a *universal-name* which is understood by both the underlying mechanisms that support conversation-based mail, and by the humans who communicate with it. In addition to universal names, a private set of aliases is maintained for each user. These aliases are used by a participant to specify other members of a conversation, and by the user interface when presenting participant names to that user.

4.2. Context Graph

The underlying structure of conversation-based mail maintains the relationship among the messages which make up a particular conversation. Informally, when a participant composes a message, he does so in the context of the messages he has already seen. Specifically, *message context* is a relation $\mathbf{R}$ that holds between messages i and j, such that $m_i \mathbf{R} m_j$ if and only if m_i had been read by the author of m_j before composing m_j. The set of such relations for all the messages in a conversation can be represented by a directed acyclic graph called a *context graph*, and denoted $G = (M, E)$. For simplicity, we let M denote both the vertices of G and the set of messages, and we use the terms "node" and "message" interchangeably. The edges of the graph represent the message context relation. An edge leading from node i to node j implies $m_i \mathbf{R} m_j$, and reads "m_i **precedes** m_j".

Figure 4.1 gives an example of a context graph for a conversation in which message a was the initial message of the conversation. Messages b and c were submitted after their respective authors had read message a, but independent of each other. Finally, message d was submitted after messages a and c had been viewed, but before message b had been read.

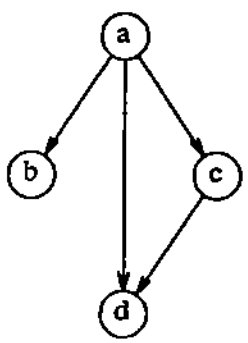

Figure 4.1
Context Graph with Four Messages. In this example, messages a and c precede message d.

Associated with each participant p_i is a subset of M, denoted $\bar{M}_i$, corresponding to those messages in M already viewed by p_i. We can informally think of the context graph as having

each node marked according to whether or not each participant has read it.

When participant p_i submits a message to a conversation, the vertex for that message is connected by edges to those nodes of G which represent messages p_i has read. Viewing a message simply causes it to be added to $\bar{M}_i$. We now formally define the operations as performed by participant p_i.

$$submit(m_{new}) \equiv G' = (M', E'), \text{ where}$$

$$M' = M \bigcup \{m_{new}\}$$
$$E' = E \bigcup \{ (m_j, m_{new}) \mid m_j \in \bar{M}_i \}$$

$$view(m_j) \equiv \bar{M}_i' = \bar{M}_i \bigcup \{m_j\}, \text{ where } m_j \in M$$

4.3. Ordering the Context Graph

Finally, we address the problem of listing component messages of a conversation. In conventional mail systems, messages are presented to the user ordered by their arrival time at the recipient's mail-box. In contrast, conversation-based mail systems use a context graph which gives information concerning the relationship among messages at the time the messages were composed.

One scheme for presenting the messages in a context graph is to display them in a linear order. To do this, we must produce a total ordering of M, given the partial ordering defined by the context graph G. The total ordering of M should preserve the critical information contained in the graph. That is, if message m_j was composed by a participant after reading message m_i, then m_i should be displayed to those viewing the conversation before m_j. It is, however, impossible to guarantee that all the messages seen before message m_j had also been viewed by the sender before composing the message.

The standard topological sort algorithm [KNUT73], produces one possible total ordering of M. Such a total ordering can be improved upon, however, by paying special attention to how "ties" are broken. For example, the messages in the context graph given in Figure 4.1 could be assigned the total ordering $[a, b, c, d]$, $[a, c, b, d]$, or $[a, c, d, b]$. The first ordering is preferred because d more naturally follows c than b.

Another scheme for displaying G involves using indentation to represent each message's *level* within the graph. For example, the graph in Figure 4.1 could be presented as

```
message a
        message b
        message c
                message d
```

In addition, a message that is defined to be a *reply* to another message should be displayed adjacent to that message. To accomplish this, the edge which connects two such messages is specially marked. Marked edges can be used both to insure that related messages are displayed together, and to allow a user to page through a sequence of messages and their replies.

5. Conversations in a Distributed Environment

As discussed in the previous section, a conversation is abstractly viewed as having a single context graph G. Participants may read the component messages and add new ones. In reality, a single context graph is not practical if the participants in a conversation are distributed over more than one computer system. In this case, we view each participant p_i as having a resident copy of a subgraph of G, denoted G_i.

When a user views the messages in a conversation, he sees only those messages contained in G_i, where $M_i \subseteq M$ is the set of messages resident at p_i's system, and $\tilde{M}_i \subseteq M_i$. Conceptually, when a participant submits a message to a conversation, that message is added to G_i. The update is then propagated to all the other participant's copies in order to effect the change to G

described in the last section. We formally state this with the following high-level algorithm, presented relative to participant p_i.

$$\begin{aligned} submit(m_{new}) \quad \equiv \quad & ADD(m_{new}, G_i) \\ & \textbf{for each } j \textbf{ such that } p_j \in P \textbf{ and } i \neq j \\ & \quad UNION(G_i, G_j) \end{aligned}$$

Algorithm 5.1

5.1 Efficient Implementation of Submit

We now discuss the implementation of the primitives used in the above algorithm. The primitive operation *ADD* is similar to the submit function defined in section four; the new node m_{new} is connected to G_i by a set of edges from all the nodes in $\bar{M}_i$. Let $\hat{E}_{new} = \{(m_i, m_{new}) \mid m_i \in \bar{M}_i\}$. We can now define

$$ADD(m_{new}, G_i) \quad \equiv \quad G_i' = (M_i', E_i') \text{ where}$$

$$M_i' = M_i \bigcup \{m_{new}\}$$
$$E_i' = E_i \bigcup \hat{E}_{new}$$

As far as implementing the *UNION* primitive is concerned, we must realize that combining two graphs which physically reside at different locations involves transporting information over computer networks. Since it is inefficient to distribute the entire subgraph G_i so it can be unioned with the other subgraphs of G, we develop an alternative approach which transports the minimal amount of information necessary to update each G_j.

Consider what information must be distributed to each G_j. Initially, the new node, m_{new}, and the set of edges $\hat{E}_{new}$ which connect it to G, must be added to each G_j. Furthermore, because computer networks can deliver messages out of order, it is possible for p_i to receive a message, read it, and respond to it before p_j receives the original message. Thus, we must

insure that all the nodes m_{new} is to connect to are also present in G_j. That is, each node m_k such that $(m_k, m_{new}) \in \hat{E}_{new}$ and its corresponding set of edges, $E_k = \{(m_i, m_k) \mid (m_i, m_k) \in E\}$ must first be incorporated into G_j. Finally, incorporating each m_k may in turn involve the same problem.

Therefore, we formally define the *UNION* primitive to be

$$UNION(G_i, G_j) \equiv G_j' = G_j \bigcup G_{new}$$

where G_{new} for a given j is the subgraph of G consisting of the set of nodes $M_{new} = \{m_i \mid m_i \in \tilde{M}_i \text{ and } m_i \notin M_j\} \bigcup m_{new}$, and the set of edges $\bigcup E_k$ for all k such that $m_k \in M_{new}$. That is, we have defined G_{new} such that $G_j \bigcup G_i \equiv G_j \bigcup G_{new}$ and an efficient implementation of *UNION* results.

As mentioned earlier, the participants in a conversation are identified by a name. Associated with each name is an *address* where the subgraph of G can be located for the named user. A message transport system similar to the one described in the first section is responsible for transporting G_{new} to each user's copy of G. The *MTS* is able to forward updates to the appropriate location by translating participant names into addresses. The exact way in which this translation, and the subsequent forwarding is implemented, is beyond the scope of this paper.

5.2 Removing Redundant Edges

As the number of messages in a conversation becomes large, and therefore $\tilde{M}_i$ grows large for each p_i, a significant amount of overhead is incurred when a new message is added, because $\hat{E}_{new}$ includes edges from every node that is in $\tilde{M}_i$ to m_{new}. We now improve our implementation of *submit* by redefining $\hat{E}_{new}$.

It is sufficient to have a path from m_i to m_j for $m_i \mathbf{R} m_j$, since if a path exists, then all edges can be regenerated if needed. Let $\tilde{G}_i$ be the subgraph of G_i made up of nodes from $\tilde{M}_i$. We can now define

$$\hat{E}_{new} = \{(m_i, m_{new}) \mid m_i \text{ is a leaf node of } \tilde{G}_i\},$$

where a leaf node is one that has an outdegree of 0.

A *reduced* context graph is one that contains no redundant edges. An edge from node m_x to node m_y is redundant if and only if there is also a path of length > 1 from m_x to m_y. Note that this definition means that more than one path may exist between any two nodes. In other words, a reduced graph contains all of the vertices, and as few edges as possible from the original graph, such that the transitive closure of the reduced graph and the original graph are equal. The reduced graph is also known as the *transitive reduction* of the original graph [AHO72]. We demonstrate later that our definition of $\hat{E}_{new}$ produces a reduced context graph. First, we make the following claim.

Claim: Removing redundant edges does not adversely affect the use of context graphs because redundant edges are ignored during the displaying of the messages in a conversation.

Support: As outlined in the previous section, when producing a total ordering of M, given the partial order defined by G, we want to insure that m_j is displayed after m_i if $m_i \mathbf{R} m_j$. A simple example will illustrate that this is still the case, even without redundant edges. (The dotted line denotes a path rather than an edge.)

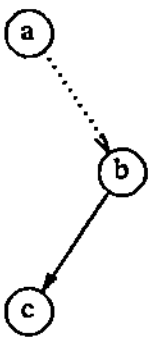

Figure 5.1
Example Context Graph

By the condition given in section 4.3, message a must be displayed before message b, and b before message c. Thus, c follows a without an edge from c to a. □

5.3 Consistent and Reduced Graphs

We now prove two lemmas about our implementation of context graphs in a distributed environment. We demonstrate that Algorithm 5.1 produces context graphs which are both reduced and consistent under the assumption that *ADD* and *UNION* are atomic operations. A set of distributed graphs is consistent if the component graphs are equivalent after the same set of messages have been incorporated into each. By definition, if a conversation is not distributed, and therefore one copy of the context graph exists, this proposition is true. Consider a situation where participant p_i adds new nodes to subgraph G_i, while G_i is subjected to possible updates caused by other participants adding messages to their subgraphs, and propagating the changes to G_i.

Lemma 5.1: Algorithm 5.1 results in a context graph G_i with no redundant edges.

Proof: Consider G_i into which message m_{new} is being added. Let m_t be a leaf node of G_i. The edge (m_t, m_{new}) is a redundant edge **iff** there is another subgraph of G, denoted G_j, such that

there is an edge to m_{new} from some node m_k in G_j, and a path from m_i to m_k. Assume, as illustrated in Figure 5.2, that G_i and G_j overlap such that a path exists from m_i to m_k. Then, m_i is not a leaf node of G_i and our premise is violated. □

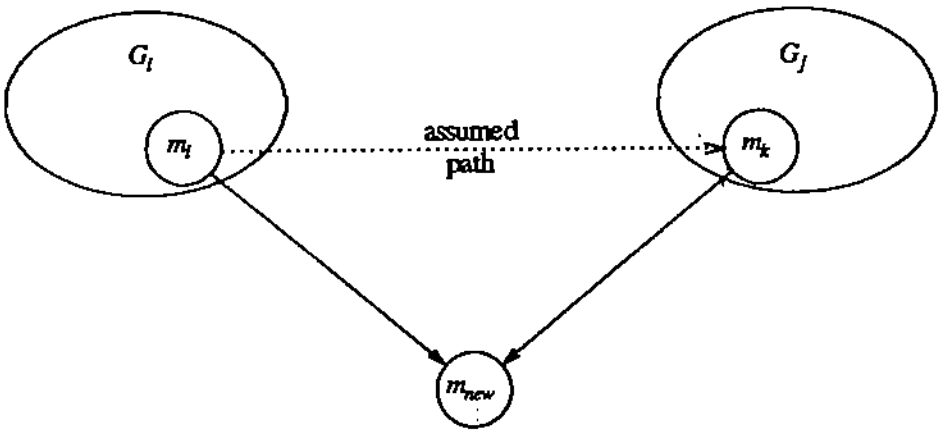

Figure 5.2
When adding m_{new} to G, assumption of a path
from m_i to m_k leads to a contradiction.

Secondly, we prove that G remains consistent for Algorithm 5.1. That is, all participant's resident subgraphs are equal.

Lemma 5.2: After a finite number of messages have been successfully submitted to a conversation, $G_i = G_j$ for all i and j such that $p_i, p_j \in P$.

Proof: If we think of the *ADD* primitive as computing the union of G_i and G_i', then we can say that G_i' is unioned with all the subgraphs of G. Since both operations are atomic and cannot interfere with each other, and because unions are commutative, the same set of subgraphs must result. □

Based on our claim concerning the unimportance of redundant edges, and the two lemmas just presented, we have demonstrated the equivalence of conversation-based mail in a distributed environment and in a single system environment.

6. DRAGON-MAIL: A Prototype

This section describes a prototype conversation-based mail system, called *DRAGON-MAIL*, which is being designed as part of the TILDE project at Purdue. The rest of this section will discuss the basic components of the prototype. We first describe the underlying structures which support the operations described in the third section. We then explain how *DRAGON-MAIL* transports messages through a distributed environment. Finally, we give an overview of how conversations and messages are presented to the user.

6.1. Underlying Structure

The data necessary to support conversation-based mail is organized into two directories. One directory contains information on a per-user basis, and the other on a per-conversation basis. For each user, there is a file in the per-user directory which contains information about all the conversations that user is involved in. This information includes the unique name of each conversation so that additional information can be accessed in the per-conversation directory.

The per-conversation directory contains a separate sub-directory for each conversation on that host. Each sub-directory, in turn, contains a set of files describing the state of that conversation, (e.g. the context graph, a list of participants, etc.), along with a file corresponding to each message that has been submitted to the conversation.

6.2. Message Transport

DRAGON-MAIL transports information between distributed copies of a conversation's context graph via existing mail transport mechanisms. Specifically, a message is given additional header lines, such as **Conversation-Id, Participants**, etc., and then deposited with sendmail [ALLM83] for delivery. The To field is defined as "dragon@host" for each host that has participants in the conversation. The receiving host has an alias for "dragon" which cause

sendmail to deliver the message to the local *DRAGON-MAIL*.

Such a scheme can be classified as *batch*, in that two distinct instances of *DRAGON-MAIL* never communicate directly, but rather depend on a separate mechanism to carry messages between them. Consequently, the subgraph G_{new}, as defined in section 5, cannot be determined. Thus, it is possible that a message may arrive that cannot be incorporated it into the local copy of the context graph because it depends on another message that has not yet arrived. Rather than have the local *DRAGON-MAIL* "go looking" for the missing message by sending requests through sendmail to various remote locations, we choose to queue the message, and try to incorporate it into the context graph at a future time. A timeout mechanism is necessary to insure that the message is eventually dealt with. Note that an *interactive* protocol between peer *DRAGON-MAILs* would allow for an immediate exchange of all necessary messages and control information. That is, the exact subgraph G_{new} could be exchanged between remote systems.

6.3. Presentation Level

We now describe the presentation-level features of *DRAGON-MAIL*; a pictorial view can be found in the appendix. Currently, this portion of *DRAGON-MAIL* is implemented on non-intelligent terminals using the curses [ARNO79] screen management package. Future plans include designing a system that will operate on work-stations connected to the local computing engine [COME84].

When a user initiates a *DRAGON-MAIL* session, he is presented with a window containing a list of conversations currently in the foreground. The list is sorted according to the relative importance of each conversation. Those conversations which contain new messages are listed before those that do not, and conversations which contain urgent messages are highlighted. (See Figure A.1.) Thus, *DRAGON-MAIL* presents an organized plan for the user to follow in processing his mail. The user then selects a specific conversation for further consideration. In addition, a

query of the conversation database produces a window of conversations which may be manipulated in the same manner as the foreground window.

Once a specific conversation is selected, a window containing a list of messages in that conversation appears. The list contains those messages which have not yet been read, along with those messages in the surrounding context. The unread messages are highlighted and urgent messages have additional markings. (See Figure A.2.) The user can also display other portions of the conversation.

Finally, when a message is selected for viewing, it is displayed in a message window. Only the message author, subject, and composition date, are displayed in addition to the message body. (See Figure A.3.)

7. Summary

We have presented an overview of a user interface for computer mail based on conversations rather than independent memos. Conversations have the advantage of being more consistent with the way humans communicate. They also provide an overall structure to mail that allows all forms of computer-based communication to be presented in one uniform environment.

We also developed a model for conversations which preserves message context, and demonstrated how such a model assists the user in managing and comprehending his mail. We then outlined how conversation-based mail is supported in a distributed environment. Finally, we presented an overview of a prototype conversation-based mail system called *DRAGON-MAIL*.

REFERENCES

AHO72 Alfred Aho, Michael Garey, Jeffery Ullman, "The Transitive Reduction of a Directed Graph", *SIAM J. Computing*, 1:2, pp. 131-137, 1972.

ALLM83 Eric Allman, *"SENDMAIL—An Internetwork Mail Router"*, *UNIX Programmer's Manual*, 4.2 Berkeley Software Distribution, Vol. 2, August, 1983.

ARNO79 Kenneth Arnold, "Screen Updating and Cursor Movement Optimization: A Library Package", *UNIX Programmer's Manual*, 4.1 Berkeley Software Distribution, Vol. 2, 1979.

BIRR82 Andrew D. Birrell, Roy Levin, Roger M. Needham, Michael D. Schroeder, "Grapevine: An Exercise in Distributed Computing", *Communications of the ACM*, Vol. 25, No. 4, April, 1982, pp. 260-273.

BORD79 Bruce S. Borden, R. Stockton Gaines, Norman Z. Shapiro, *The MH Message Handling System: User's Manual*, R-2367-AF, Rand Corp., November, 1979.

BROT81 Douglas K. Brotz, *Laurel Manual*, XEROX, CSL-81-6, Palo Alto, California, May, 1981.

COME84 Douglas Comer, *Transparent Integrated Local and Distant Environment (TILDE): Project Overview*, Purdue University, Tilde Report, CSD-TR-466, March, 1984.

CROC77 David H. Crocker, *Framework and Functions of the "MS" Personal Message System*, R-2134-ARPA, Rand Corp., December, 1977.

CROC79 David H. Crocker, Edward S. Szurkowski, David J. Farber, "An Internetwork Memo Distribution Capability – MMDF", *Proceedings of the Sixth Data Communication Symposium*, November, 1979, pp. 18-25.

DENN81 Peter Denning, Douglas Comer, *The CSNET User Environment*, Computer Science Department Purdue University, July 1981, unpublished note.

HEND79 D. Austin Henderson, Theodore H. Myers, "Issues in Message Technology", *Proceedings of the Fifth Data Communication Symposium*, November, 1978, pp. 6.1-6.9.

KNUT73 Donald Knuth, *The Art of Computer Programming*, Vol. 3, Addison-Wesley, Reading Mass., 1973.

LEVI79 Roy Levin, Michael Schroeder, *Transport of Electronic Messages Through a Network*, XEROX, April, 1979.

LIPI74 H. M. Lipinski, R. H. Miller, "FORUM: A Computer-Assisted Communications Medium," *Proceedings of the 2nd International Conference on Computer Communications*, pp. 143-147, August 1974.

MYER76 T. H. Myer, C. O. Mooers, *Hermes User's Guide*, Bolt Beranek and Newman, Cambridge, Mass., 1976.

PICK79 John R. Pickens, "Functional Distributed Computer Based Messaging Systems", *Proceedings of the Sixth Data Communication Symposium*, November, 1979, pp. 8-17.

POST79 Jonathan B. Postel "An Internetwork Message Structure" *Proceedings of the Sixth Data Communication Symposium* November, 1979, pp. 1-7.

POST82 Jonathan B. Postel "Simple Mail Transfer Protocol", *RFC 821*, August, 1982.

SCHK81 Peter Schicker, "The Computer Based Mail Environment—An Overview", *Computer Networks*, Vol 5, 1981, pp. 435-443.

SCHM79 Eric Schmidt, "An Introduction to the Berkeley Network", *UNIX Programmer's Manual*, 4.1 Berkeley Software Distribution, Vol. 2, 1979.

SHOC78 John F. Shoch, "Inter—Network Naming, Addressing, and Routing", *COMPCON*, 6Fall 1978, pp. 72-79.

SHOE79 Kurt Shoens, "Mail Reference Manual", *UNIX Programmer's Manual*, 4.1 Berkeley Software Distribution, Vol. 2, 1979.

APPENDIX

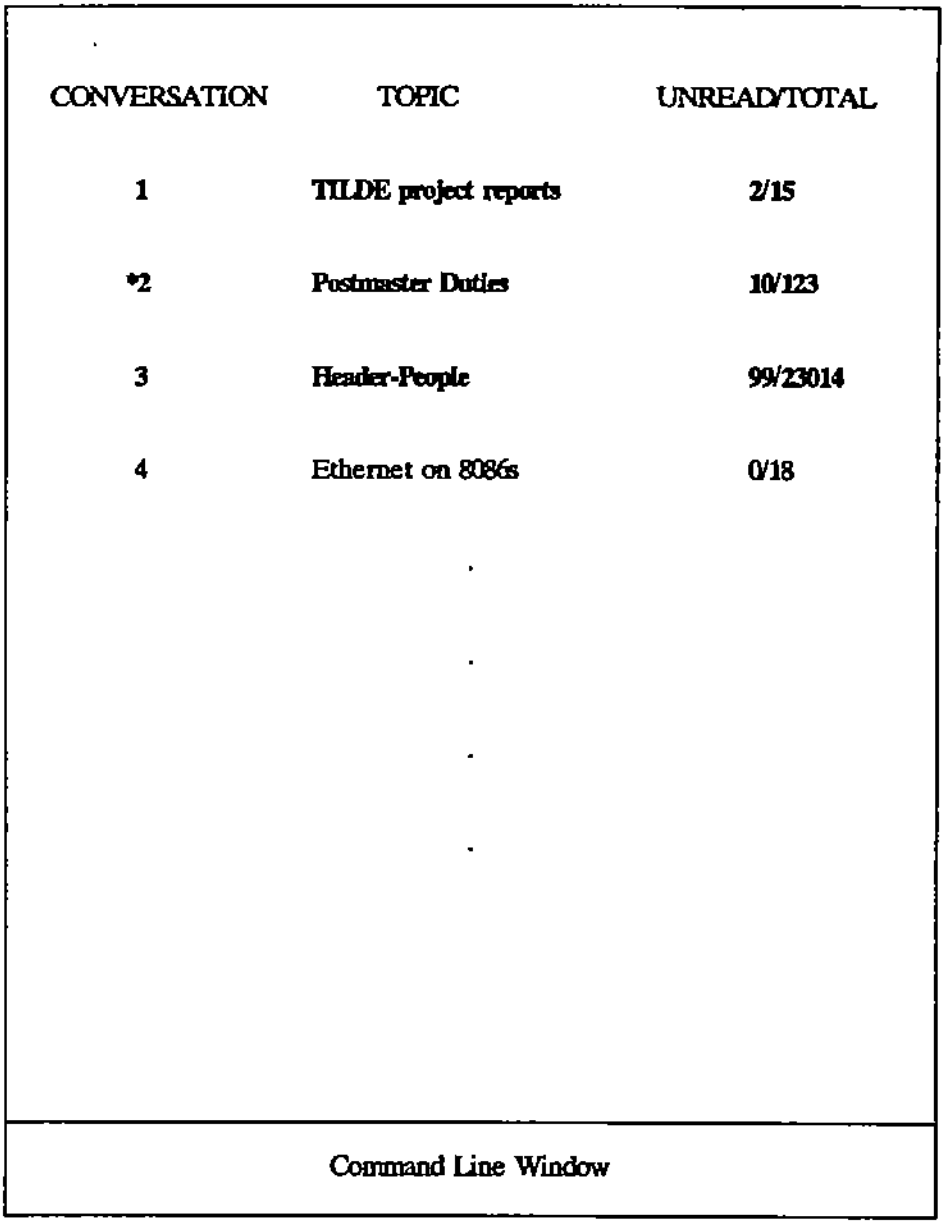

Figure A.1

Screen displayed when a mail session is initiated. Contains a brief synopsis of the conversations in the foreground. Lines are highlighted, by **boldface** in this document, to indicate that the conversation contains an unread message. Conversations marked with an asterisk contain urgent messages. The user may execute one of the operations described in section 3.1, including selecting a conversation for further viewing.

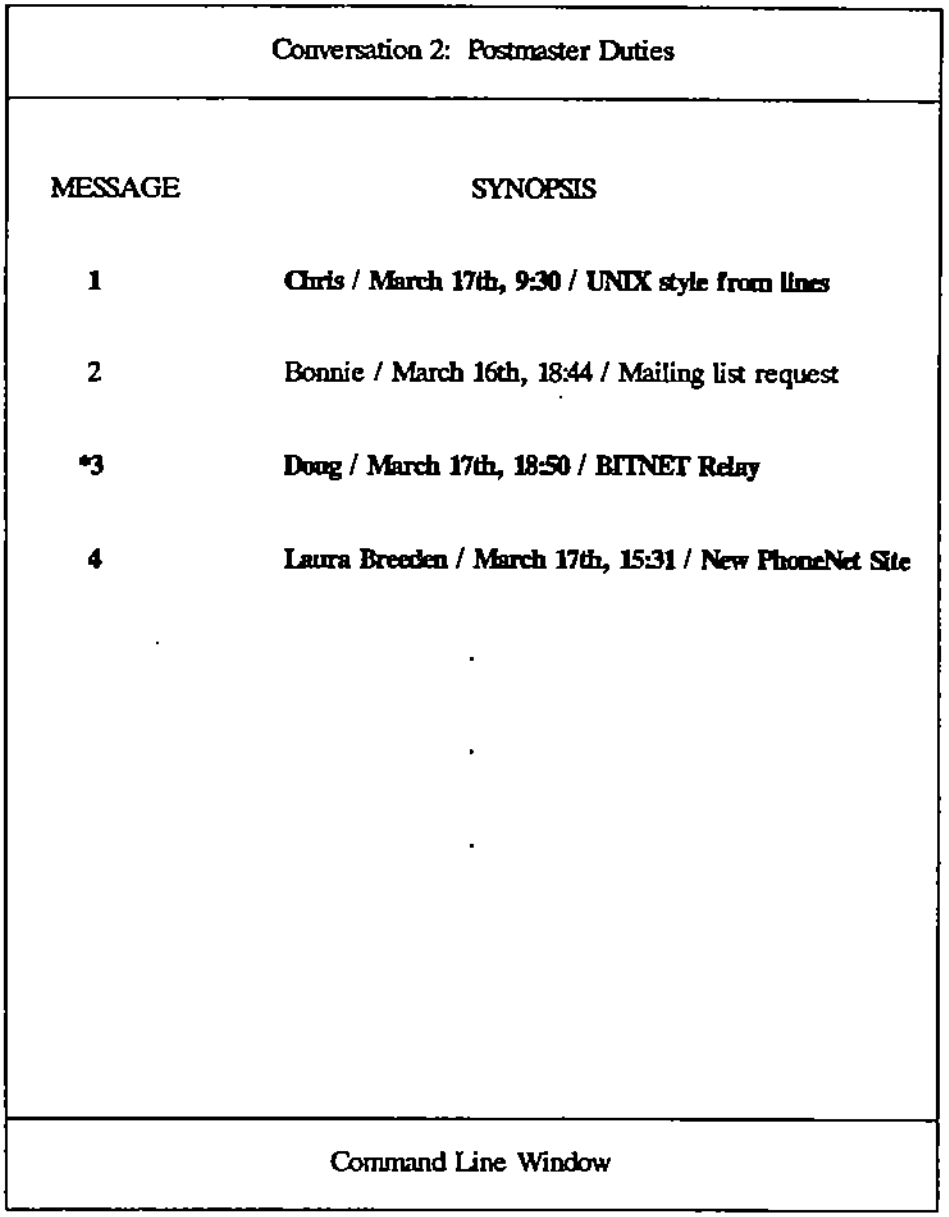

Figure A.2

Screen displayed when a specific conversation is selected for detailed viewing. Contains a brief synopsis of messages in the conversation. Unread messages are highlighted, and urgent ones are marked with an asterisk. Previously read messages (i.e. non-highlighted) appear in this display to give the surrounding context of new messages. The operations defined in section 3.2 can be executed at this stage.

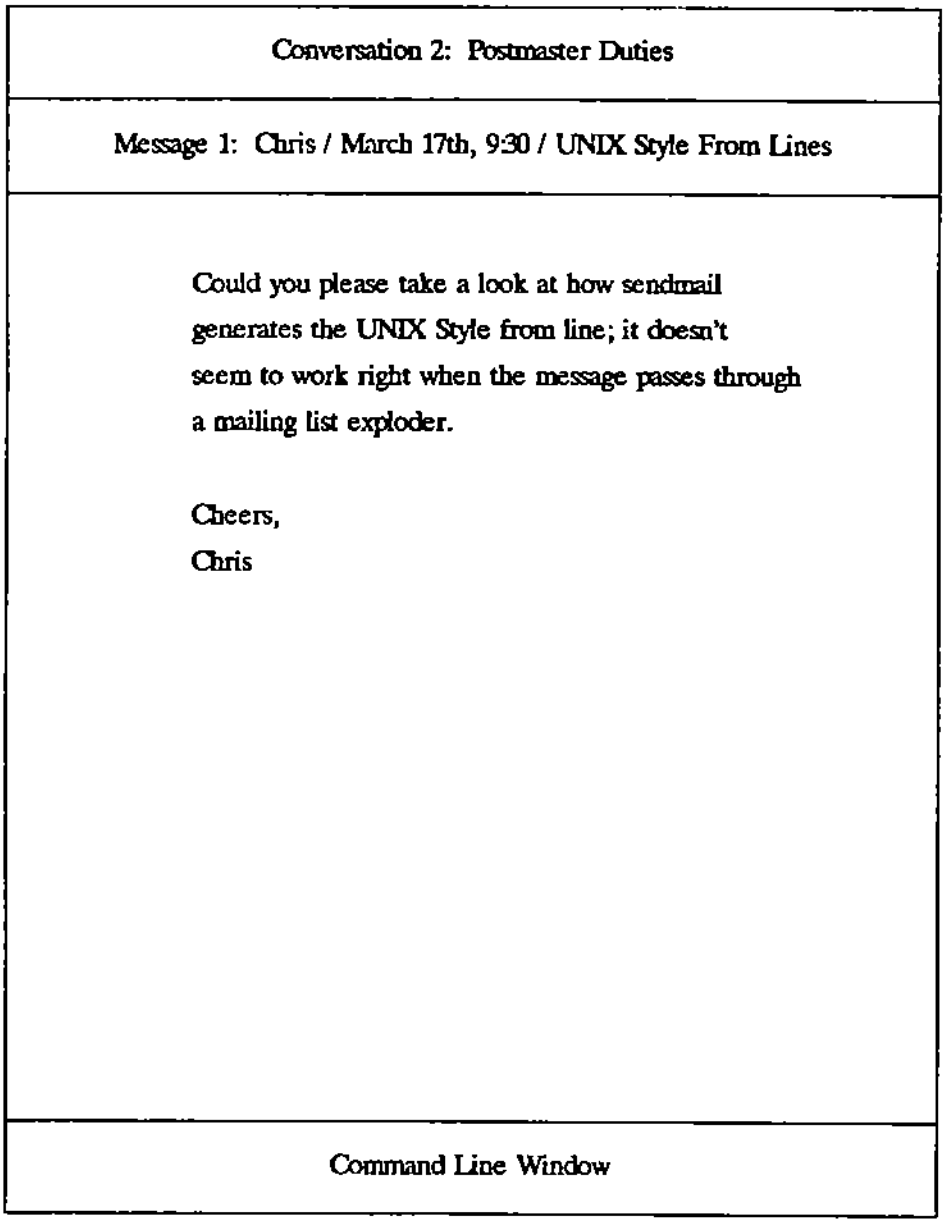

Figure A.3

Screen displayed when a specific message is selected. The operations defined in section 3.3 can be executed at this stage.