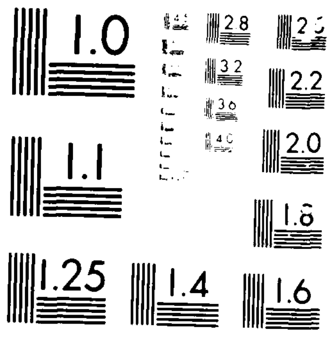
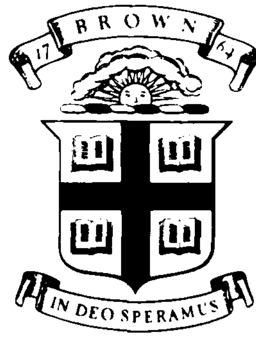
AD-A168 668 MAILDRIVER: A DISTRIBUTED CAMPUS-WIDE MAIL SYSTEM(U) 1/1
BROWN UNIV PROVIDENCE RI DEPT OF COMPUTER SCIENCE
R YAMPELL JAN 86 CS-86-07

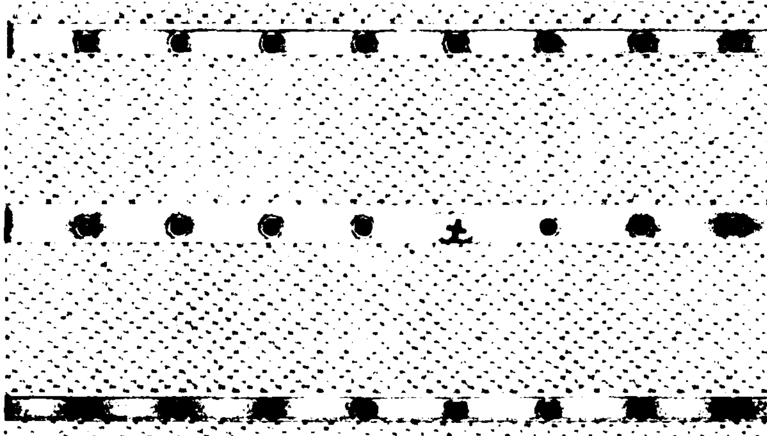
UNCLASSIFIED F/G 9/2 NL



AD-A168 668



BROWN UNIVERSITY



Department

of

Computer Science

DTIC FILE COPY

DTIC
ELECTE
JUN 9 1986
S D
A

86 5 044

Maildriver: A Distributed Campus-wide Mail System

Rich Yampell

Brown University
Dept. of Computer Science
Providence, Rhode Island 02912

Technical Report No. CS-86-07
January, 1986

No DoD funds for this report.
Per Ms. Tremblay, Brown University,
Dept. of Computer Science

This document has been approved for public release and sale; its distribution is unlimited.

Maildriver: A Distributed Campus-wide Mail System

by Rich Yampell

ABSTRACT

Maildriver is a distributed system for a future campus-wide computer mail service. The goal of this paper is to set forth the design aspects of **Maildriver** insofar as they relate to a distributed system. Thus, there are aspects of **Maildriver** which are not covered herein, but which are presumably part of separate research (such as the mapping of user names, user authentification, details of networking, etc.). This paper is divided into four parts: Part I discusses **Maildriver** from a user's point of view. Part II is concerned with implementation details of the various **Maildriver** functions. Part III deals with robustness and crash recovery. Part IV concludes. In addition, there is an Appendix which contains a pseudo-code listing, based on the C language, of some of the critical **Maildriver** processes (specifically, the master processes).

January 18, 1986

Maildriver: A Distributed Campus-wide Mail System

by Rich Yampell

Part I. A User's View of Maildriver

1. Introduction

Maildriver is a campus wide computer mail system. It allows any member of the campus community with any kind of computer access (i.e. a personal computer) to exchange computer mail with other such people in the campus community. **Maildriver** users connect to any one of a set of central, dedicated **Maildriver** machines to either send or receive messages. Because **Maildriver** is a distributed system, all services are available to the user, regardless of the **Maildriver** machine to which he is connected.

Maildriver is an acronym for **M** -aildriver **A** -bsolutely **I** -nsures **L** -etter **D** -elivery. **R** -ich **I** -s **V** -ery **E** -xcited. **R** -eally.

1.1. Environment of Maildriver

Maildriver is assumed to run on a set of dedicated **Maildriver** machines, all of which are networked such that any given machine may communicate directly with any other. Rough initial guestimating suggests three such machines as a starting point, although the **Maildriver** design allows for the addition of more. In any case, no hard assumptions are made concerning the number of **Maildriver** machines.

Maildriver also assumes facilities for users to connect to **Maildriver** machines, either through phone lines or directly. Client computers which communicate with **Maildriver** computers are presumed to have front-end programs which allow them to communicate with **Maildriver.** Such front-ends would, of course, have to be tailored for their target machines. Thus, if user John Doe uses his MacIntosh to connect to **Maildriver,** he would have a front end program tailored for the MacIntosh which knows how to communicate with **Maildriver.** The design of such client front-end programs seems straight-forward, and is outside the scope of this paper.

1.2. Clients of Maildriver

To be a client of the **Maildriver** system, one must fulfill three criterion. First, one must have access to a computer which can physically access **Maildriver.** Second, one must have a front-end program on that computer which can communicate with **Maildriver.** Third, one must be a registered **Maildriver** user. It is assumed that there will be some administrative organization to oversee the

functioning of **Maildriver;** a user would become registered through this organization.

In addition, for certain privileged members of the administrative organization, there are special administrative functions available which may not be run by normal clients.

2. Design Goals

The chief goal in designing **Maildriver** was to create a robust, distributed system in which a user would always have access to the full functionality of the system. In particular, the user should *always* be able to read *all* of his mail. An additional goal was to deal reasonably efficiently with bulk mail. It was recognized that in a University environment, there would likely be a large amount of bulk mailings and junk mail (i.e. notices to all undergraduates, memos to all faculty, etc.). Since the number of recipients of such mail could conceivably be quite large, **Maildriver** tries to make such mailing as efficient as possible.

3. Functionality

The **Maildriver** system makes the following functions available to clients of the system:

send letter -- a letter, already prepared on the client's machine via some type of editor, is deposited with the Maildriver system for delivery.

receive mail -- Maildriver fetches all accumulated mail for the user and sends it over to the client machine.

delete letters -- Maildriver deletes the specified letters from the user's mailbox.

list letters -- fetches a quick list of pending mail for the user. This is intended to roughly implement the Unix (tm) "from" command.

message-polling -- a quick check is made to see if the user has any pending mail.

The "receive mail" function sends all letters for the user to his client machine. That is, it finds any mail, anywhere in the system for the user. When the mail is sent over, it is not deleted. Mail only goes away when specifically requested to via the "delete letters" function. Thus, the typical sequence would be for the client front-end to request the user's mail, and then request deletion of all letters the user does not want preserved.

The "message-polling" function is intended to be used by client front-end programs running in the background to periodically check if the user has mail.

Since it will necessarily be run frequently and by many users, it is designed to run as quickly as possible. Thus, only a "quick" check is made for mail; that is, the **Maildriver** machine to which the user is connected only checks its own storage to see if there is mail, rather than polling the entire **Maildriver** system. While this is only a heuristic, it will be accurate most of the time. This "quick" checking approach also applies to the "list letters" function.

In general, **Maildriver** functions send back an acknowledgement when the task is completed. If a client receives an acknowledgement, it may proceed with the certain knowledge that the task has been/will be fulfilled. If no acknowledgement is forthcoming, something may be awry with that particular **Maildriver** machine (i.e. it has crashed), and the client would then simply connect to another **Maildriver** machine and repeat the request. Note that no damage will be done by a partially completed request.

In addition to these general functions, the following administrative functions are available to qualified administrators:

add user -- a new user is added to the system

delete user -- an old user is removed from the system

change master -- forces a specified machine to become the "Master". More on this later.

4. A Sample Session

Let us consider a sample session of user interaction with **Maildriver.** Take the case of our good friend John Doe and his trusty Brand X personal computer.

Johnny wants to write a letter to his girlfriend Pauline, who he knows is on the **Maildriver** system. He fires up his trusty Brand X (which includes starting up a daemon called XCLOCK, which periodically polls **Maildriver** for mail via the "message-polling" function) and gets into XTEXT, the Brand X text editor. Using XTEXT, he composes his letter to Pauline. While editing the letter, the daemon finds out that he in fact has mail waiting for him and lights up a MAIL light on his screen. John finishes the letter, saves it, and enters XMAIL, the front-end program to **Maildriver** for the Brand X. He directs XMAIL to send the letter to Pauline. XMAIL gets the letter, connects to **Maildriver,** and sends the letter via the "send letter" function. **Maildriver** sends back an acknowledgement that it got the letter, and XMAIL tells John that everything is A-OK. Then John directs XMAIL to get his mail for him. XMAIL sends **Maildriver** the "receive mail" directive and **Maildriver** sends back all the letters waiting for John. XMAIL saves the letters as they come in and presents them to John in a user-friendly way (Brand X computers are notoriously user-friendly). It turns out that John got three letters. John reads them all, and decides that he wants to preserve one of them within **Maildriver** and deal with it later; this he informs XMAIL. To comply with this request, XMAIL sends a "delete letters" request to **Maildriver** for the two letters that John does *not* want preserved. Again, **Maildriver** sends a back an acknowledgement. John is finished, and XMAIL breaks connection.

Note that the specifics of communication protocol are outside the scope of this paper and are not described here.

Part II. Implementation Details (or, "A look under the hood")

5. Overview of System Internals

In order to meet the design goal of providing full service to all users at all times, **Maildriver** is a fully replicated system. This means that all services are available through any **Maildriver** machine, and in particular, user mail is replicated onto each **Maildriver** machine. Thus, when a given letter is deposited into the **Maildriver** system, each **Maildriver** machine gets a copy of that letter. Similarly, when a letter is deleted, it must be deleted off of every machine in the system.

To coordinate all this replication, **Maildriver** designates one of its machines to be the "Master". The Master machine functions just like any of the other machines, except that it runs an additional process, the Master process, which maintains open communications channels with all the other machines.

The job of the Master is two-fold. First, it directs and keeps track of the state of letter replication. Second, it keeps track of which **Maildriver** machines are up and which are down; for each down machine, it keeps a list of what actions that machine will need to perform when it comes back up in order to catch up to the rest of the world.

In addition to the Master, one machine is designated the Lieutenant. The job of the Lieutenant is simply to keep a duplicate copy of the Master's information; in the event that the Master crashes, the Lieutenant takes over and becomes Master.

The initial appointments of Master and Lieutenant are specified in startup files which are read when the system first comes up.

Each machine maintains a set of special directories. The most important of these is the WorkSpace directory. This directory is used to process the delivery of mail. In addition to the WorkSpace directory, each machine has a Master directory, a Cemetery directory, and a Lieutenant directory. Only the current Master and the current Lieutenant, respectively, actually use these directories (but since any machine may at some point attain either of these ranks, all machines have these directories set up in advance).

Aside from the special directories, each machine maintains a directory for each **Maildriver** user. These directories are known as the user InBoxes. Each file in an InBox directory is a discrete letter.

Every machine has running on it a process known as the Server process. The job of the Server process is to wait for a user to make connection, and then fork off a copy of itself to process the user's requests. The Server process is then,

essentially, the "front-end" of **Maildriver.** It is the only part of the system that clients actually see.

Each machine also has a back-end process which actually handles message delivery, called the Delivery process. The Delivery process takes its orders from the Master, which generally consist of things like sending and deleting letters.

Finally, each machine runs what is called a Pulse process. The sole job of the Pulse process is to let the Master know that that machine is still up and running. The Pulse process normally sits, dormant, until a polling message comes in from the Master, to which sends a simple response indicating life. Thus, the idea is that the Master takes the "pulse" of a given machine.

6. The Structure of the Master

The job of the Master is complex and multifaceted. In order to keep things organized and running smoothly, the Master process forks off several auxiliary processes to perform specific tasks. Thus, the Master machine actually runs a *set* of master processes; of these, the parent and chief is the Master process.

The most important of these is called the Caretaker. The Caretaker is responsible for issues of life and death. It regularly polls the Pulse processes of the sundry machines to verify that they are still alive. In this manner, it can determine when a machine has died and can take appropriate action. This will be discussed in detail in section 10. (of course, it is useless for the Caretaker to check the Pulse of its own machine; hence, the Lieutenant is responsible for polling the Master's Pulse process).

The other processes created by the Master process are called Sender processes. One Sender process is created for each **Maildriver** machine in the system. The job of the Sender is straight-forward: it accepts messages for its machine, queues them up, and sends them over when appropriate. In this way, both the Master and Caretaker processes can send messages to the sundry machines in an orderly fashion.

The setup, then, is as follows. Both the Master and the Caretaker can send messages to any machine via the Sender processes. The Sender processes are connected to the Delivery and Server processes of their respective machines. The Master and Caretaker can also communicate directly with each other. In addition, the Caretaker has separate network connections to the Pulse processes of each machine. Finally, the Master has a separate connection to the Lieutenant process on the Lieutenant machine (and the Lieutenant has another separate connection to the Master's Pulse).

Since all communication out from the Master goes out through the Sender processes, they will not be discussed further; that is, we will in general refer to the Master sending a message to a machine with the implicit understanding that in fact the Sender acts as intermediary.

7. Message Acceptance: beneath the "send letter" function

7.1. The Job of the Server Process

When a user wishes to send a letter via the **Maildriver** system, he connects to a **Maildriver** machine (which is to say, he connects with the Server process for that machine; the Server process then forks off a copy of itself to service the user) and deposits his letter via the "send letter" function.

When the Server process gets the "send letter" request, it sets about creating a thing known as a "Protoletter". First, it creates a unique filename of the form <local machine name>.<time stamp, given that machine's view of time>. Thus, a filename might look something like MD1.4039382. It then creates a file by this name in the WorkSpace directory and writes a one page [that is, disk page] header containing the list of recipients of the letter [note: mail aliases are expanded at this time through a process outside the scope of this paper] and a State flag, which in general may be set to either "valid" or "bogus"; it sets it now to "bogus". It then reads the body of the letter from the client and appends it onto the Protoletter. [Note that the header is a full disk page long, regardless of how much of it is actually used. This is because writing one disk page is considered an atomic operation]

Once it has the Protoletter set up, it notifies the Master and sends it the Protoletter. The Master takes the Protoletter and puts it in its Master directory (using the original unique filename, of course). When it has it, it sends the filename to the Lieutenant (not the file, just the filename). When the Lieutenant acknowledges, it rewrites the header, changing the State field to "valid", and sends acknowledgement back to the original machine [note: the original machine may well have been the Master machine itself. Remember that the Master machine functions just the same as any other machine, except that it has the Master processes on it]. When the original machine gets acknowledgement from the Master, it in turn sends out acknowledgement to its client. Thus, to insure letter delivery, what is really necessary is to get a valid copy out to the Master. Note that recipient parsing is done on the original machine to avoid overloading the Master machine.

Now the client can go his merry way, and the forked Server process is free to die (if the client has no further requests). The Master picks up the ball from here.

7.2. The Job of the Master

The Master now sends a copy of the Protoletter to each **Maildriver** machine's Delivery process (including its own, of course). Each Delivery process then copies the Protoletter into the WorkSpace for its machine and proceeds to process the delivery of that letter. Actually, there is one exception to this procedure: the Master does not send the whole Protoletter to the machine which originally received the letter (since that machine already has a copy of it); it just sends it a special message which in human terms translates out to "you already have the letter, just mark it 'valid' and process it" [note that the Master doesn't even need to remember which machine originally received the letter, since the machine name is part of the filename].

Each Delivery process receives the Protoletter from the Master and sends back an acknowledgement. It then processes the Protoletter in the manner described below, and then sends another acknowledgement to the Master [thus, the first acknowledgement means "I got it", and the second means "I've finished it"]. The Master, meanwhile, keeps track of which machines have acknowledged and which have not. When all machines have acknowledged that they're finished, the Master can discard these records, as well as the copy of the Protoletter in its Master directory (remember that by now it has a copy in its WorkSpace).

7.3. The Job of the Delivery Process

The Delivery process must now perform the actual delivery of the letter, using the Protoletter. What it does is read the header of the Protoletter, which contains the recipient list for that letter. Then, for each recipient, it puts a link to the Protoletter into the InBox directory for that user. It then rewrites the header for the Protoletter to no longer contain that user in the recipient list. It proceeds this way until the recipient list is empty, at which point it can delete the Protoletter and send final acknowledgement to the Master.

If at any point in delivery the Delivery process tries to deliver a letter which is already there, it ignores the new one in favor of the old one. More specifically, if the call to "link" fails, and the error number shows the error to be an attempt to create a link where a file already exists, the Delivery process simply continues, pretending there was no error (as opposed to taking some kind of emergency action for an unexpectedly failed syscall). This should virtually never happen, but there may be some very weird extreme circumstances where it might. This rule of thumb works around all such instances.

Note the key idea here is the the Delivery process makes *links*, not copies, of the Protoletter. Thus, there is only one copy of a letter per machine, regardless of the number of recipients of the letter. This is the vital step towards realizing our goal vis-a-vis bulk mail. Since letters never get edited or changed in any way once delivered, there is no reason not to do it with links. Note also that this decision to use links is what leads to the one letter per file mail structure (as opposed to one long file as the user's inbox, a la Unix (tm)).

7.4. The Job of the Lieutenant

During letter delivery, it is important that the Lieutenant keep track of what's going on. Just as the Master machine is just a normal machine with a Master process running on it, so the Lieutenant machine is a normal machine with a Lieutenant process on it. The Lieutenant process sits waiting to hear from the Master (and occasionally polling it to make sure its still alive). When the Master first gets a new letter, it sends the filename to the Lieutenant, who records the filename for future reference. In this way it knows that there is a letter in mid-delivery. When all Delivery processes have acknowledge back to the Master, the Master sends the Lieutenant a special acknowledgement which means, essentially, "its ok, that letter got sent ok, you can forget about it", and the Lieutenant deletes that send request from its records. All this will become relevant in section 11.

8. Message Retrieval: beneath the "receive mail" function

When a user uses the "receive mail" function to get his mail, it is our goal to make sure he gets *all* of his mail, even if some of it is only half delivered. When the client machine calls up a **Maildriver** machine, gets connected to a Server process, and makes a "receive mail" request, the Server process goes through a number of steps to get that client's mail.

First, the Server process sends a message to the Master, a message which in English would translate roughly to "Hey, this guy here wants his mail. If you've got anything for him, send it right over." The Master then scans all of its recently received letters, looking for any which have not yet been replicated to the Server's machine and for which the user in question is, in fact, a recipient. If it finds any such letters, it immediately sends them to the Delivery process in the normal manner. These letters, then, will show up presently as Protoletters on the desired machine; all that really happens is that the user's mail is moved-to-the-front-of-the-line, as it were.

Next, the Server process examines each Protoletter in the WorkSpace. For each Protoletter, it reads the header page and looks to see if its user is on the recipient list. There are actually four possible cases, with corresponding actions taken.

If the user is not in the list, it can ignore this Protoletter and go on to the next one.

If the user is in the list, but the State flag is marked "bogus", then the letter is not done being replicated (it is probably one being sent over by the Master on rush order, as a result of the first step), and what we want to do is wait on it. The Server process puts the name of this Protoletter on a special list called the RTCBACIL list (the "Remember To Come Back And Check It Later" list) and goes on to the next Protoletter.

If the user is the *first* person in the list (and the State is "valid"), it means that there is a Delivery process running right now delivering the letter to the user, so the thing to do is just wait. Again the name is put on the RTCBACIL list and scanning continues.

If the user is on the recipient list (and the State is "valid"), but is *not* the first person, then the Server process locks the Protoletter file, and proceeds to deliver the letter itself, proceeding just as the Delivery process would. Since the file is locked, the Delivery process will have to wait before it can start delivery to a new user (and thus we avoid conflicts of two processes trying to write to the same file-- both the Protoletter file *and* the actual letter file). After the Server process has delivered the letter and rewritten the Protoletter header, it unlocks the file and proceeds to the next Protoletter.

Finally, the Server process must process the RTCBACIL list. To do this, it repeatedly scans the list, taking the same actions as when it scanned the WorkSpace initially, except of course that instead of putting something on the RTCBACIL list, it just leaves it there, and when the user *isn't* on the recipient list, that name is removed from the RTCBACIL list (a user would cease to be a recipient if A) the Delivery process has already delivered it or B) the Server

process delivered it itself).

At this point, all the mail there is for the user has wound up in his InBox. The Server process may now send all the letters out to the client program.

9. A Look at Other Functions

9.1. The "delete letters" function

Deleting a letter is not something that needs to happen particularly quickly. That is, a user is not sitting around waiting for immediate results of a deletion. This fact is made use of in the algorithm to delete a letter.

When a client program gives a "delete letters" directive, the Server process processes each letter specified in the delete list-- that is, each letter the user wants deleted-- in the following manner. First, it verifies that the letter does in fact exist in the user's InBox directory (no point in deleting a letter he doesn't actually have). Then it sends the request to the Master, specifying the user and letter names; the Master in turn sends the request to the Lieutenant. The Lieutenant gets the request, marks it down, and sends back acknowledgement to the Master. Now that the Master and the Lieutenant have made note of what letters need to be deleted, the Master sends back acknowledgement to the Server process, who in turn sends acknowledgement to the client program. The user may go about his business.

Now, the Master examines its view of the world. For each letter in the delete list, it looks to see if that letter is in mid-delivery. If not, deletion may proceed in a straight-forward fashion. The Master sends out delete requests to the Delivery processes of the sundry **Maildriver** machines. When each machine has acknowledged completion of the deletion, the Master sends word to the Lieutenant, who now ceases to remember the deletion request.

If the letter *is* in mid-delivery, then something more elaborate is required in order to avoid the case of a machine attempting to delete a letter which it has not yet even received. Consider, for example, the story of John and Pauline using a **Maildriver** system configured for four machines, A, B, C, & D. Suppose that Pauline connects to machine A and invokes the "send letter" function to send a letter to John. Replication and delivery of the letter now proceeds as described in section 7. Meanwhile John connects to machine B and asks for his mail via the "receive mail" function. As it happens, B has already managed to deliver its copy of Pauline's letter to John's InBox, in time for John to receive it; John gets it and reads it right away. Now, having read it, he wishes to delete it, and issues an appropriate "delete letters" directive. However, despite the fact that Pauline's letter has already been successfully delivered to machines A, B, and D, it seems that machine C is still in the process of delivering it. Things could get real sticky if the Master were to send out the deletion request right at this moment, because C could find itself attempting to delete an as-yet undelivered letter. This is precisely the situation we wish to avoid.

The way around this problem is to simply defer deletion until delivery is completed. Thus, if the Master sees that the letter to be deleted is in mid-delivery, it

puts the letter on a queue (the Delete queue), and forgets about it. When a delivery completes, the Master checks the Delete queue; if it finds the just-completed letter there, it performs the deletion as above. While this algorithm may not be the most efficient time-wise, it *is* clearly the simplest and cleanest. Moreover, as already noted, speed is *not* a prime concern in letter deletion.

9.2. The "list letters" and "message-polling" functions

These two functions are very similar in nature and are both handled in a trivial fashion for efficiency reasons. In either case, the Server process simply looks at the InBox directory for the user on its local **Maildriver** machine. For "message-polling", it returns true or false depending on whether or not there is mail waiting in the user's InBox. For "list letters", it returns the list of what letters exist in the InBox.

Again, note that no attempt is made to garner information from the rest of the **Maildriver** system. For "message-polling", the reason is that message polling is likely to happen so frequently that it is desirable to have it be as small a drain on the **Maildriver** system as possible. The "list letters" function, on the other hand, is likely to be invoked rather rarely, so much so that it hardly seems worth the effort to race the user's mail through just to get it into the list.

9.3. The administrative functions

In general, for all administrative functions, the Server process just passes the request along to the Master [The Server first verifies that the user is privileged to perform an administrative action, but how this is done is outside the scope of this paper].

The "add user" and "delete user" administrative functions are really very trivial to do. Again, the Master first off informs the Lieutenant of the request. Then the user is added/deleted from the system namespace somehow [again, outside the scope of this paper], and requests to add/delete the InBox for that user are sent out. When the Master gets back all the acknowledgements, the Lieutenant is informed that all went well.

The "change master" function would likely get very little use. Its only purpose is to allow the administrators to force a specific **Maildriver** machine to be the Master. This would only happen if, for instance, one of the **Maildriver** machines was more powerful than the others and therefore ought to carry the extra burden of being Master. If this were the case, though, it would normally be appointed the Master anyway by the default startup routines, so this function would only be useful after the more-powerful machine had crashed (whereupon another machine became master) and was now up again.

To implement the function, the Server process, again, sends the request to the current Master. The current Master sends out messages to the Caretaker and Sender processes which mean, in English, "Finish up what you're doing and exit". Then it sends all its special Master data (the contents of the Master directory) to the Master-to-be, who now takes over as Master (the old Master sends out an announcement to all the **Maildriver** machines about the change). If the new

Master had previously been the Lieutenant, it must now, of course, appoint some other machine Lieutenant and send over the Lieutenant information.

Part III. Robustitude

10. When a server crashes

10.1. When a server goes down

Eventually, of course, machines crash. When a **Maildriver** machine crashes, life goes on. Steps are taken to see to it that things proceed smoothly. In general, the idea is to keep track of everything which happens while the machine is dead, so that it may catch up when it comes back up.

It is the job of the Caretaker to periodically poll the sundry **Maildriver** machines to verify that they are still up and running. If it detects a down machine, it places the name of that machine in a Dead Machine List (DML). Then it creates a list for that machine, called the Dead Machine Catchup List (DMCL), which will contain all actions which the dead machine will need to perform to get caught up when it eventually comes back to life. It also notifies the Master of the death. The Master does several things to deal with the death. First, changes its own file descriptor with which it normally communicates with that machine; it changes it to communicate instead with the Caretaker. Thus, it will be able to continue normal processing in a normal fashion-- requests for the dead machine are now simply re-routed to the Caretaker. Next, it sends word of the death to the Sender process for the dead machine, which then proceeds to send everything in its queue to the Caretaker. The Caretaker, then, receives all requests for the dead machine, both initially from the Sender, and subsequently from the Master, as normal processing continues. Finally, the Master, not surprisingly, sends word of the death to the Lieutenant, which keeps a DML and DMCLs of its own. Since the Lieutenant already receives special notification of each request in the system as it happens, it does not need additional notification for each dead machine request. When it gets general notification, it simply uses its own DML to put things in DMCLs as appropriate.

Note, of course, that if the machine that died was the Lieutenant, then the Master must promote a new Lieutenant first thing before it can do anything else. It may choose the next Lieutenant in any number of ways: for example it may pick the machine with the lightest load; it might also resort to a built-in numbering scheme. The choice of next Lieutenant is not too critical; what *is* critical is that one be chosen and sent all relevant information (pending sends, pending deletes, DML's and DMCL's).

Normal processing now continues amongst the living machines, except that no further requests are sent to the dead machine-- they instead end up at the Caretaker, on an appropriate DMCL (and on the Lieutenant's duplicate DMCL). The exception is for delete requests. If a dead machine requires a delete request, the DMCL is first scanned to see if it contains a delivery request for the same letter; if so, then the delivery request is removed and the delete request is discarded.

No sense delivering a message which will immediately be deleted!

Also, if a given request is for letter delivery, both the Caretaker and the Lieutenant record only the filename of the letter in the DMCL -- not the whole letter. This is simply to save time. The Caretaker, however, can easily get the letter anyway. It just makes a link into the Cemetery directory of the copy of the Protoletter that exists in Master directory. We are guaranteed that a copy exists there since the Master is currently sending out requests to deliver that letter.

10.2. When a server comes back up

When a dead server comes back up, it starts out by cleaning up its WorkSpace, getting rid of useless things, and finishing up what it can. It goes through all the Protoletters it finds there; if the Protoletter's State flag is set to "bogus", it simply deletes the Protoletter (this would be the case if the machine was in the middle of receiving the Protoletter when it crashed). Otherwise, each Protoletter accurately shows the state of delivery of that letter; the server may simply pick up delivery where it left off, since the header contains a list of who still needs to receive the letter.

When it has cleaned up its WorkSpace, it is ready to get connected to the rest of the system and get brought up to date. To do this, it broadcasts a message to all the **Maildriver** machines, a message which translates into English as "I'm here! I'm here! So who's the Master these days, anyway?". Only the Caretaker sends back a response, identifying itself as Caretaker on the Master machine. The Caretaker then proceeds to provide the reborn machine with requests, which it takes from the appropriate DMCL. When the DMCL has been emptied, the Caretaker removes the reborn machine from the Dead Machine List, and notifies the Master. The Master changes its file descriptor back to the Server for the reborn machine and sends word to the Lieutenant, which removes that machine from *its* DML and destroys that machine's DMCL. The machine is now officially back up.

Note that the Caretaker's DMCL is updated each time a request is successfully completed from it (that is, each time the reborn machine successfully completes the request). Thus, if the reborn machine dies again before the DMCL is emptied, the Caretaker still has an up-to-date list of what needs to be done to catch-up that machine. However, the Lieutenant is not informed upon successful completion of each catch-up request. The likelihood of two machines, including the Master, going down at once is rather lean; in that rare event, the reborn machine will simply be issued a few redundant requests which will be essentially ignored [again, writing a file which is already there is defined to be a noop, as is deleting a letter which is not there]. This seems a reasonable tradeoff.

11. When the Master crashes

Eventually, the fates conspire to be truly nasty, and the Master will crash.

It is the job of the Lieutenant to poll the Master regularly to make sure it is still up. If the Lieutenant finds that the Master is dead, drastic measures are in order. The Lieutenant takes over and becomes Master. Fortunately, it has been

provided with sufficient information and is up to the task.

The first thing that the Lieutenant does is to appoint a new Lieutenant (see section 10). It then writes out all its special information to appropriate files which will be read by the master processes once they get started. Finally, it execs the Master process.

The Master process now starts up as usual, setting up connections and forking processes off. It broadcasts a message to all surviving **Maildriver** machines; in English, this message would be akin to "The Master is dead! Long live the Master!! Anyway, I'm the new master, so please send me any requests from now on, including anything for which you are still waiting for acknowledgement from the old Master".

After everything is set up, it reads the file left behind for it by the Lieutenant. This file contains all requests which were begun but uncompleted when the old Master died. The new Master reads this file and re-issues each of these requests from scratch. This will perhaps cause some redundant requests to go out, but again this will not harm anything (hey, nobody ever said that replacing the Master was going to be easy!). As usual, a delete request for a partially delivered letter is queued until delivery is complete. If a request is a delivery request, then the Lieutenant will only have the filename in his records. To get the actual file, it first searches its own storage. If it cannot find it there (i.e. it never received its copy), it contacts the original recipient machine (again, the original recipient machine is part of the filename) to get the file. If the original recipient machine is not available (i.e. is dead), it broadcasts a desperate request for any machine that has the file. If no copy can be found, then delivery will have to wait until a machine with a copy comes back to life. It should be emphasized that this would be an *extremely* rare occurance. In this case, it must put the filename in a special list, which it checks against every time a machine comes back to life. And, of course, it notifies the new Lieutenant about this.

By now, there is a new Caretaker. The Caretaker may also find files for it left behind by the Lieutenant. The Caretaker examines the Cemetery directory, looking for files of the form "FromLieu.<machine>". Such a files contains the DMCL for that machine. The Caretaker reads these files in and builds its lists. It, too, must go through a desperate search for any sending files and uses the same algorithm as the Master did. When it has finished all this, it then simply proceeds to function as Caretaker.

When the old Master comes back up, it goes through the normal restart procedures (see section 10). It ceases to have any Master status and erases everything in its Master and Cemetery directories (similarly, when an old Lieutenant comes back up, it erases its Lieutenant directory. In short, whenever a machine comes back up, it erases both its Master and Lieutenant directories). The only way for that machine to become Master again is for an administrator to issue a "change master" request.

12. When a process crashes

The **Maildriver** system consists of many different processes interacting together. It is possible that any given process could fail for a number of reasons. While **Maildriver** contains many checks and balances to handle *machines* crashing, to have the same checking on a *per-process* basis would be absurdly expensive. On the surface, this would appear to be something of a problem, but in fact it is not so. To see this, one must consider the reasons that a process might die.

Processes can die for any of the following reasons: 1) The process exits in a normal fashion, (i.e. a call to exit()), 2) An error occurs and the process receives some sort of signal about it (such as SIGBUS -- Bus Error), 3) The process is sent some sort of signal from another process (such as SIGKILL -- absolutely kill the process), or 4) The operating system glitches badly and somehow loses or corrupts critical information about the process.

Case 1 can happen on occasion under controlled circumstances, in which case the loss of the process is expected and no problem, and cannot happen otherwise because of the way the code is written.

Cases 4 should not ever happen; if it does, it means that something is *seriously* wrong with the system (i.e. hardware), and it is reasonable to expect the system to crash imminently *anyway*, so it is pointless to try to do anything about it, and in any case **Maildriver** already handles the case of a machine crashing, so there is no problem.

Cases 2 and 3 appear more interesting. Assuming that the system has already been properly written, debugged and *severely* tested, neither case should ever occur. If either *does* occur, it is likely (though not definite) that something is seriously wrong with the system. It *might* be possible to take some sort of corrective action, but in all likelihood human intervention is desirable, to find out just what's wrong. In light of this likelihood, and in view of the fact that tracking dead processes is expensive, the **Maildriver** solution, radical though it may seem, is as follows: crash the system. Yes, you read that right. Crash the system. Although first a message should be written somewhere where a human will find it, explaining the decision to crash. Actually, crashing the system is not as radical as it sounds. Operating systems do it when they find bad inconsistencies; **Maildriver** is just doing the same thing. And again, it is easily handled, since there already exists safety valves for dead machines.

13. Summary of Robustitude

Maildriver is clearly very robust. Because mail is replicated to all **Maildriver** machines, a user can connect to any machine and get full service. In particular, if a machine crashes while a user is using it, his client program can simply connect to another machine and repeat the last request, with no loss of functionality. Since every action taken by the system requires acknowledgement and unacknowledged requests are repeated, one way or another, there can be no lost or partially completed requests. If acknowledgement is given, then the request is guaranteed to be safely at the state indicate by the acknowledgement. And no damage can be done by redundantly repeated requests. Since a dead machine is caught up to the rest of the world when it comes back on line, there is never any

inconsistency in the system. And since the Lieutenant watches over the Master, no particular **Maildriver** machine is critically important; any can fail and life will still go on. And if a process should fail, it brings the whole machine down with it, to insure no inconsistencies accrue.

In short, every eventuallity is covered. A user need never be aware of the state of the system. As long as he can connect to a live machine, he has full services available to him.

Part IV. Conclusion

Maildriver is a highly reliable system for computer mail. If it has any serious flaws, they are likely in the realm of time efficiency. Full replication is expensive. Still, it is not clear just how bad and in what ways the system clogs up. None the less, a few possibilities for possible improvement suggest themselves and are mentioned here, just as a starting point. In practice, one would have to experiment and find just the right combination.

One clear potential problem is the overloading of the Master. It is not clear just how much the Master machine will burdened by its extra responsibilities. Hopefully the extra burden will not be too bad. If it is, though, certain tasks could potentially be delegated to other machines, at varying loss of reliability. For example, the Lieutenant could be put in charge of, say, catching up a machine which has been down. The algorithm would remain the same, with the roles of Lieutenant and Master switched.

Another possibility which might improve performance would be a change in the letter delivery algorithm. Instead of the recipient machine sending the Protoletter to the Master, who then distributes it across the system, one could instead have the recipient machine simply send a message to the Master saying that it has received a new letter. The Master would then direct the other machines to get the Protoletter directly from the recipient machine. While this would result in a substantial lowering of the Master's overhead and distribute it more evenly across the network, there is a price to be paid in terms of reliability. Specifically, the recipient machine would have to acknowledge to its client before a second copy of the Protoletter was made; this would leave open the possibility that the recipient machine could crash after acknowledging but before actually effecting replication. Thus, the acknowledgement to the client was in fact a lie. It follows that there are only two ways to avoid telling this lie. One would be to delay acknowledgement until all machines have taken their copy; this is silly as it results in more overhead than the original scheme, and keeps the user waiting to boot. The other would be to wait until at least one other machine has gotten it. The problem with this is that it creats additional overhead in terms of keep track of just who has what copies of what. The logical extension to fix *this* problem is to not wait until any old machine gets it, but to wait until some specific machine has it. This brings us back to where we started from, since this is essentially the original Master scheme already presented. Given then that there is no way to decrease the overhead without losing reliability (and credibility), it comes down to playing trade offs. In general this change would seem an ill advised move unless the Master was severely overloaded and the change would remedy the situation in a noticeable way.

It is also not clear just how long the "receive mail" function will take in terms of retrieving partially delivered mail. It may be that the extra time may render it not worth it for some/many/most users. It would be a very simple thing to add a "quick receive mail" function which would just fetch the mail off the local machine. At this writing, though, it is not clear that this is necessary.

In any case, despite its possible drawbacks, **Maildriver** seems a solid system. It clearly appears to meet its design goals of high reliability, full access, and efficient bulk mailing.

And it certainly has a catchy name.

Appendix: C/Unix-based Pseudo-code of Master Processes

Note: we assume the following bug fixed:

fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | FASYNC)

works for sockets (and hence pipes), and that moreover, the signal handler which processes the SIGIO receives the value of fd as the arg "code". (see fcntl(2), sigvec(2))

```
/**************************************************\

            M A S T E R

   This process is the master. It must set things
   up initially, fork off Sender and Caretaker
   processes, and appoint a Lieutenant. Then it
   proceeds to perform the normal duties of
   coordinating Maildriver.

\**************************************************/

#include <signal.h>, etc.

#define DONT_CARE (-1)

/*------------------------------------------*/

int lieutenant, caretaker, *machines, *save_machines;   /* file descripters */

something *ActiveMachineList;          /* list of active machines */

extern void EmergencyBailout();

struct {   char *filename;
        int machine;
     } *DoYouHaveIt;         /* list of files we couldn't find */

/*------------------------------------------*/

HaveLetterDeleted(message) whatever message;

{  char *filename = get filename from message,
       *user = get user who wants deletion from message;
   int machine;
   FILE *pending = create <filename>.deleting in Master directory;

   for (each machine)
      {  send a deletion request to machine, including
           filename and user;
         write machine to pending;
      }
```

```
   fclose(pending); }

/*------------------------------------------*/

SendLetterToMachine(message, machine, fp) whatever message; int machine; FILE
*fp;

{  if (machine == original server)
      send "mark valid and process" to it
     else
   if (machines[machine] == caretaker)
      send filename of letter to it
     else
      send entire letter to it;
   get "I got it" from machine;
   write machine to fp; }

/*------------------------------------------*/

SendOffLetter(message, first_machine) whatever message; int first_machine;

{  int machine;
   char *filename = filename in message;
   FILE *pending = create <filename>.sending in Master directory;

   if (first_machine != DONT_CARE)
      SendLetterToMachine(message, first_machine, pending);

   for (each machine)
      if (machine != first_machine)
         SendLetterToMachine(message, machine, pending);

   fclose(pending); }

/*------------------------------------------*/

ProcessLifeOrDeath(message); whatever message;

{  int machine = the machine mentioned in the message;

   if (machine has died)
      {  send message to machines[machine] about death;
         machines[machine] = caretaker;
         if (machine == the Lieutenant)
            {  lieutenant = pick & connect to some random lieutenant;
               inform it that it is now lieutenant;
               for (each .sending & .deleting file in
                    the Master directory)
                  send the send/delete request to lieutenant;
               send it the ActiveMachineList;
            }
      }
     else
   if (machine has come back to life)
```

```
        {  send message to machines[machine] about rebirth;
           machines[reborn machine] = save_machines[reborn machine];

           for (each entry in the DoYouHaveIt list)
              if ((machine == (othermachine = machine in entry)) &&
                 ((ask machine if it has file in entry) == FOUND))
                 {  read file from machine into Master directory;
                    remove entry from DoYouHaveIt list;
                    send the file to othermachine;
                 }

        }

     inform lieutenant of birth or death; }

/*-----------------------------------------*/

ProcessRequest(message); whatever message;

{  int machine = requesting machine from message;

   switch(type of request)
      {  SendLetter:        read letter from server and
                              put it into the Master directory;
                          send filename of letter to Lieutenant;
                          await "I got it" from Lieutenant;
                          send "I got it" to server;
                          put filename of letter on delivery queue;
                          break;

         DeleteLetters:     send request to Lieutenant;
                          await "Got it" from Lieutenant;
                          send "Got it" to server;

                          check for letter in Master directory;
                          if (its there)
                             put message on delete queue;
                            else
                             HaveLetterDeleted(message);
                          break;

         ReceiveMail:       recipient = get recipient from message;
                          for (each letter in delivery queue)
                             if (recipient will receive letter)
                                SendOffLetter(message, machine);
                          break;

         AddUser:           send message to Lieutenant about addition;
                          await "Got it" from Lieutenant;
                          fp = creat add.<user>;
                          for (each machine)
                             {  send add request to that machine;
                                write machine to fp;
                             }
```

```
                    fclose(fp);

                    break;

      DeleteUser:       send message to Lieutenant about deletion;
                    await "Got it" from Lieutenant;
                    fp = creat del.<user>;
                    for (each machine)
                       {  send delete request to that machine;
                          write machine to fp;
                       }
                    fclose(fp);

                    break;

      ChangeMaster:     send message to new master about change;
                    send all Master files & queues to new
                      Master;
                    for (each machine)
                      send message about change of master;
                    send message to caretaker about change;
                    send message to lieutenant about change;
                    exit(0);
   } }

/*---------------------------------------------*/

ProcessAcknowledgement(message) whatever message;

{  char *filename = get filename of letter from message;

   switch(type of acknowledgement)
     {  delivered letter:  open <filename>.sending in Master directory;
                    delete acknowledging machine from it;

                    if (<filename>.sending is now empty)
                       {  send "forget it" to Lieutenant;
                          await "I got it" from Lieutenant;
                          delete <filename>.sending;
                          delete <filename> from Master dir.
                          if (filename is on delete queue)
                             {  remove it from queue;
                                HaveLetterDeleted(filename);
                             }
                       }

                    break;

      finished deletion:  open <filename>.deleting in Master directory;
                    delete acknowledging machine from it;

                    if (<filename>.deleting is now empty)
                       {  send "forget it" to Lieutenant;
                          await "I got it" from Lieutenant;
```

```
                        delete <filename>.deleting;
                        delete <filename> from Master dir.
                     }

                  break;

       finished add user: open add.<user> in Master directory;
                  delete acknowledging machine from it;

                  if (add.<user> is now empty)
                     {  send "forget it" to Lieutenant;
                        await "I got it" from Lieutenant;
                        delete add.<user>;
                     }

                  break;


       finished del. user: open del.<user> in Master directory;
                  delete acknowledging machine from it;

                  if (del.<user> is now empty)
                     {  send "forget it" to Lieutenant;
                        await "I got it" from Lieutenant;
                        delete del.<user>;
                     }

                  break;

     } }

/*-----------------------------------------------*/

void MessageHandler(sig, fd) int sig, fd;

{  read message from fd;
   if (fd == caretaker)
      ProcessLifeOrDeath(message);
    else
   if (message is new request)
      ProcessRequest(message);
    else
   if (message is an acknowledgement)
      ProcessAcknowledgement(message);

   send "Got it." to fd; }

/*-----------------------------------------------*/

StartProcesses() {  int *for_caretaker = calloc(# of active machines, sizeof(int)),
      machine;

   machines = calloc(# of active machines, sizeof(int));
```

```
    /* first, set up all the pipes we're gonna need */

    for (machine = 0; machine < # of active machines; machine++)
       {  get two pipes;
          put one into for_caretaker[machine];
          put the other into
             both machines[machine] and save_machines[machine];
          call fcntl to set them up to be asynchronous;
       }
    caretaker = yet another pipe;

    for (machine = 0; machine < # of active machines; machine++)
       {  fork off a new process;
          exec sender with args:
             machine, machines[machine], for_caretaker[machine];
       }
    fork off another process;
    exec Caretaker with for_caretaker array as args;

    for (machine = 0; machine < # of active machines; machine++)
       close(for_caretaker[machine]); }

/*------------------------------------------*/

/* on a change of master, we only have a filename.  we
   need the actual file to send!  desparately try to
   find the file */

FindFile(message) whatever message;

{  char *filename = filename contained in message;

    if ((scan each user InBox for filename) == FOUND)
       link(found file, Cemetary);
     else
    if (machine contained in filename /* machine which originally
       received the letter */ is not on DeadMachineList)
       {  send request for that file to that machine;
          read letter from machine into Cemetary;
       }
     else
    if ((ask every other machine if they have it) == FOUND)
       read letter from that machine into Cemetary;
     else
       put the filename and the machine that needs it
          on the DoYouHaveIt list; }

/*------------------------------------------*/

main(argc, argv) int argc; char **argv;

{  signal(SIGSEGV etc, EmergencyBailout);

    if (argv indicates this is a fresh startup)
```

```
    {   read a startup file to find which machine is initial lieutenant;
        lieutenant = get socket connection to lieutenant;
    }
  else /* we are replacing a dead master */
    socket = whatever argv says the descriptor is;

read a startup file to build ActiveMachineList;
StartProcesses();

signal(SIGIO, MessageHandler);

if (we are replacing a dead master)
    {   read FromLieutenant file in Master directory
          and re-send all the requests found therein;
          if request is send letter
              FindFile(request);
        broadcast "I'm the new Master" to all machines;
    }

for(;;)
    {   sigpause(0);
        while (delivery queue is not empty)
            {   SendOffLetter(first thing on queue, DONT_CARE);
                remove first thing from queue;
            }
    } }

/*---------------------------------------------*/
```

```
/**************************************************\

          C A R E T A K E R

  The Caretaker is in charge of the cemetary.  In
  particular, it is responsible for knowing which
  machines are dead or alive.  When a dead server
  machine comes back to life, the Caretaker is
  responsible for bringing it up to date.

\**************************************************/

#include <signal.h>, etc.

#define NO_PULSE ??     /* NO_PULSE should have a numeric value indicating
                     how much time to wait while polling a machine
                     before deciding it is dead (i.e. has no pulse).
                     Infinately tweekable */

#define TIME_BETWEEN_POLLS ??   /* same kind of deal */

/*--------------------------------------------*/

int *machines, *pulse, master;

something *ActiveMachineList;   /* list of active machines */

struct {  char *filename;
          int machine;
     } *DoYouHaveIt;          /* list of files we couldn't find */

Queue *DMCL;          /* array of Dead Machine Catchup Lists */

extern void EmergencyBailout();

int FinishUpAndDie = 0;

/*--------------------------------------------*/

GetPostMortemRequest(message)
whatever message;

{  int machine = get machine from message;

   if (request in message is for letter deletion)
      for (each request in DMCL[machine] queue)
         if (request in queue is to send the letter
            that we now want to delete)
            {  remove the send request from the queue;
               remove the link in the Cemetery;
               send back "I got it" to master;
               return;
            }
```

```
    if (request is for letter sending)
        {   get filename from message;
            link(filename in Master directory, same name in Cemetery);
        }

    add request to DMCL[machine];
    send back "I got it" to master;
}

/*-------------------------------------------*/

SendPreNatalRequest(message)
whatever message;

{   int machine = resurrecting machine (from message);

    if (DMCL[machine] is not empty)
        {   send first thing on DMCL[machine] to machines[machine];
            if (first thing is send letter request)
                send letter over (using link in Cemetary);
            remove first thing from DMCL[machine];
        }
    else
        {   send "That's all folks!" to machine;
            add machine to active machine list;
            send "<machine> is back on line" to master;
        }
}

/*-------------------------------------------*/

ResurrectMachine(message)
whatever message;

{   int machine = get machine from message;

    send "Hi there.  I'm the master" to machine;

    for (each entry in the DoYouHaveIt list)
        if ((machine == (othermachine = machine in entry)) &&
            ((ask machine if it has file in entry) == FOUND))
            {   read file from machine into Cemetary;
                remove entry from DoYouHaveIt list;
                if (othermachine is dead)
                    put send request in DMCL[othermachine];
                else
                    send the file to othermachine;
            }

    SendPreNatalRequest(message);
}

/*-------------------------------------------*/
```

```
void AnswerThePhone(sig, fd)
int sig, fd;

{  read message from fd;

   if (message is change-of-master notification)
      FinishUpAndDie = 1;
    else
   if (fd == master)
      GetPostMortemRequest(message);
    else
   if (message is "Hey, I'm here!" from ex-dead machine)
      ResurrectMachine(message);
    else
   if (message is "Ok, got it.  What next?")
      SendPreNatalRequest(message);
}

/*------------------------------------------*/

/* on a change of master, we only have a filename.  we
   need the actual file to send!  desparately try to
   find the file */

FindFile(message)
whatever message;

{  char *filename = filename contained in message;

   if ((scan each user InBox for filename) == FOUND)
      link(found file, Cemetary);
    else
   if (machine contained in filename /* machine which originally
      received the letter */ is not on DeadMachineList)
      {  send request for that file to that machine;
         read letter from machine into Cemetary;
      }
    else
   if ((ask every other machine if they have it) == FOUND)
      read letter from that machine into Cemetary;
    else
      put the filename and the machine that needs it
         on the DoYouHaveIt list;
}

/*------------------------------------------*/

main(argc, argv)
int argc;
char **argv;

{  int machine;

   signal(SIGSEGV etc, EmergencyBailout);
```

```
    parse argv to build an array ("machines") of descriptors for
      communicating with the sending processes for each machine;
      /* the descriptors are already open-- we have inherited them--
        we just need to figure out which is which */

    master = whatever descriptor argv says goes to the master;

    initialize DoYouHaveIt list;
    read a startup file to build ActiveMachineList;
    request = allocate space for a request queue per machine;
    for (each machine on list)
       {  pulse[machine] = set up a socket connection (*synchronous*)
                    to that machine's pulse process;
          request[machine] = initialize queue for that machine;
       }

    for (each file in the Cemetary of the form FromLieu.<machine>)
       {  put machine in the DeadMachineList;
          read the file into DMCL[machine].  while doing so,
             check for and remove send/delete pairs;
             if (the request is a send request)
                FindFile(request);
          delete the file;
       }

    signal(SIGIO, AnswerThePhone);

    for(;;)
       {  for (each active machine)
             {  send "Are you there?" to pulse[machine];
                attempt to read "Yes I'm here" from pulse[machine],
                   but time out after NO_PULSE time;

                if (timed out)
                   {  remove machine from active list;
                      send "<machine> has died" to master;
                   }
             }
          sleep(TIME_BETWEEN_POLLS);
          if (FinishUpAndDie)
             for(;;)
                {  if (all the request queues are empty)
                      exit(0);
                   sleep(TIME_BETWEEN_POLLS);
                }
       }
}

/*-----------------------------------------*/
```

```
/**************************************************\

            S E N D E R

   This process accepts requests from the Master
   or Caretaker and sends them to its target
   machine (specified in "argv").

\**************************************************/

#include <signal.h>, etc.

/*------------------------------------------*/

int target, master, caretaker;  /* file descriptors */

struct q_record *Queue;         /* message queue */

extern void EmergencyBailout();

int MachineAlive = 1, FinishUpAndDie = 0;

/*------------------------------------------*/

InputHandler(sig, fd)
int sig, fd;

{  read message from fd;
   if (fd == target)
      {  verify that this is an acknowledgement meant for
            someone else (that is, Master or Caretaker);
            /* this includes "I am here!" for Caretaker */
         send message to the someone else;
      }
     else
   if (message is a death notification)
      MachineAlive = 0;
     else
   if (message is a rebirth notification)
      MachineAlive = 1;
   if (message is change-of-master notification)
      FinishUpAndDie = 1;
     else
      {  append message onto Queue;
         send back "I got it" over fd;
      }
}

/*------------------------------------------*/

/* note:  things can get added to the Queue (via interrupts) while
   this function runs, except during the critical section */

ProcessOutputQueue()
```

```
{  while(the Queue is non-empty)
     {  get first message on Queue;
        send it to target;
        while (MachineAlive)       /* could change via interrupt */
           attempt to read acknowledgement from target, but
           without blocking;

        if (! MachineAlive)
           goto DIED;

        sigblock(SIGIO);              /* begin critical section */
        remove this message from Queue;
        sigsetmask(0);               /* end critical section   */
     }
   return;

DIED:
   send entire Queue to caretaker;
}

/*-----------------------------------------------*/

main(argc, argv)
int argc;
char **argv;

{  signal(SIGSEGV etc, EmergencyBailout);

   sscanf(argv[2],"%d", &master);
   sscanf(argv[3],"%d", &caretaker);
   target = set up and open connection to target machine (argv[1]);

   /* arrange for connection to be asynchronous */
   /* as we will also be handling acknowledgements */
   /* back to the Master or Caretaker */
   fcntl(target, F_SETFL, fcntl(target, F_GETFL) | FASYNC);

   /* we have already inherited asynchronous pipes to the Master and
     Caretaker. install handler to talk to them */
   signal(SIGIO, InputHandler);

   /* now hang out and process interrupts as they come in */

   for(;;)
      {  sigpause(0);
         ProcessOutputQueue();
         if (FinishUpAndDie)
            exit(0);
      }
}

/*-----------------------------------------------*/
```

```
/***************************************************\

          LIEUTENANT

  The Lieutenant's job is to monitor the doings
  of the Master, so as to be able to take over
  if the Master dies.  It is also the Master's
  poller and initiates the take over if the
  Master shows no pulse.

\***************************************************/

#include <signal.h>, etc.


#define NO_PULSE ??     /* NO_PULSE should have a numeric value indicating
                   how much time to wait while polling a machine
                   before deciding it is dead (i.e. has no pulse).
                   Infinately tweekable */

#define TIME_BETWEEN_POLLS ??   /* same kind of deal */

/*---------------------------------------------*/

something *DeadMachineList;    /* list of dead machines */

Queue pending, *DMCL;  /* DMCL == Dead Machine Catchup List */

/*---------------------------------------------*/

void MessageHandler(sig, fd)
int sig, fd;

{  whatever message = read message from fd;

   switch(message)
     {  "sending letter":
        "deleting letter":
        "adding user":
        "deleting user":    put request on pending queue;
                  for (each machine on DeadMachineList)
                     put request on DMCL[machine];

        "letter sent":
        "letter deleted":
        "user added":
        "user deleted":     remove request from pending queue;

        "new master":       connect to new master;

        "dead machine":     put dead machine on DeadMachineList;

        "live machine":     delete DMCL[machine] queue;
                  remove machine from DeadMachineList;
```

```
    }

  send "I got it" to master;
}

/*--  ------------------------------------*/

BecomeMaster()
{  somehow pick a machine to be next lieutenant;
   set up connection to that machine (and inform him
      that he is now lieutenant);

   write the pending queue to the file FromLieutenant
      in the Master directory;

   for (each machine in the DeadMachineList)
      write a file FromLieu.<machine> in the Cemetary
      containing the contents of DMCL[machine];

   exec Master process (passing fd of new Lieutenant, and the fact
      that this is a replacement, not a fresh start);
}

/*----------------------------------------*/

main()
{  int machine;

   signal(SIGSEGV etc, EmergencyBailout);

   initialize empty DeadMachineList;
   request = allocate space for a request queue per machine;
   for (each machine on list)
      request[machine] = initialize queue for that machine;

   master = set up socket connection to master;

   /* arrange for connection to be asynchronous */
   fcntl(master, F_SETFL, fcntl(master, F_GETFL) | FASYNC);

   pulse = set up socket connection to pulse process on master;

   signal(SIGIO, MessageHandler);

   for (;;)
      {  send "Are you there?" to pulse;
         attempt to read "Yes I'm here" from pulse,
            but time out after NO_PULSE time;

         if (timed out)
            BecomeMaster();

         sleep(TIME_BETWEEN_POLLS);
      }
```

```
}
/*-------------------------------------------------*/
```

```
/***************************************************\

                PULSE

   The pulse process simply waits to be polled, so
   that it may demonstrate that its machine is
   currently up and running.  Thus, the poller is
   in fact checking the "pulse" of the machine.

\***************************************************/

#include <signal.h>, etc.

/*------------------------------------------*/

int poller;    /* descriptor of process taking pulse */

extern void EmergencyBailout();

/*------------------------------------------*/

void Pulsate()
{  read message from poller;
   verify that message is "Are you still there?";
   send back "Yup, still here";
}

/*------------------------------------------*/

main()
{  signal(SIGSEGV etc, EmergencyBailout);

   poller = set up and open socket connection to outside world;

   /* arrange for connection to be asynchronous */
   fcntl(poller, F_SETFL, fcntl(poller, F_GETFL) | FASYNC);

   signal(SIGIO, Pulsate);

   for(;;)
      sigpause(0);
}

/*------------------------------------------*/
```

END

DTIC

7-86