DISTRIBUTED FILE SYSTEMS IN AN AUTHENTICATION SYSTEM

by

John W. Merritt

B. S. E. E., University of South Carolina, 1979

A Master's Report

submitted in partial fulfillment of the

requirements for the degree

Master of Science

Department of Computer Science

Kansas State University
Manhattan, Kansas

1986

Approved by:

R A McBride

Major Professor

Dedicated

To my wife Jan, who watched over the homestead while I was away at school and to my two children, Trey and Emily. Also, acknowledgement to my major professor Dr. Rich A. McBride for his help and suggestions.

CONTENTS

LIST OF FIGURES

"DISTRIBUTED FILE SYSTEMS IN AN AUTHENTICATION SYSTEM"

1. INTRODUCTION

With regard to distributed databases[1], the issues of how to provide: user authentication for privacy and security, optimization of response times, and integrity of data are of rising concern in the computer industry. As an example, an automatic-teller-machine system provides authentication of the user and protection against illegal accesses to any user's bank account records. The proper user is verified by means of a personal card and his special code or password. If the password is correct, the system permits protected transactions on his account. The transactions are protected in that software mechanisms are provided to ensure the integrity of the user's bank records. All transactions and possible errors are recorded. Also, the transactions are carried to completion without an inconvenience to the user.

The Electronic Transfer Act in 1980 was passed to protect users against various losses from errors and unauthorized use [Fernandez 81]. However, even with governmental protections, if the user was not confident that the system provided authentication, reliable transactions, and fast response times, the system would not be useful.

1.1 SYNOPSIS

Using these ideas of authentication and a distributed file system, a secret mail facility was designed and implemented on the UNIX[2] operating system by a two member team. The implementation was written using the C-Programming Language [Kernighan 78].

1. Historically the terms distributed database and distributed file system have been used synonymously, though this usage is in error [Martin 81]. Throughout this paper the more appropriate phrase term file system will be used, but I will use database terminology where applicable.

2. *UNIX is a Trademark of AT&T Bell Laboratories

All mail that may be of a sensitive nature should be protected from outside probing or interception. Secret mail was designed to provide this protection with single encryption for local messages and double encryption during the remote transfer of messages. The messages remain encrypted until the intended receiver requests delivery. The user's data file, which contains information pertinent to the encrypting and decrypting of messages, is also protected from outside scanning and corruption. Only the user has permission to access the data file as long as the application program is executing on his behalf; outside the secret mail environment the user cannot read the data file. Software mechanisms are provided within the file system to ensure data integrity. The file system was organized so that the user would experience minimum response times.

The project was broken into three functional areas:

1. The Authorization Server
2. The Name Server
3. The Message Server

The Authorization Server provides the user interface, the authentication of the user, and message encryption and decryption. The Name Server provides the management of the file system and the generation of a unique conversation key. The conversation key is used for encrypting and decrypting messages. A conversation is when one user sends a encrypted message to another user, who can decrypt the same message using the conversation key. The Message Server provides retrieval and decryption of messages that have been sent. The implementation of the Message Server was done jointly by both members of this project team, while the Authorization Server and the Name Server were implemented by each team member. The main emphasis of this paper is on the Name Server and the Message Server's interactions with the file system.

The main chapters deal with concerns in distributed file systems, the detailed design of the Name Server and the Message Server, a comparison of this project to similar systems, and a conclusion. In particular, chapter two covers the concerns of a distributed file system; the areas discussed are the optimization of user response times by using files similar in structure to a relational database [Codd 70], the centralization and decentralization of a file system, and the issue of file system security. As a foundation for a discussion of the detailed design of the Name Server and the Message Server in chapter three, a step-by-step analysis of the secret mail facility and user commands are provided. The Name Server and Message Server are discussed in detail with emphasis on: the optimization of users' response times, data integrity, and how the servers interact with the file system. Also presented are user scenarios of file system accesses, and a discussion of how both the Name Server and the Message Server maintain local and remote data integrity. In chapter four a comparison is done against the following systems:

1. GRAPEVINE [Birrell 82]

2. CSNET [Solomon 82]

3. UNIX™ "Secure Mail" facility [UNIX 83]

These systems were selected due to their capabilities for transferring files between users. The final chapter discusses how the implementation of this project could be of value to others, and it also summarizes known problems and suggestions of future enhancements.

2. *MAJOR CONCERNS IN A DISTRIBUTED FILE SYSTEM*

2.1 DEFINITIONS

A distributed file system is a dispersion of files across many computers. A file system should not be confused with a database. Databases use the concept of data/program independence, whereas a file processing system is data dependent [Cardenas 79]. Each application program in a file processing system contains the definition of data as used in a file. In a file processing system any change in the attributes of a file, even minor changes, involves recompiling the application program.

Individual application programs or file servers perform transactions on a file system. A file server is a program that only accesses and transfers or stores portions of a file and nothing else, whereas an application program will perform many functions outside the accessing of files. Transactions are defined as a sequence of actions on data items in a file which preserve the consistency of the data [Eswaran 76]. An action, in this context, is a single event, such as, reading or writing a data item to a file.

2.2 OPTIMIZATIONS

2.2.1 File Structures

A concern for the success of a distributed file system is in optimization. The optimization or efficiency of a file system is measured in terms of response times that users experience during transactions. Response times are generally dependent on the amount of data accessed, but in a distributed file system, data communications between computers is usually a more dominant factor [Ullman 82]. Transactions need to be handled promptly or the system will be effectively unusable. One way of optimizing is by the organization of the file structure. The organization of the data can significantly influence response times. The data needs to be easily accessible and structured for quick file reads and writes. The relational model is a structure designed for quick accesses of data [Codd 70]. The data resides in what is called a flat file, where each record instance has a similar number of fields. The other data structures' hierarchical and network models are not limited to being in a flat file structure [Cardenas 79]. The relational model provides a simple approach to accessing data. The data in a relational model is stored in the form of tables called relations. These relations are made up of fields called attributes. An attribute or a set of attributes which when given values will uniquely determine a tuple in the relation is called a key [Cardenas 79]. The key is used to access a particular row or tuple in the table. The tabular structure of the secret mail project's file system is similar to relations. Each individual user will have a file that is associated only with him. The user's file is structured in the form of a table with columns and rows of data. Each row is one relation of data attributes. In this approach, the Name Server is able with a key to find the proper records, once the user's identification is known. The UNIX™ operating system provides all users with a unique identification [Ritchie 74]. In addition, the operating system provides a file management system; therefore, the mechanisms used in finding and accessing files are quick and efficient [Ritchie 74].

2.2.2 *File System Organization*

Another means of optimization is through the distribution of the file system. There are three basic types of file system organizations:

1. Centralized
2. Decentralized
3. Combination of both [Martin 81]

There has been much controversy over which is the optimum choice. In a centralized system all computers communicate with a central computer that is responsible for maintaining a master copy of the data files. In a decentralized system the data is fully replicated, i.e. each system has a copy of all the data files. The third choice is to have a partially replicated system. In this case, the system is said to be both centralized and decentralized. There are some advantages and disadvantages to having a decentralized system. A decentralized system is more reliable since a failure of one computer does not bring down the total system. The storage of frequently used data in a decentralized system can also be located at optimum places for fast accesses, which means quick user responses. However, a disadvantage is that updates in a distributed system are more complicated due to the need to keep the various file tables identical despite network delays, corrupted messages, and system failures [Tanenbaum 81]. Security in a decentralized file system is also an issue, since multiple locations will need protection [Fernandez 81]. In a centralized system only one location has to be concerned about security.

The secret mail file system structure is a combination of centralized and decentralized. This organization worked well with the project's application. There are unique files on each computer (therefore, providing for a centralization of certain files), and there are files that have to be decentralized, since the files need to be replicated on each computer. Generally the replicated files are maintained by the Name Servers on each computer. However, the file that contains the list of available computers within the secret mail network can only be changed by an administrator.

2.2.3 Reliable Communication Network

Another means of optimizing a file system is by having a reliable and fast communication link between computers. The Open Systems Interconnection(OSI) reference model that was developed by the International Standards Organization(ISO) is used in the design of networks today [Tanenbaum 81]. The OSI reference model provides for seven layers of protocol, namely:

1. The Physical Layer
2. The Data Link Layer
3. The Network Layer
4. The Transport Layer
5. The Session Layer
6. The Presentation Layer
7. The Application Layer

File system transactions are performed in the application layer. This is under the control of the application programmer. The presentation layer performs any data transformation that may be needed. Text compression, encryption, and any virtual-terminal protocol or file transfer protocol are all handled in the presentation layer. Transactions that link to other machines depend on the session and transport layers. The session layer provides process to process connection. The transport layer provides the reliable host to host communication link protocol that is needed [Tanenbaum 81]. The other three layers of the OSI model are also of importance, but primarily deal with routing and data transmission in a communication subnet.

2.3 DATA INTEGRITY

2.3.1 Concurrency Control

2.3.1.1 File Locks

In the design of a distributed file system, a sub-system for concurrency control is needed to maintain data integrity. Two processes that try to write to the same file have the potential for corrupting the data in the file. The file can be corrupted with writes intermixed from both processes. There are several methods to control concurrency problems. One method is the "wait" principle. If two processes both try to execute transactions that conflict, one must wait for the other to finish before proceeding. The mechanism used here is called locking [Bhargava 82].

A locking operation is when a process generates a flag or lock, such as, the creation of a file, as a signal to all other processes that a another particular file is being updated. All other processes before accessing the particular file for updates would first verify that a lock has not been created. If the lock is detected, then the process waits until it is removed. Once the lock is removed, then the second process would repeat the same scenario as the first process in generating a lock.

One of the most widely used methods is called two-phase locking [Eswaran76]. The requirement is that in any transaction, all lock operations must proceed all unlock operations. One characteristic of having a locking mechanism of this type is that the system makes the transactions visible as opposed to transparent. Therefore, on failures the sub-system has to manage the removal of the locks that are left or the system can deadlock [Tanenbaum 81]. Deadlock can occur in this situation since no process will be able to access the data, even though, there are no contentions.

2.3.1.2 Time-Stamping

Another method of concurrency control is called time-stamping. The time-stamp determines the order in which transactions are executed. For instance, the transaction that is the oldest would go first in a first come first serve scheduling of transactions [Lampson 78].

2.3.1.3 Concurrency Control in Secret Mail

The secret mail project uses the two-phase locking mechanism. All files that could have conflicts in writing, are guarded by means of the locking mechanism. A lock file is created in the same file directory as the file that needs to be accessed. The Name Server checks to see if a file with the file name "file-name_lock" is found. For example, if the Name Server needed to access the data file called "user", it would first check to see if a file called "user_lock" existed. If the lock file existed, the Name Server would wait until the lock file was removed before accessing the file "user". However, if the lock file was not removed after a fixed amount of time, the Name Server would record an error in the ERRLOG directory. An administrator would have to remove the lock file before that particular user's file could be used. Assuming the lock file was removed, the Name Server, just before accessing the user's file, would create another "user_lock" file to warn other Name Servers processes the user's file is in use.

2.3.1.4 Atomic Transactions

Data integrity can also be controlled by having atomic transactions. Atomic means that either the transaction is carried out to its completion or not at all. For example, a transaction may have the need to do several disk accesses, there is alway danger that the system will crash halfway through the transaction and leave data in an inconsistent state. The central idea in an atomic transaction is in two phases [Lampson 78]:

1. Record the information necessary to do the writes in a set of intentions without changing the data stored by the system. The last action taken in this phase is said to commit the transaction.

2. Perform the writes, actually changing the stored data.

Theoretically, in an atomic transaction, if a crash occurs after the transaction commits ,but before all changes are made, then the second phase is restarted. The restarts can happen as many times as necessary, to make all the changes. The writing of the intentions set also needs to be atomic.

2.3.1.5 Record Errors

As usual, there is no guaranteed protection from all types of data corruption. The best that can be done is to provide the mechanism that best suits the application. Even though the data can become inconsistent, from operating system errors or hardware malfunctions, there are still methods to correct some of the problems. One way is to record known errors that are seen by a file server into an err_log file. Using this information an administrator of the system, if possible, can correct the problem. The administrator is the supervisor of the file system, who provides services that requires human intervention to the system, such as, rmoving lock files. Even if the inconsistences cannot be corrected, keeping an err_log file is good software practice. Also, the err_log file may point to potential software problems that could be corrected. It is clear that without some scheme of concurrency control the file system could be corrupted.

2.4 FILE SECURITY

2.4.1 Authentication

An area of concern in security is the malicious or accidental corruption of a file. In both cases the file needs to be protected. Protection applies to both distributed and nondistributed systems. The main goal is to allow only authorized users the right to read and write within file, unauthorized users should not be able to access the file or prevent an authorized user from accessing it. One common method is to authenticate the user by requiring a password before accessing a file [Davies 84]. In an authentication service the password is stored and usually encrypted. A common encryption scheme used is the Data Encryption Standard(DES) [NBS 77]. In the secret mail project, the password is verified by the authorization server using DES before the Name Server is called to access the file.

2.4.2 Permission Rights

Another method of security is by having the permission rights set so that only a privileged program is allowed access [Cardenas 79]. Generally this is a mechanism that can be provided by the operating system. The UNIX™ operating system provides the owner of a file with the control to set all permissions. In the secret mail software, the permission rights are set so only the secret mail facility has permission to read and write. Permissions are set so that a user can never access a file without going through the Name Server of the secret mail system. The permission rights are set by using a facility in the UNIX™ operating system of setting the user's process identification to the that of the owner during the execution of the secret mail program.

3. DETAIL DESIGN of the NAME SERVER and MESSAGE SERVER

3.1 SECRET MAIL SYSTEM OVERVIEW

Appendix 3 contains functional block diagrams of the Name Server and the Message Server modules, and the source code to the Name Server and the Messager Server. In this chapter, the term user and sender will refer to the person who initiated the first sequence of a command, and the receiver is the member toward whom the sender or user has directed an action. The following list are the commands available in secret mail:

1. enrollx
2. connectx
3. sendx
4. disconnectx
5. getx

6. editx

7. helpx

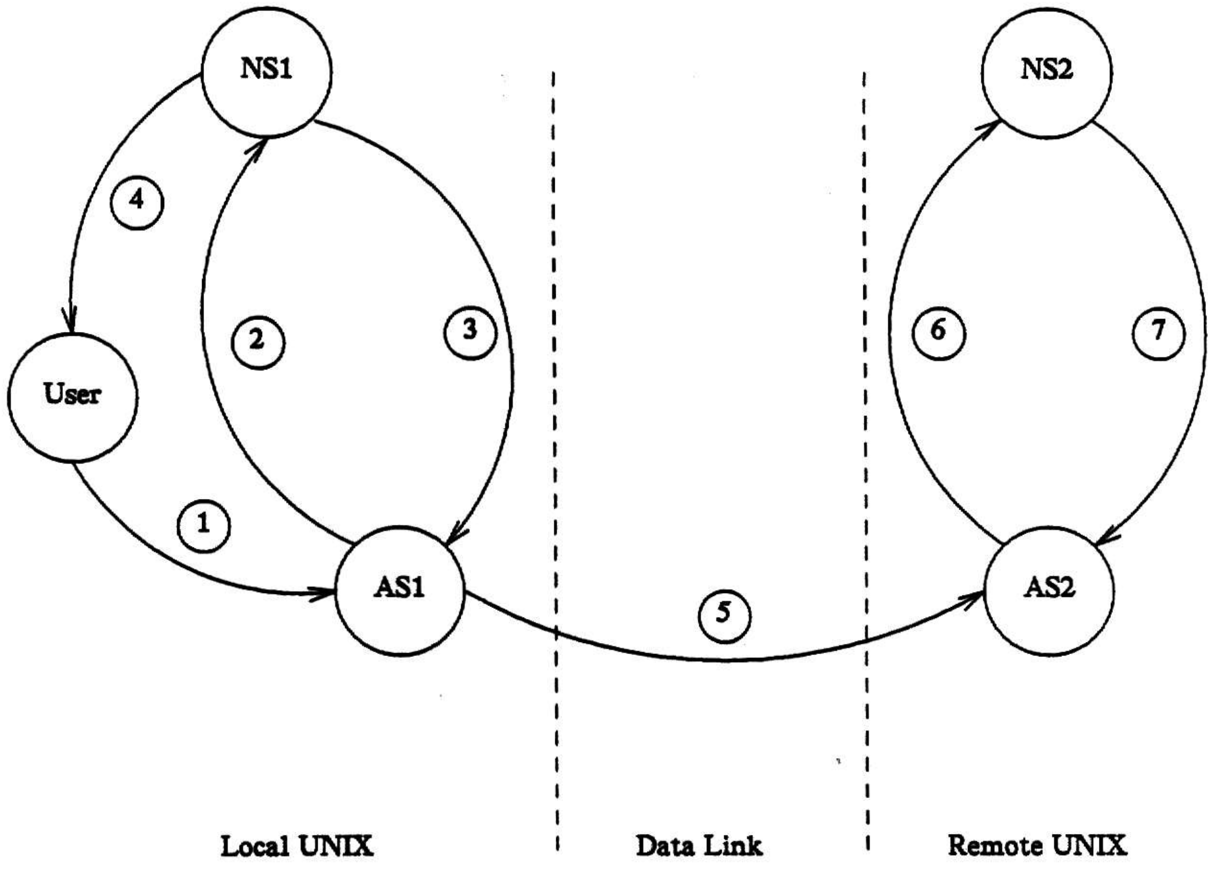
If installed, the secret mail facility is available to anyone who has a login identification. A user only has to enroll into the system to become a member. To enroll in the system, the user types in the *enrollx* command. Before entrance into the system is allowed, the potential member must provide a password that will be encrypted and recorded in the file system. The password will be needed to use any of the other commands available in secret mail.

A feature of secret mail is the *connect* command. The user must specify that a software connection is to be created between himself and another member, the receiver. The receiver can be either a local or a remote member. In either case, both the user and receiver must be enrolled. If the receiver is not enrolled, then the Name Server will inform the authorization server of an invalid request. In addition, the Name Server will display a message onto the user's terminal of why the request cannot be honored. Assuming the receiver is enrolled, the Name Server checks that an active connection does not already exist. If an active connection does not exist, a connection is created and recorded in both member's file system.

At the time the connection is made a unique conversation key for message encryption is created and recorded. If the connection is to a remote computer, the conversation key and the remote computer's authorization key are passed back to the Authorization Server. The Authorization Server then encrypts the conversation key using the remote computer's authorization key. The encrypted conversation key is then sent to the remote computer's Authorization Server. The remote computer's Authorization Server will make a request to its Name Server, where a connection is made in the receiver's file.

Either member can now send messages to one another by using the *sendx* command.

Refer to figure that follows:



1. User issues sendx command to send secret mail.
2. AS1(Authorization Server) request for Conversation Key and the remote's Authorization Key.
3. NS1(Name Server) passes back Keys or invalid request.
4. If an invalid request NS1 displays message to User.
5. Remote: send doubly encrypted message to remote AS2.
6. Remote AS2 calls local NS2 for Authorization key.
7. NS2 returns encryption key.

Figure 1. *SENDX* Data Flow Diagram For Remote Mail

When the request to send a message is made, the Authorization Server prompts the user for his or her password to authenticate the request. The Authorization Server calls the Name Server to return the encrypted password of the user making the request. The Authorization Server will then verify the typed in password. If the password is valid, the Name Server is called again for a request to send a message. The Name Server verifies that the receiver is enrolled and that an active connection exists to the receiver. The Authorization Server will either receive the conversation key and the computer's authorization key (if remote), or an invalid flag is set. For a local send request, the message is only encrypted with the conversation key. The encrypted message is stored in the receiver's secret mailbox. Remote messages are doubly encrypted for security and protection across insecure communication channels. The message is first encrypted with the conversation key, and then the encrypted message is encrypted again with the remote computer's authorization key. The local Authorization Server sends the doubly encrypted message to the remote computer's Authorization Server. The remote computer's Authorization Server receives the message, and decrypts the message using its authorization key. The encrypted message is then stored in the receiver's secret mailbox. The receiver is notified by the system mail facility that someone has sent him or her secret mail.

The *getx* command provides the facility of retrieving secret mail. When the *getx* command is requested, the Authorization Server prompts for the user's password for authentication. If verification is correct then the Authorization Server goes to the user's secret mailbox and retrieves the newest message first. Each message has a clear text header on it. Parts of the header information are passed to the Name Server. The Name Server searches through the user's file for the correct conversation key based on the header information passed, and passes it back to the Authorization Server. Finally, the message is decrypted and displayed on the user's terminal.

The *disconnectx* is another command in secret mail that is available. The command provides the capability to deactivate an active connection if one exists. Once deactivated, the connection with its unique conversation key is no longer available for use in sending messages. The Message Server will remove the connection record when there are no more messages that depend on that key. When a user requests a disconnect from another member, the Authorization Server authenticates the user. The Authorization Server then calls the Name Server for a disconnect request. The user's file is searched, and the connection is deactivated. If the connection cannot be found, the Name Server will flag the request as invalid. If the disconnect is to a member on a remote computer, the local Name Server calls the remote computer's Name Server. In this case, the remote computer's Name Server receives the request and deactivates the connection.

The *editx* command is provided as a facility for the enrolled member who wants to change his or her password.

The *helpx* command provides a short descriptions of the available commands and how to use them.

3.2 DESIGN ASSUMPTIONS

To facilitate the implementation of the project, several design assumptions were made. Several of the operating systems facilities were assumed to be sufficient for providing the foundation for a first time implementation. For example, the "uux" command is used as the mechanism for communicating to a remote computer. Also, the "mail" command is considered acceptable in notifying receivers that they have secret mail.

The emphasis of the project was security, but no measure was taken toward hardware security and protection . It was assumed that computer facilities are under close supervision. Hardware crosstalk problems and wiretapping were assumed non-existant.

3.3 DETAILED DESIGN of the NAME SERVER

3.3.1 Directory Structures

In appendix 2 are diagrams of the directory structure of the file system and the file structure. The directory structure is flexible enough to be placed almost anywhere (which is a characteristic of the UNIX™ operating system), but variable path names have to be set within the "constant.h" file. The directory structure is designed to be flexible. There are four directories, namely, a main directory called secretmail, and three subdirectories. The three subdirectories are USERS, MACHINES, and ERRLOG. The ERRLOG directory is only for recording errors. In the USERS directory are the individual user files. Each file is owned by secret mail with the read and write permission rights set only for secret mail. The same permission rights are on the MACHINES directory files. The MACHINES directory consists of an available machines file and files that list all enrollees on each computer in the network. The available machines file is a list of each computer's system name in the secret mail network and its associated authorization key. The authorization key is used when the Authorization Server sends a message to a remote computer.

3.3.2 Name Server Module

The Name Server consists of a set of functional modules. The modules are:

1.	NS_Request()	(Name Server)
2.	P_Request()	(Password Request)
3.	E_Request()	(Enroll Request)
4.	C_Request()	(Connect Request)
5.	D_Request()	(Disconnect Request)

6.	MS_Request()	(Message Server is called through the Name Server)
7.	H_Request()	(Help Request)
8.	ED_Request()	(Edit Request)
9.	renroll	(the remote process during enrollment)
10.	rdiscon	(the remote process to disconnect)

The NS_Request Server or Name Server selects the proper module, based on the Authorization Server's request, and opens the err_log file for recording errors. The Authorization Server and the Name Server pass all information concerning a user's request by means of a data structure. This structure can be seen in the appendix 1 with definitions of each data element. The Name Server relies on the Authorization Server to provide the necessary information in the data structure so that the right information is retrieved. Once the Name Server is called, it determines what request the Authorization Server is making by means of one of the structure's data elements called "cmd". The Name Server then calls the appropriate module and passes the data structure to it. All of the modules return the data structure back to the Name Server, which returns it back to the Authorization Server. Each module in the Name Server will be discussed separately.

3.3.3 E_Request Module

During an enroll request the Authorization Server passes the data structure to the Name Server. The Name Server checks the data element "cmd" in the structure to determine the type of request. Assuming the request is for enrolling, the Name Server calls the E_Request module or enroll module, and the data structure is passed in the call. The E_Request module retrieves the user's name, which is the user's "login" name from the data structure.

The E_Request module then checks in the USERS directory to see if the user has a file. If the user's file does exists, then the user is already enrolled, and the data structure element "FLAG" set to "INVALID". The data structure is passed back to the Name Server who passes it back to the Authorization Server, who checks the data element "FLAG". In addition, the E_Request module displays the message "*User* Already Enrolled" to the user's terminal. If the user was not enrolled, the request is considered valid. The enroll module creates a file in the USERS directory using the user's "login" name as the file name. The E_Request module takes the encrypted password from the data structure and writes it to the user's file. The user's file is then closed, where closed is the writing of the file to disk.

Next, the file with the enrollee's names under the MACHINES directory is opened, where open is to access a file. The enrollee file uses the computer system name as the file name: this file contains a list of all users for that computer. The system name being the address name that UNIX^{TM} computers use in remote communications. Before opening the enrollee's file, a lock file is created using the computer's system address as the prefix to the file name. If a lock file already exists with the same name, the E_Request module goes to "sleep". "Sleep" is a UNIX^{TM} system call that removes a process from the run state and places it into a dormant state. In the secret mail project, the process sleeps for a two second period. The lock mechanism tries for up to a total of twenty seconds to create a lock file. If it cannot create a lock file in that period of time, an error is recorded, and the data element "FLAG" is set to "INVALID". The E_Request module would return back to the Name Server, which returns back to the Authorization Server. The user's "login" name is appended to the bottom of the file and closed, and the lock file "user_lock" is removed. The E_Request module then opens the available machines file under the MACHINES directory. The available machines file has two columns, the computer addresses that are in the secret mail network and the authorization key. The first field, "computer-address", in the available machines file is read to find the name of each remote computer that needs an updated copy of the list of enrollees. While reading the available machines file, a UNIX™ system "popen" function is used to write to the system "uux" command. The "popen" creates a software communication path between two processes called a pipe. The enrollee file is read and written to the pipe. The "uux" command creates a pipe to a communication channel based on the remote computer's address and executes the process "renroll" to read data from the communication channel. As the E_Request module writes the data from the enrollee file to the pipe, the "uux" function reads the pipe and writes the data to the communication channel pipe, which is read by the "renroll" process on the remote computer. The "renroll" takes the data read from the communication channel and writes it to the sending computer's file name. Each remote computer goes through the same process. The user is unaware of all the activity that transpires. The method used to prevent the user from being aware of the remote communications was to create a "child" process of the enroll process. Where a "child" is a separate process created by a "parent" process, in this case the enroll process. Therefore, the user is not inconvenienced by the transporting of files from one computer to another. If errors occur during the opening of files, a record is made in the ERRLOG directory in the err_logMMDD file. The "MMDD" signifies the month and day.

3.3.4 P_Request Module

The password request or P_Request module takes the data structure that is passed from the Name Server and retrieves the user's "login" name. The user name is used to first verify enrollment. This is done by checking to see if the user's file exists in the USERS directory. If the user's file does exists, then the file is opened and the first string of data in the file is read. This string is the encrypted password that was created during the enrollment session. The retrieved password is stored in the structure under the data element "usr_pswd". The structure is passed back to the Name Server, which, in turn, passes it back to the Authorization Server. The Authorization Server then compares the password in the structure to the password the user typed in.

3.3.5 C_Request Module

As in the previous modules, the Name Server passes the data structure to the C_Request or connect module using the same procedures that were described in the E_Request module. The data structure which is passed, contains the user's data information. The C_Request module first checks that the sending user is enrolled, using the E_Request module's scheme of verification. If the receiving user is also enrolled, then the C_Request module determines if that user's computer address indicates a local or a remote computer. If local, the C_Request module verifies that the receiver is enrolled. Suppose the receiver is enrolled, then the C_Request module checks to see if an active connection exists. Assuming that such a connection does not exist, the C_Request module creates a conversation key based upon the UNIX^{TM} system time. The conversation key is a nine digit number acts as the encryption key used in sending messages between the two users.

A lock file is created before the user's file is opened. The user's file is then opened and a record is created for the connection to the receiver. The record consists of: a status flag (two characters) that signifies either an active or inactive connection, the receiver's computer address, and the conversation key. The receiver's file is also updated with the new connection record. The receiver's record consists of: the status flag, the sender's name, the same conversation key, and the address of the sending computer (in this case the sending and local computer are the same).

If the receiver had been on a remote computer, the local Name Server would not directly be able to add the connection record to the receiver's file. The C_Request module determines if the receiver is on a remote computer. After the sender's connection record has been created, the data structure is passed back to the Name Server with the following additional data elements: the conversation key and the authorization key of the remote computer.

There is a unique authorization key for each computer in the secret mail network. The key is used for encrypting messages that are for remote computers. The data structure is passed back to the Name Server, which, passes it to the Authorization Server. The Authorization Server takes the conversation key from the data structure and encrypts it using the authorization key of the remote computer. The encrypted conversation key is sent to the remote computer's address using the "uux" command. The remote Authorization Server receives the encrypted conversation key and calls its local Name Server with the request for its authorization key. The data structure is passed to the C_Request module, where the request is honored. The Authorization Server receives back the data structure and uses the authorization key to decrypt the encrypted conversation key. The decrypted conversation key is placed into the data structure with the sender's name and the sender's computer address. The C_Request module receives the data structure and writes the connect record into the receiver's file. Now, the sender and the receiver files are consistent.

3.3.6 D_Request Module

The D_Request or disconnect module has the function of deactivating an active connection between two members. The Name Server passes the data structure to the D_Request module. The module retrieves the user's name, the user's computer address, and the receiver's name, and then it verifies that both members are enrolled. If verified, the D_Request module creates a lock file and opens the user's file in the USERS directory. Next, the connection records are scanned, starting at the bottom of the file. While scanning the records, a comparison is made between the receiver's name and the third field of the record, and the receiver's computer address and the second field of the record. If a match in both fields is found, the status flag, first field, of the record is checked to see if the record is active. If the record is active, then the status flag character string is changed to be deactivated using "DL" as the indicators. The characters "DL" stand for disconnect local. The user's file is closed and the lock file is removed.

The D_Request module then checks to see whether the computer's address is a local or remote computer. If local, the receiver's file is changed. In this case, the comparison of the second and third fields of each connection record is made against the user's name and the computer's address. If the computer's address is a remote computer, then the D_Request module does a "popen" to the "uux" command, similar to the enroll module, and executes a process on the remote computer called "rdiscon". The process "rdiscon" is passed the data elements of the receiver's "login" name, the sender's "login" name, and the sender's computer address. Before the local D_Request module passes the data elements of the structure, the data elements of the receiver's name and the user's name are swapped around because the software is designed to act as if the "rdiscon" process is a local disconnect request. Then the same scenario as with the local disconnect request is performed. The connection record in the receiver's file is deactivated. Now, both member's files are consistent with one another. As before, the sender is unaware of all the transactions that took place. The remote processing, again is done as a "child" process similar to the E_Request module. The design is such that either member can request a disconnect.

3.3.7 ED_Request Module

The edit request or ED_Request module provides the capability of changing a user's password. Like the previous modules, the data structure is passed to the edit module. The user is authenicated, and if valid a lock file is created in the USERS directory and the user's file is opened. The first string of characters in the user's file is the encrypted password. The data element of the new password is written over the old password. The user's file is closed and the lock file removed.

3.3.8 H_Request Module

The H_Request or help module provides the user with a short description of the secret mail facility. The user does not have to be enrolled in order to invoke the module.

3.4 DETAIL DESIGN of the MESSAGE SERVER

The MS_Request module or Message Server is considered as a separate server from the Name Server, even though the Name Server treats the Message Server as a module. The assumption, since the Message Server provides a large service of cleaning up deactivated connections and retrieving the proper conversation key, is that it should be classified as a server.

The Message Server only comes into service when a *getx* command is issued from the user. When the Authorization Server has a request from the user to see his or her secret mail, the user's mailbox is opened and the newest message is retrieved. Each message has associated with it a cleartext header at the beginning of the message. The header contains the sender's name and computer address and the time that the message was sent. The header information is retrieved by the Authorization Server and stored in the user's data structure. The Authorization Server passes the structure to the Name Server. The Name Server determines from the data structure element "cmd" that the request is for the Message Server and passes the data structure to it.

When the Message Server is called the following data structure elements are included:

1. The Sender's Name

2. The Address of the Sender's Computer

3. The Receiver's Name (the issuer of the *getx* command)

4. The Time the message was sent.

The time is a nine digit number that depicts the time in seconds based from a reference date on January 1, 1970 [UNIX™ 83].

Before opening the user's file in the USERS directory, a lock file is created in the directory to prevent the Name Server from trying to write to it. The same rules as before, concerning the locking mechanism, apply here. If the lock cannot be created, an error is recorded in the err_log file, and the Message Server returns back to the Name Server with the data element "FLAG" set to "INVALID".

Assuming the lock file is created, the Message Server opens the receiver's file in the USERS directory. Going toward the bottom of the file, the Message Server scans each row or record of data. While scanning, the Message Server looks for a match on the second and third fields of the record with the sender's computer address and sender's name, respectively. When a match is found, the time of the message is compared against the fourth field, the conversation key. If the time (numerically) is greater than or equal to the conversation key, then the proper conversation key has been found and it is placed in the data structure and passed back to the Name Server who passes it back to the Authorization Server. The lock file is removed before the data structure is passed to the Name Server. The Authorization Server then takes the conversation key from the data structure and uses it to decrypt the mail message.

If, for instance, the above match was found and the time of the message was less than the conversation key, then that record in the user's file is not the right one. When this happens the Message Server looks at the first field of the record, i.e., status flag. The status flag may indicate that the record has been deactivated at some previous disconnect session. If the status flag indicates deactivation, the Message Server changes the status flag to the characters "XX" which marks the record for later deletion. The Message Server continues to search until the proper conversation key has been found. Once the conversation key has been found it is passed back to the Authorization Server. If the user has deleted all mail in his mail box the Authorization Server calls the Message Server, again. The Message Server would then search for all records that have been marked for deletion and delete them by copying the user's file to a temporary file minus the records with the status flag as "XX" and then copying the temporary file back. The user's lock file is then removed.

3.5 DETAILED SCENARIO

The Name Server only permits one active connection between any two members at any time. However, there could be several deactivated connections records between the same members. For example, two users are enrolled and user A creates a connection to user B for the first time. User A sends a message to user B and then issues a disconnect to user B, deactivating the first conversation key. Even though user B has not read his mail messages, user A can still request another connection to user B. This is valid since no active connection exists. An active connection would be created then for user A and user B. Such a sequence of events could continue repeatly. When user B does read his mailbox, the Messages Server will select the proper conversation key for each message. Each time the Message Server finds a match in the record against the sender's name and computer address, and the time is less than the conversation key and the status of that connection is deactivated, then the record containing that key in the user's file is marked for deletion. This can be done because messages are read in last-in-first-out order.

The design of the system is such that the file should always be in a sequential order. Since all connect records are appended to a user's file, the conversation key, which represents the system time at creation, will always be increasing in order from the top to the bottom of the user's file. Therefore, the active user's file record will always be below all the deactivated records in the file for the same set of similar members. See the following figures.

MODE	Description
CL:	connection was made on local computer
CR:	connection was made from remote computer
DL:	disconnection was made on local computer
DR:	disconnection was made from remote computer

Figure 2.a. Definitions of Status Flags

[Top of File]			
Status Flag	Machine	Receiver	Conversation Key
CL	ksuvax1	User_e	598789876
DL	ksu832	User_c	498789475
DL	ksu832	User_c	498789675
CR	ksuvax1	User_c	598789990
DL	ksu832	User_c	498789878
CL	ksuvax1	User_d	598789991
DL	ksu832	User_c	498789975
CL	ksuvax1	User_b	598789876
CL	ksu832	User_c	498789999

Figure 2.b. Example: User_a's file

Figure 2.b is an example of a user's file. Especially note that User_a has one active connection to the member (ksu832, User_c) and several deactivated connections. If User_a has been neglecting his or her secret mail, then the example file figure 2.b is possible. Once the mail has been completely read the deactivated connections will be cleaned up. Also, note that the conversation keys are in increasing order from the top of the file to the bottom. Another point that can be made here is as long as the conversation keys for a particular computer stay increasing in order, it does not matter if the conversation keys in general are not. As long as the conversation keys in the user's file are increasing in order, the Message Server will find the proper key.

3.6 RENROLL and RDISCON

Renroll and rdiscon are the processes that read data from a communication channel that is connected to another computer. Both processes are executed by the UNIX™ "uucp" package using the "uux" command. These processes are special functions in the Name Server for remote communication. They do not communicate back to the Authorization Server.

4. SYSTEM COMPARISON

4.1 OVERVIEW

For a comparison to the secret mail project three systems were selected:

1. GRAPEVINE [Birrell 82]

2. CSNET [Solomon 82]

3. UNIX™ "Secure-Channel" Mail Facility [UNIX 83]

The three systems above were selected on their common facility for transferring computer mail between computers. The database and file structures are areas stressed in the sections which follow.

4.2 GRAPEVINE

4.2.1 Overview

GRAPEVINE provides for authentication, message delivery, determining a resource's location, and access control services. The database is distributed and replicated; it took two to three people three years to implement. It is used within the XEROX™ Corporation research and development community.

4.2.2 Design

There is a registration database with information about the users, the computers, the services, the distribution lists, and the access control lists of the system. There are two types of entries in the registration database: group and individual. An entry in this database is referred to as an RNAME. Group entries contain a set of RNAMEs, while an individual entry consists of a password, a list of computer mailboxes, and a user connect site.

The philosophy of GRAPEVINE is different than that of the secret mail project. GRAPEVINE is designed to communicate with different types of computers on an Ethernet network [Metcalf 76]. The design consists of having each user computer communicate to a dedicated GRAPEVINE computer, called a server. Servers share copies of a particular registration database. The intent is that if one server is down then another server can be used. Each user's computer runs what is called the GRAPEVINE-USER package. This package provides the capability to communicate with different servers. If the first server the computer tries is down, then the next one on the list of available servers to the user's computer is tried. This provides for a high probability that users will be able to communicate to at least one server.

Once a server is contacted, it then processes the request to see where to route the message. A server does not necessarily know the final destination of the message but does know of a server that would. All servers share a common directory registry called the GV registry. This registry provides servers with information about all of the servers in the system. In this scheme, each server, which the user's computer knows about, is capable of providing a mailbox. As a result of this, there is a good chance that mail can be sent to at least one of the servers the user's computer knows about.

The secret mail project is not as complex as the GRAPEVINE system. Secret mail is only valid between computers that are running the UNIX^{TM} operating system. GRAPEVINE is flexible enough to communicate to a variety of computer operating systems. Since the secret mail project did not have the requirement of communicating with different types of computers, the design was simpler. Secret mail only communicates directly to the recipient's host computer, there are no servers in between. If the recipient's computer is down, then communication is tried later. As in the GRAPEVINE system, if the user's computer is down the message cannot be retrieved, either. However, in the GRAPEVINE system the sending host computer is no longer concerned with trying to submit the message.

The GRAPEVINE database is similar to secret mail's file system in having individual entries, a password and an address entry. In particular, the GV registry is similar to the secret mail's file of available machines. Secret mail's files that contain all enrollees for a particular computer are also synonymous to the GRAPEVINE servers replication of individual registration databases.

GRAPEVINE and secret mail are similar in the methods of updates and deletion to the database and files. In GRAPEVINE, once a registration server has been requested to change a registry, it ensures that the other servers update their particular copy. Secret mail provides the same type of feature in updating and deleting the connection record in the user's file. This method, however, is not without problems in either system. If two conflicting updates occur simultaneously then, data becomes corrupted. The GRAPEVINE designers planned a partial remedy of prolonged database inconsistencies. The remedy is that during the night, servers with common registries would compare their copies with one another and do merges on the copies to resolve any inconsistencies. Since GRAPEVINE makes use of time-stamping data, the comparison is reliable. Secret mail could have a similar scheme, since time-stamping (the conversation key) is also associated with each record and file.

Both systems require an administrator to supervise the systems, to correct inconsistencies, and to initiate new installations.

One problem, that GRAPEVINE experiences, occurs as a result of a malformed registration database. This situation takes place if the original server creates the error and passes the error to all the other servers that contain the registry. Secret mail has not seen this particular problem, but it is assumed that no matter how well a system is designed, it probably will not be immune to all possible software catastrophes.

4.3 CSNET

4.3.1 Overview

The CSNET project like Grapevine is designed to facilitate computer mail. CSNET is installed among academic computer science departments and other groups doing computer-science research in the United States. It uses the following networks:

1. ARPANET.

2. Telenet.

3. PhoneNet.

4.3.2 Design

CSNET uses a database called the central directory database. The database is located at the University of Wisconsin on the Service Host computer. Communication to the central database can be from remote CSNET member hosts which run CSNET software, other hosts that have the capability of exchanging mail with CSNET member hosts, or directly to the Service Host by phone lines. Normally, the user communicates to the central database through an agent name server on a host member computer. The agent name server formats and sends the requests to the central database. CSNET depends on the sender and receiver host computer mail transport system for the mechanism of transporting messages from source to destination. A user-interface program on the user's computer interacts with the mail transport system. On each CSNET member host resides a set of host tables and on the user's computer a set of local users tables that provide directory information to the central database.

CSNET uses the same style request format for sending mail to another computer as secret mail. Both depend on unique computer names and user names. The mailing address consists of "user-name@host-name". CSNET also provides for a type of "wild-card" addressing on the user's name. An example, if someone was not sure about the spelling of a person's name, then the insertion of the "*" character signifies to match any combination of characters. The system returns with a list, if there is one, of all combinations of the name. The user would then select the proper correct spelling. This capability is not incorporated into the secret mail name server, but it is considered to be a desirable future feature.

The CSNET database is centralized, whereas the secret mail file system is both centralized and decentralized. Like the GRAPEVINE system, CSNET also caters to a variety of computer types which adds a flexibility that secret mail does not have. CSNET uses a monitor program to control concurrency problems. The database structure consists of records that are fixed length, one per entry, with a separate overflow area for long entries. It uses inverted indices, one for each word appearing in the database. A hash table structure is used to speed accesses to the index. On the other hand, secret mail uses a tabular structure of variable length records.

4.4 UNIX SECURE MAIL

4.4.1 Overview

Berkeley's secure mail facility is a simple design which provides for the security of mail on a UNIX environment computer [UNIX 83]. This facility does not provide for remote computer mail. A person has to enroll to participate, and the user unique "login" identification is used in the enrollment. The enrollment service requests for a key to be used in future user authentication. The key is encrypted and maintained in a file using the user's identification as part of the file name. There are commands to retrieve the messages and send messages called xget and xsend, respectively. All commands require that the user type in the key. The key authenticates the users. A public key is used for encrypting and decrypting.

4.4.2 Design

Secure mail and secret mail are similar. The same type of commands are used, also the use of individual files to store the password information and the use of the users name as a prefix to the file name are similar.

However, in secure mail the method of storing only the encrypted password in a file and the creation of individual files for each mail message is not efficient. Also, secure mail does not provide any user authentication beyond requiring the user's password. A public key is used for encrypting and decrypting all users' messages. Further, secure mail makes use of a conversation connection between two users.

Secure mail provides minimum authentication and does not make use of the standard mail facility. If secure mail had made use of the standard mail transport mechanisms, the capability of remote mail could have been implemented.

5. CONCLUSION

5.1 SUMMARY

The secret mail project is an implementation of a computer mail service that provides authentication and data security. The UNIX™ operating system was chosen for the project's environment because of the existing mail transportfacilities and the project team's familiarity with UNIX. The project was designed to be user friendly and not to inconvenience the user with services that were time consuming.

By providing encryption and special unique keys between two users, the project provides security that is not available with the present UNIX™ mail system. Although the encrypted messages could be deciphered by an experienced cryptographer, it would require a large amount of effort to generate the encryption key. Considering the typical user on a UNIX computer, the encryption method using the Data Encryption Standard is satisfactory. The use of a special conversation key between two users provides additional protection, since a new unique key can be created for each message sent.

The capability of remote message transfer is a major feature of secret mail. Basically, any UNIX™ environment computer can connect to the secret mail network with minimum effort. As with most systems, there are always problems and the need for improvements, and secret mail is no exception.

5.2 PROBLEMS and ENHANCEMENTS

An enhancement that is considered to be a good future endeavor is encrypting the file system, which would provide an additional security measure. The file system's present method of protection lies in the access permissions of the files; outside of the secret mail programs, only the owner has permission to do anything to his files The file's owner is considered to be the administrator of secret mail. The administrator's position is not intended to be a full time job, but instead someone who supervises the system periodically and answers questions that users may have.

Generally, a user can do only what the software programs allow. The method for controlling permission rights is done by using the UNIX™ facility of setting a user's process identification (uid) to that of the owner during the execution of the secret mail program. In general, the setting of the user-id-bit on a file works, but there is the possibility that the user of secret mail, through a system error or intentionally, may end up with the "uid" bit still set for the user after the completion of the secret mail program. Therefore, the user would have the same permission rights as the owner. This is an infrequent problem and is not considered serious enough to warrant the current correction. The setting of the user-id-bit is used extensively in the UNIX™ operating system.

An administrative tool would be a useful feature. At present, the administrator has to edit the available machines file for changes, due mainly to the installations of new computers in the secret mail network. These changes would have to be done on each computer in the network. A tool that would perform changes on the local and remote computers would provide a convenience and a less error prone system. For instance, if there are several administrators because of distances between computers, maintaining replicated files in a distributed system can be a potential problem.

Another problem in the project's file system is the need of more mutual consistency and atomicity of a transaction. Mutual consistency and atomicity are goals of secret mail, but the mechanisms used are weak. As in the GRAPEVINE system, the idea of performing a comparison on files at night, during low usage of the system, could help prevent long term inconsistency among the remote computers in the network. This comparison feature is considered as an item to be implemented in a possible future development of secret mail. The main problem with inconsistencies in secret mail is a situation where two members do a connect simultaneously to each other, especially, if the members are on different computers. Since the file system remote mechanism is not as instantaneous as one would like, the above situation could cause multiple active conversation keys to the same member. Due to time constraints on the project's implementation a preventative mechanism was not designed.

The "uux" command is used in remote communications. A more efficient method could be implemented, either by modifying the present use of "uux" or by using a completely new communication link package to be developed. Modifying the present use of the "uux" command would provide a better system that is less error prone and could help maintain the mutual inconsistency of the file system.

There are probably other problems that have yet to surface and enhancements to be considered, but the present implementation is sufficient to use without any major difficulties.

5.3 VALUE of SECRET MAIL

The secret mail facility can be of service to those users that require the need of secure and authenticated message protection. An application might be a situation where a letter that contains sensitive material is being transmitted from one location to another. Normally, the U.S. Postal Service provides this type of letter service through registered mail. The postal service requires the signature of the receiving party on delivery. The only authentication is the mailing address. A questionable procedure with the postal service, though, is that the registry receipt is left in the mailbox until the receiving party comes home. Furthermore, the transferring of a register letter usually takes twenty-hours to deliver.

In secret mail, the same letter can be sent in a few minutes, assuming the communication link is available and the receiver is authenticated. The letter is also doubly encrypted on transferring from source to destination and remains in a single encrypted form in the receiver's secret mailbox until the intended reader requests it.

In perspective, secret mail provides a securer delivery mechanism than the postal service provides. Even in the general use of secret mail, the users usually only want an intended receiver to see the mail; therefore, whether the mail messages are ordinary or sensitive, secret mail can provide a reliable service.

REFERENCES

[Bhargava 82]	Bhargava.Bharat."Resiliency Features of the Optimistic Concurrency Control Approach for Distributed DataBase Systems" IEEE.Reliability in Distributed Software and Database System" 1982
[Birrell 82]	Birrell. Andrew D.,Lerin. R.,Needham. R.M.,Schroeder. M.D. "GRAPEVINE: An Exercise in Distributing Computing" Comm.ACM,(April.1982).Vol.25,Num.4
[Cardenas 79]	Cardenas. Alfonso F.,"Database Management System" Allyn and Bacon,Inc.,1979
[Codd 70]	Codd,E.F.,"A Relational Model of Data for Large Shared Data Banks",Comm.ACM,(June.1970).Vol.13,Num.6
[Eswaran 76]	Eswaran. K.P., et al.,"The Notions of Consistency and Predicate Locks in a Database System",Comm.ACM,(Nov.1976).Vol.19. Num.11,624-633
[Fernandez 81]	Fernandez,E.B.,Summers. R.C., Wood,C.,"Database Security and Integrity",The System Programming Series. Addison-Wesley Publishing Company,1981
[Kernighan 78]	Kernighan. B.K., Ritchie,D.M.,"The C Programming Environment" Printice-Hall,Inc., Englewood Cliffs. New Jersey. 1978
[Lamport 78]	Lamport. L.,"Time, Clocks, and the Ordering of Events in Distributed Systems"Comm.ACM,(July 1978).Vol.21,Num.7,558-564
[Lampson 81]	Lampson. B.W., et al.,(Atomic Transactions)"Distributed Systems-Architecture and Implementation". Springer-Verlag Berlin Heidelberg New York,1981,246-265
[Martin 81]	Martin. J.,"Design and Strategy for Distributed Data Processor" Prentice-Hall. Inc.,Englewood Cliff. New Jersey,1981
[Metcalfe 76]	Metcalfe. R.M.,Boggs. D.R.,"Ethernet: Distributed Packet Switching for Local Computer Networks",Comm.ACM,Vol.19,Num.7. (July 76),395-404
[NBS 77]	National Bureau of Standards,"Data Encryption Standard",1977
[Ritchie 74]	Ritchie. D.M.,Thompson. K.,"The UNIX Time-Sharing System" Comm.ACM,(July 1974).Vol.17,Num.7,365-375
[Solomon 82]	Solomon. M., Landweber. L.H.,Neuhengen. D.,"The CSNET Name Server",Computer Network,(July 1982).Vol.6,Num.3.
[Tanenbaum 81]	Tanenbaum. Andrew S.,"Computer Networks",Prentice-Hall. Inc., Englewood Cliffs New Jersey,1981
[Ullman 82]	Ullman. J.D.,"Principles of Data Base System" Computer Press. Inc.,1982
[UNIX 83]	"UNIX Programmer's Manual",4.2 Berkeley Software Distribution. Virtual VAX-11 Version. Univerity of California. Berkeley. CA.,(August 1983)

Appendix 1

SECRET MAIL DATA STRUCTURE AND DEFINITIONS

DATA STRUCTURE PASSED BETWEEN SERVERS

```
struct USER_INFO {
        int FLAG;
        int cmd;
        char *usr_id;
        char usr_pswd[25];
        char *mach_id;
        char *rem_usr_id;
        char C_key[12];
        char AS_key[25];
        }
```

DATA STRUCTURE DEFINITIONS

1. FLAG - the variable status flag set by the Name Server and the Message Server that defines if a request was "VALID" or "INVALID".
2. cmd - the variable set by the Authorization Server that defines a request for the Name Server. The following are the possible requests that the Authorization Server can make to the Name Server:

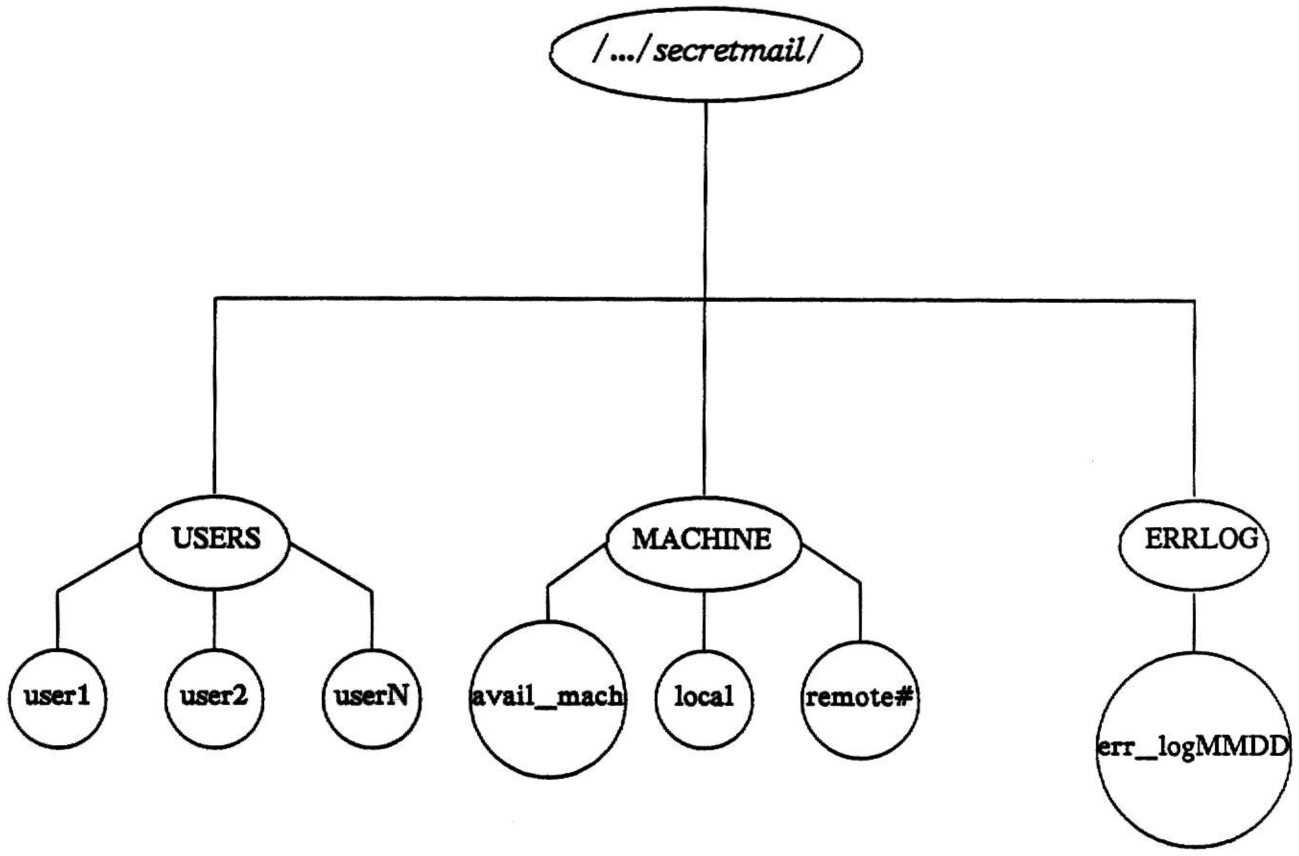
Command	Value
PASSWORD	1
ENROLL	2
CONNECT	3
SEND	4
DISCONNECT	5
REMOTE	6
MSERVER	7
ASKEY	8

3. usr_id - the user's "login" identification
4. usr_pswd - the user's password
5. rem_usr_id - the receiver's "login" name
6. mach_id - the receiver's host computer name or address
7. C_key - the conversation key
8. AS_key - the authorization key

Appendix 2

DIRECTORY AND FILE STRUCTURES

DIRECTORY STRUCTURES



USER FILE STRUCTURE

PASSWORD	NEW LINE

STATUS MODE FLAG	SPACE	RECEIVER'S HOST ADDRESS	SPACE	RECEIVER'S NAME	SPACE	CONVERSATION KEY	NEW LINE

Password - fixed string of 13 characters

Status Mode Flag - fixed string of 2 characters

Receiver's Host Address - variable length string of characters

Receiver's Name - variable length string of characters

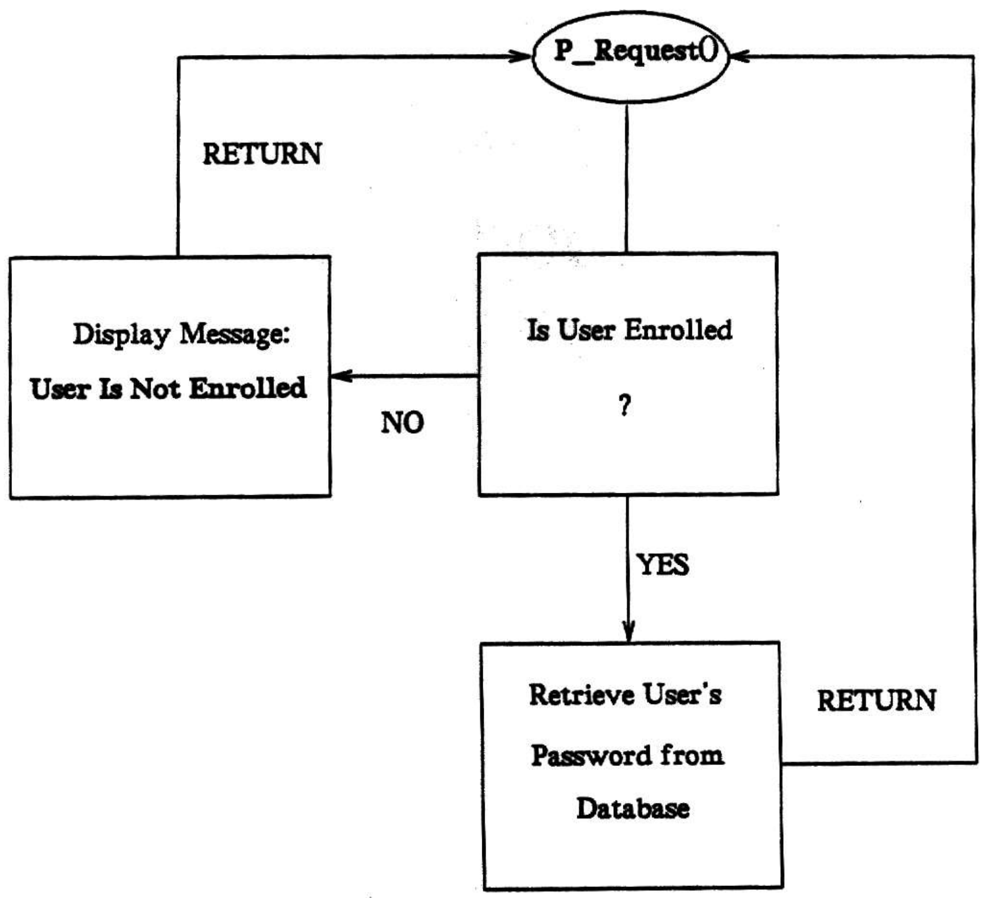
Conversation Key - fixed string of 9 integers

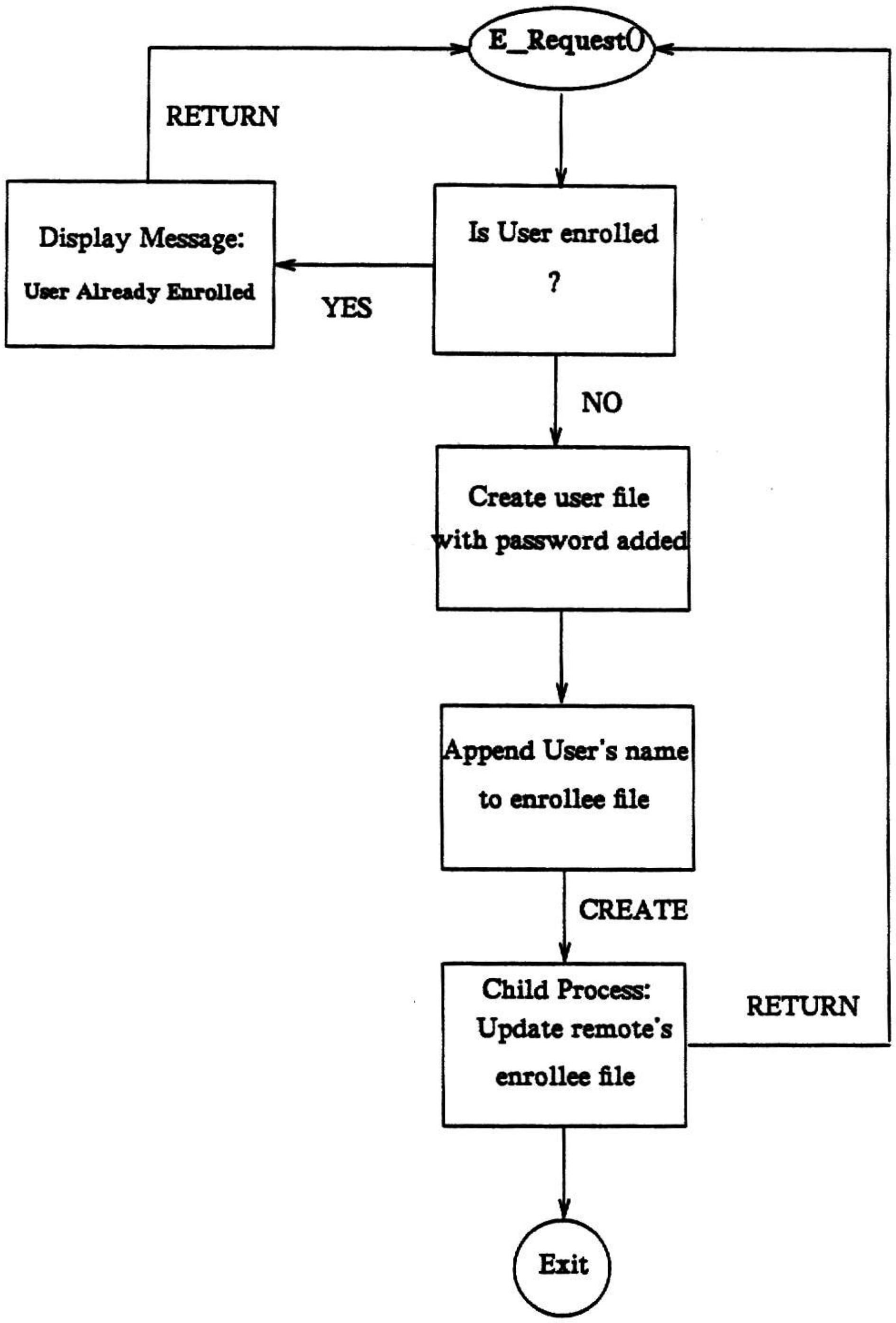
Space - blank character

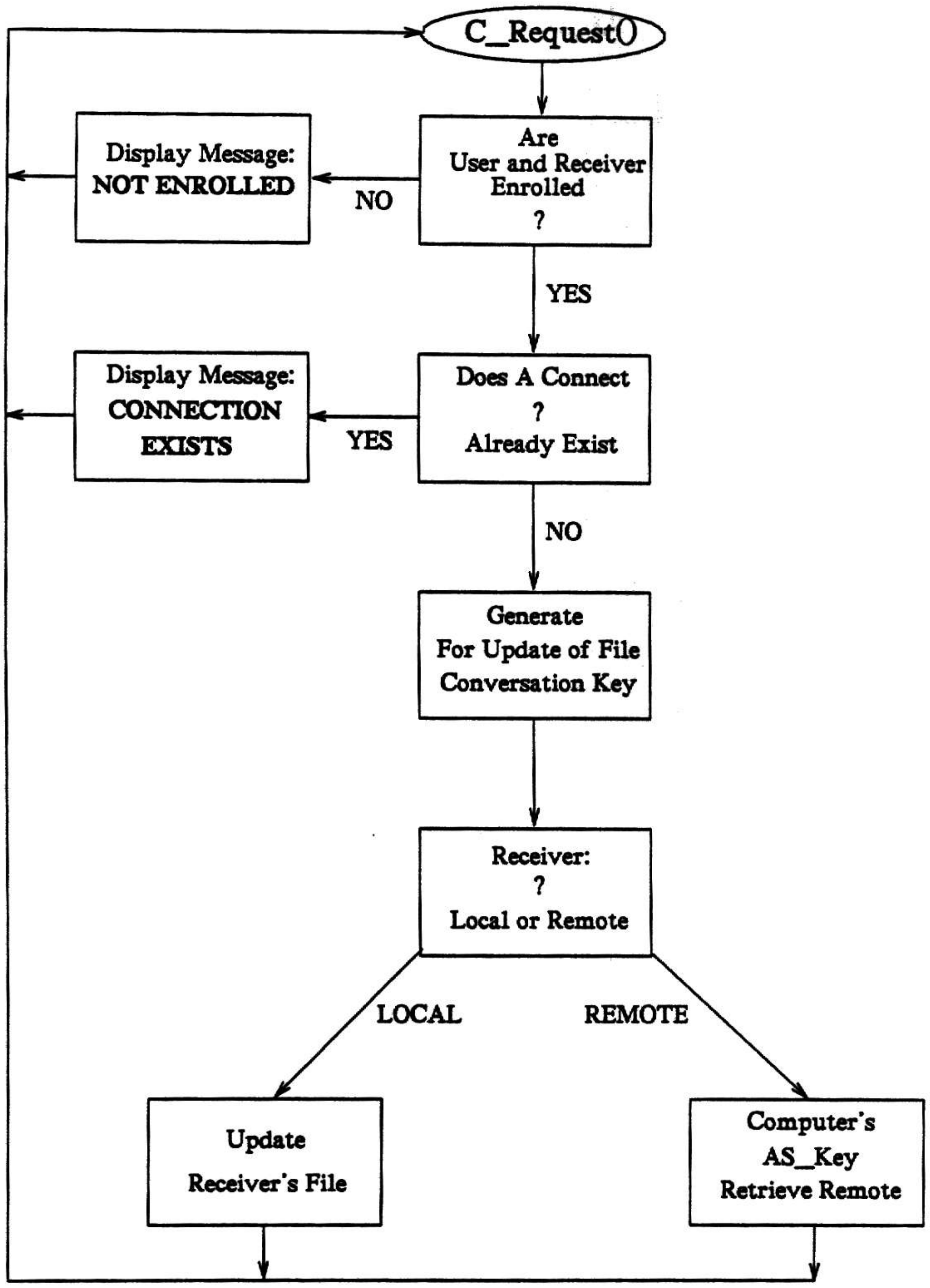
New Line - a new line character

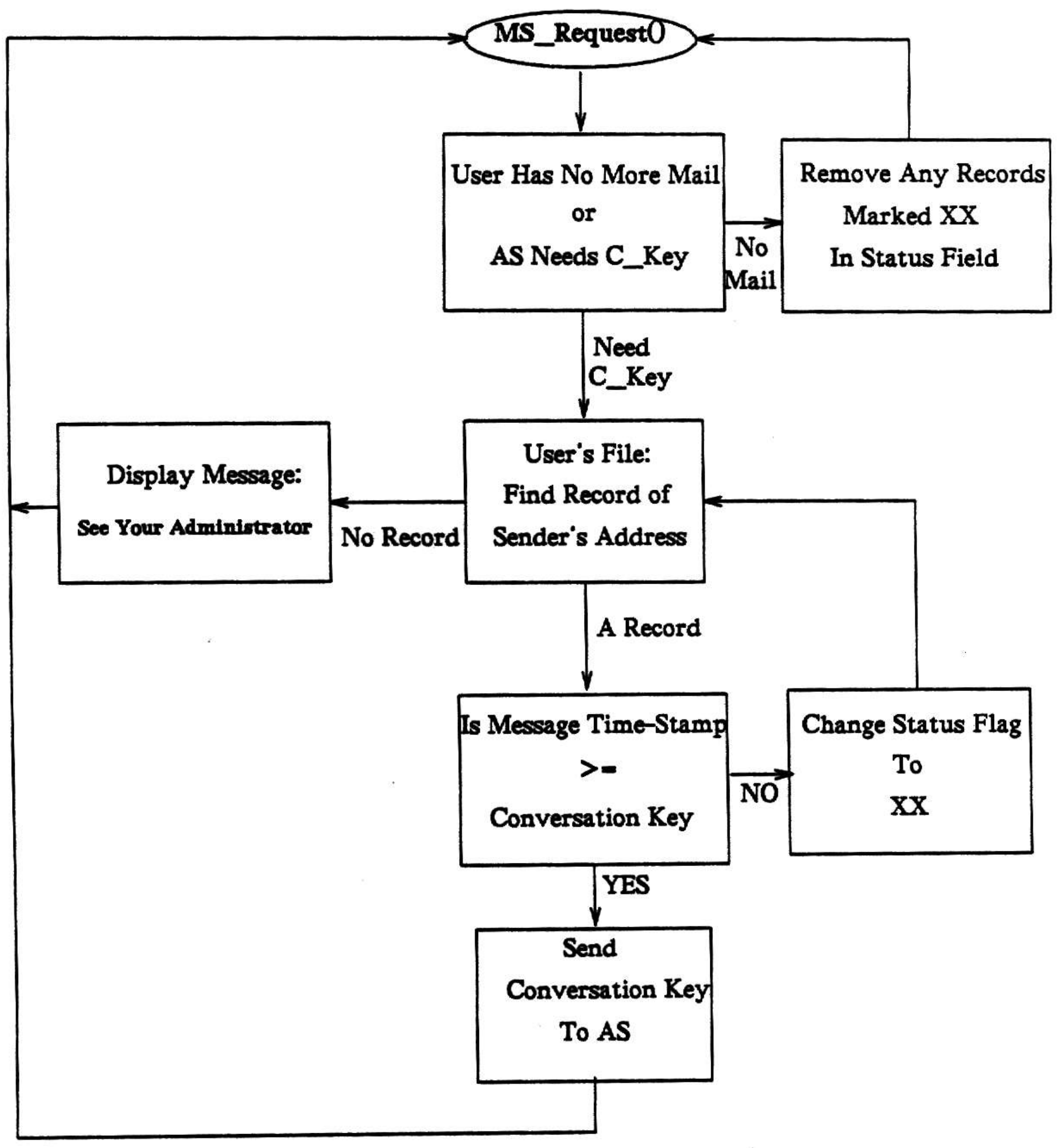
Appendix 3

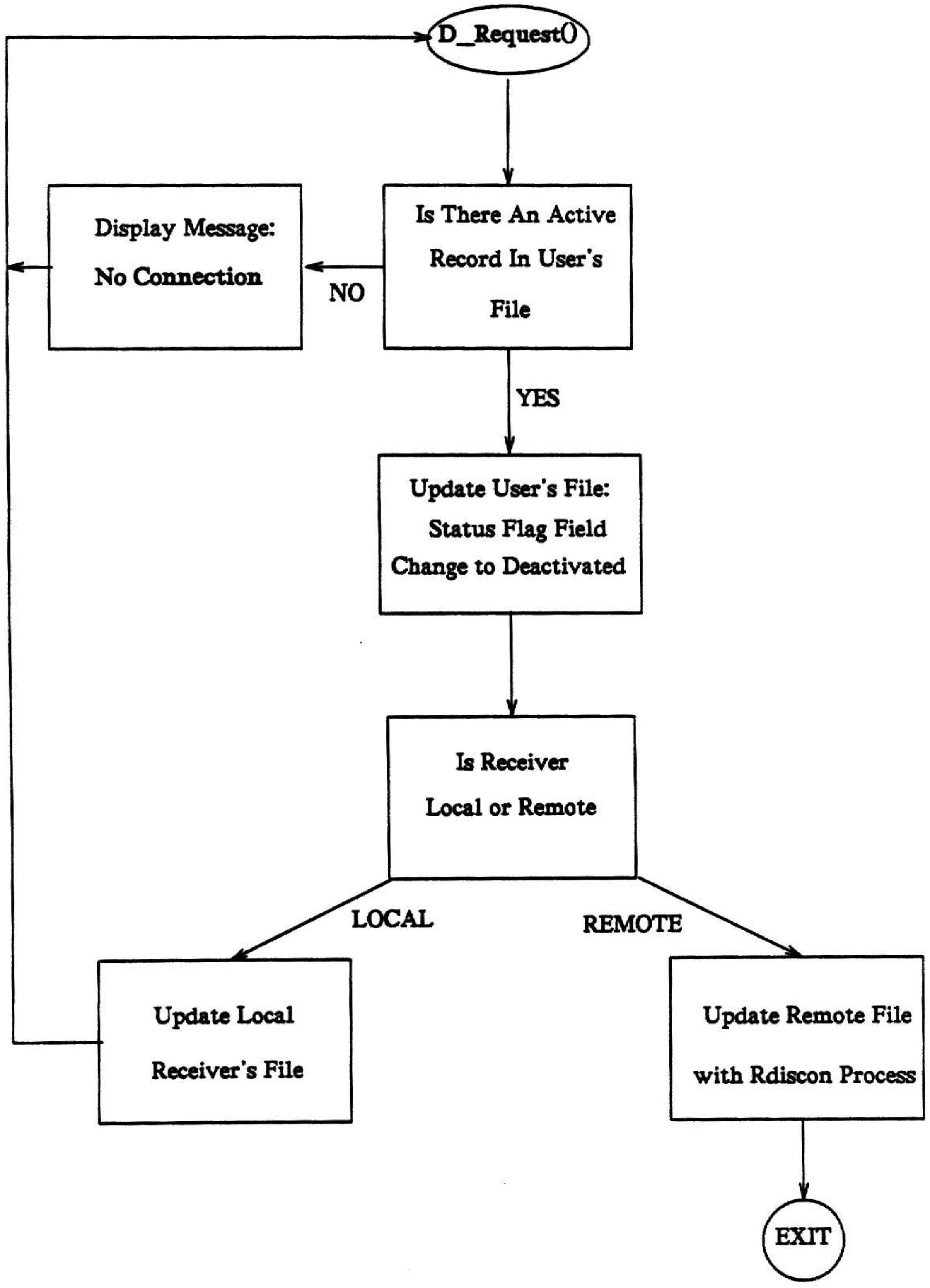
FUNCTIONAL BLOCK DIAGRAMS OF MODULES AND PROCESSES

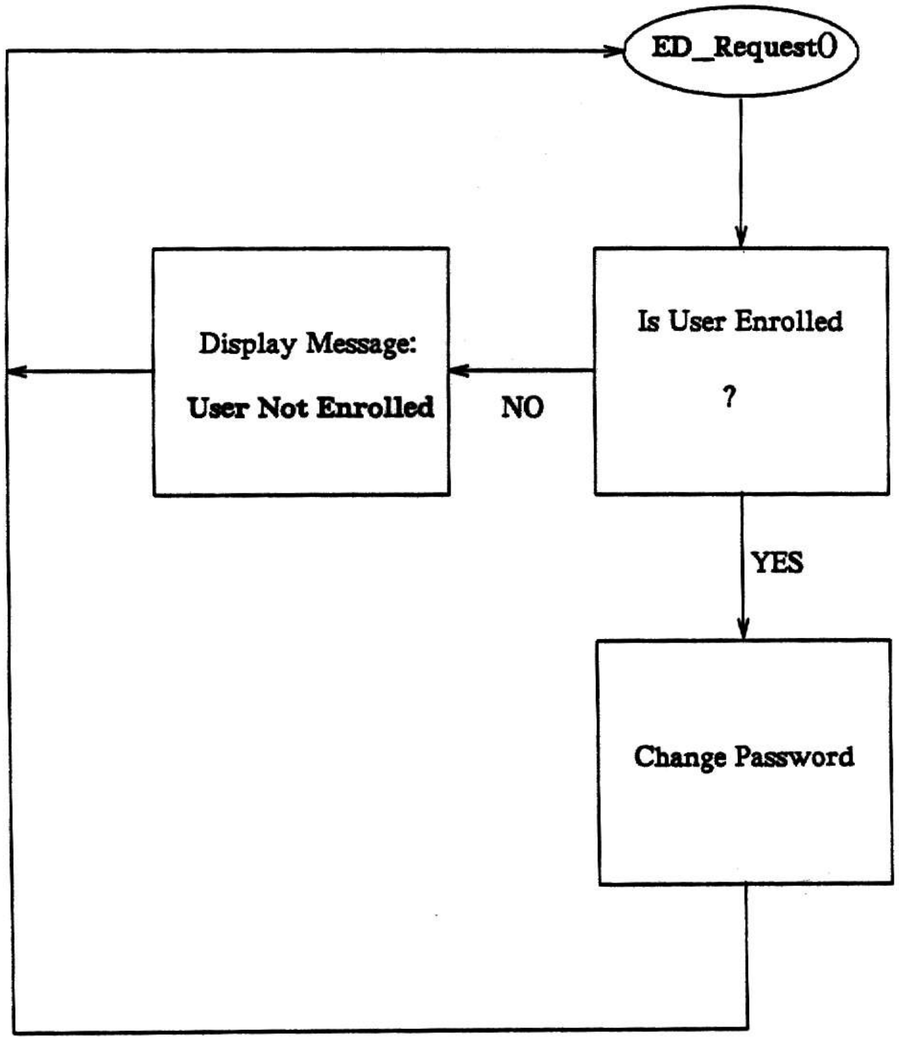

P_Request()
RETURN
Display Message:
User Is Not Enrolled
NO
Is User Enrolled
?
YES
Retrieve User's
Password from
Database
RETURN

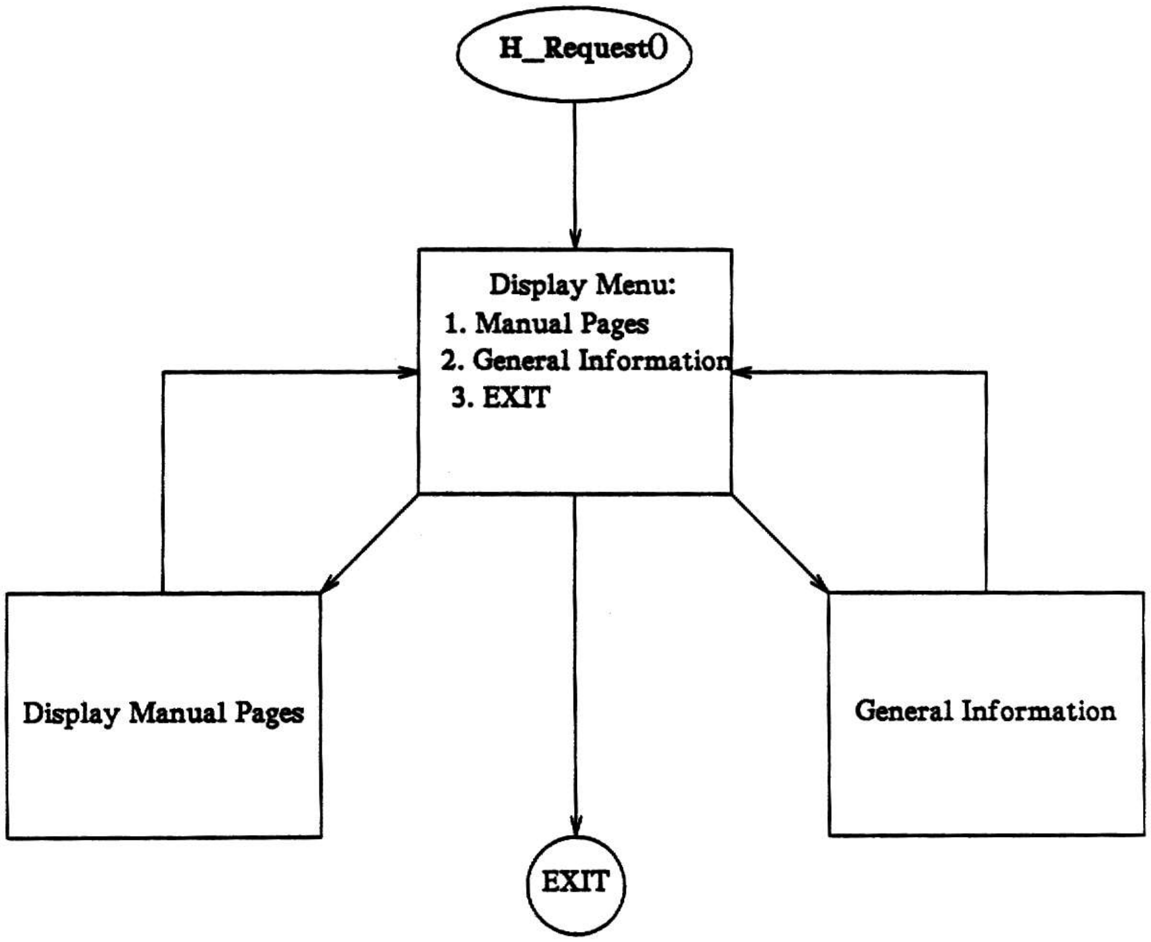

E_Request()
RETURN
Display Message:
User Already Enrolled
YES
Is User enrolled
?
NO
Create user file
with password added
Append User's name
to enrollee file
CREATE
Child Process:
Update remote's
enrollee file
RETURN
Exit

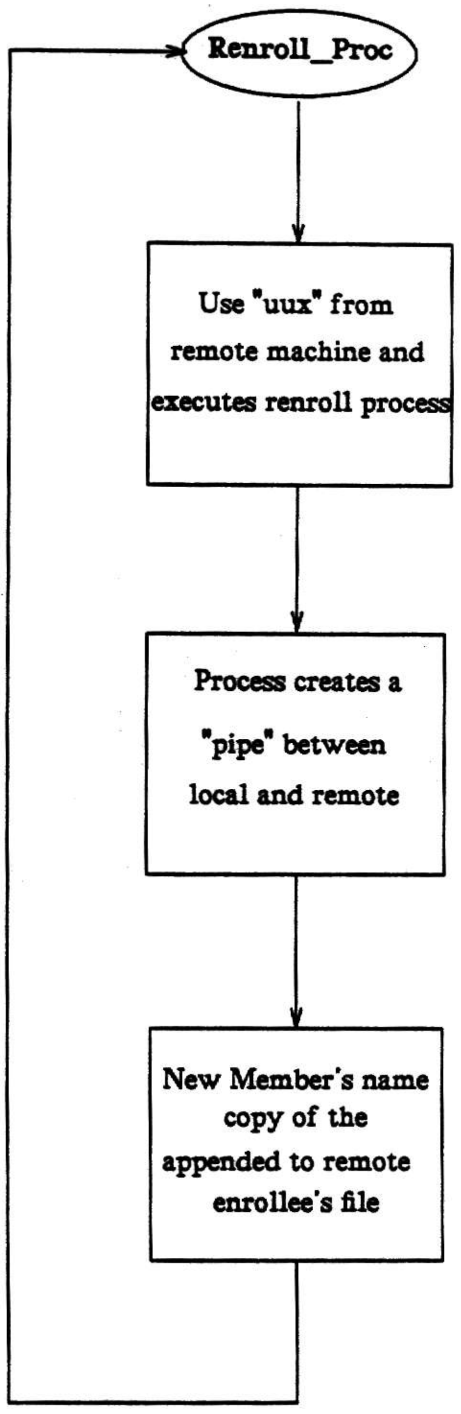

C_Request()
Are User and Receiver Enrolled ?
NO
Display Message: NOT ENROLLED
YES
Does A Connect ? Already Exist
YES
Display Message: CONNECTION EXISTS
NO
Generate For Update of File Conversation Key
Receiver: ? Local or Remote
LOCAL
REMOTE
Update Receiver's File
Computer's AS_Key Retrieve Remote

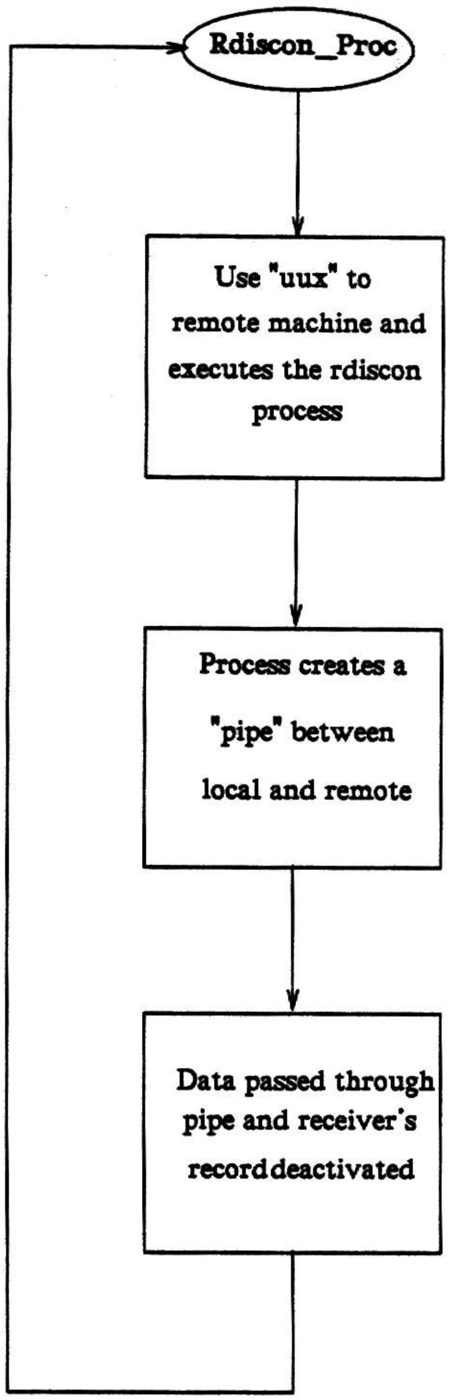

MS_Request()
User Has No More Mail or AS Needs C_Key
No Mail
Remove Any Records Marked XX In Status Field
Need C_Key
User's File: Find Record of Sender's Address
No Record
Display Message: See Your Administrator
A Record
Is Message Time-Stamp >= Conversation Key
NO
Change Status Flag To XX
YES
Send Conversation Key To AS


D_Request()
Is There An Active Record In User's File
NO
Display Message: No Connection
YES
Update User's File: Status Flag Field Change to Deactivated
Is Receiver Local or Remote
LOCAL
REMOTE
Update Local Receiver's File
Update Remote File with Rdiscon Process
EXIT


ED_Request()
Is User Enrolled ?
NO
Display Message:
User Not Enrolled
YES
Change Password


H_Request()
Display Menu:
1. Manual Pages
2. General Information
3. EXIT
Display Manual Pages
General Information
EXIT


Renroll_Proc
Use "uux" from remote machine and executes renroll process
Process creates a "pipe" between local and remote
New Member's name copy of the appended to remote enrollee's file


Rdiscon_Proc
Use "uux" to remote machine and executes the rdiscon process
Process creates a "pipe" between local and remote
Data passed through pipe and receiver's record deactivated

Appendix 4

SOURCE CODE

```
/*NS_Request.c Module - Name Server Request Module */
#include "database.h"
#include <sys/time.h>
#include "constant.h"

extern struct DATABASE *P_Request();
extern struct DATABASE *E_Request();
extern struct DATABASE *C_Request();
extern struct DATABASE *D_Request();
extern struct DATABASE *MS_Request();
extern int lock(), unlock();
extern char *cat();
extern int Rec_err();
extern int err_flag;


extern struct DATABASE *
NS_Request(p_DB)
struct DATABASE *p_DB;
{
        char err[128];

        /*      strcpy(&err[0],"This is a test in the NAME SERVER0);
        **      Rec_err(&err[0],OPEN);
        */

        switch(p_DB->cmd)
        {
                case PASSWORD: p_DB = P_Request(p_DB);
                        break;
                case ENROLL: p_DB = E_Request(p_DB);
                        break;
                case CONNECT: p_DB = C_Request(p_DB);
                        break;
                case SEND: p_DB = C_Request(p_DB);
                        break;
                case DISCONNECT: D_Request(p_DB);
                        break;
                case MSERVER: MS_Request(p_DB);
                        break;
                case EDIT: ED_Request(p_DB);
                        break;
        }

 return(p_DB);
 }
 /*** April, 1986 ***/

/* Connect request verifies that the machine id
 * is a valid machine. If valid a Conversation Key is
 * generated and passed back to the Authorization Server(AS)
 * The C_Key is stored in the user's data file along with the
 * machine and the receiver's id.
 * Remote connects assumes no other connections exist since
 * the sender's database (where connect was issued) has
 * verified that a connection does not already exist.
 * Send request verifies that a connection exist for the user.
 * If true, the user's database struct is loaded with
 * information and passed back to the AS.
 *
 * Author: J. W. Merritt
 *
*/

#include "database.h"
#include "constant.h"

extern int lock(), unlock();
extern char *cat();
extern int Rec_err();
char err[128];

extern struct DATABASE *
C_Request(p_DB)
struct DATABASE *p_DB;
{


  char conn_mach_id[25], rec_usr_id[25], Mode_flag[25];
  /*flag:Connect,Local,Disconnect,Remote*/
  char *modType1 = "CL";
  char *modType2 = "CR";
  char ver_receiver[25];
  char *s_ptr, name[25];
  int chmod(), access(), value, byte, as_key = 0;
  long time();
  long *tloc;
  FILE *FP, *MFP, *EFP;
  struct stat stbuf;
  int gethostname(), enroll_flag = 0;

  if((value = gethostname(name,sizeof(name))) == -1)
  {
    Rec_err("CONNECT: Error on gethostname function\n",OPEN);
    p_DB->FLAG = INVALID;
    return(p_DB);
  }
 if(p_DB->FLAG == ASKEY)
 {
  s_ptr = cat(MACH_DIR,"avail_mach");
  if((FP = fopen(s_ptr,"r")) == NULL)
  {
   Rec_err("CONNECT: Open to avail_mach file\n",OPEN);
   p_DB->FLAG == INVALID;
   return(p_DB);
  }
  else
  {
   while((byte = fscanf(FP,"%s %s\n",
    &conn_mach_id[0],&p_DB->AS_key[0])) != EOF)
   {
     if(strcmp(&name[0],&conn_mach_id[0])==0)
     {
    p_DB->FLAG = VALID;
    fclose(FP);
    return(p_DB);
     }
   }
   p_DB->FLAG = INVALID;
   fclose(FP);
   return(p_DB);
  }
 }

 if(p_DB->FLAG == REMOTE)
 {
  s_ptr = cat(USERS_DIR,p_DB->usr_id);
  lock(s_ptr);
  if((EFP = fopen(s_ptr,"a")) == NULL)
  {
sprintf(&err[0],"CONNECT-REMOTE: Opening of %s's file
      ruserid=%s machid= %s key=%s\n",
    p_DB->usr_id, p_DB->rem_usr_id,p_DB->mach_id,
    p_DB->C_key);
   Rec_err(&err[0],OPEN);
   p_DB->FLAG = INVALID;
   return(p_DB);
  }
  else
  {
   fprintf(EFP,"%s %s %s %s\n","CR",p_DB->mach_id,
    p_DB->rem_usr_id, &p_DB->C_key[0]);
   fflush(EFP);
   fclose(EFP);
   unlock(s_ptr);
   p_DB->FLAG = VALID;
   return(p_DB);
  }
```

```
}

s_ptr = cat(USERS_DIR,p_DB->usr_id);
if((value = access(s_ptr,0)) == -1)
{
 fprintf(stderr," %s Is Not Enrolled \n", p_DB->usr_id);
 p_DB->FLAG = INVALID;
 return(p_DB);
}

if( strcmp(&name[0],p_DB->mach_id) == 0)
{
  s_ptr = cat(USERS_DIR,p_DB->rem_usr_id);
  if((value = access(s_ptr,0)) == -1)
  {
    fprintf(stderr,"Local Receiver %s Is Not Enrolled \n",
   p_DB->rem_usr_id);
    p_DB->FLAG = INVALID;
    return(p_DB);
  }
}
else
{
 s_ptr = cat(MACH_DIR,p_DB->mach_id);
 if(( EFP = fopen(s_ptr,"r")) == NULL)
 {
     fprintf(stderr,"\n%s Not a Valid Machine\n",
                  p_DB->mach_id);
     p_DB->FLAG = INVALID;
     return(p_DB);
 }
 else
 {
 while((byte = fscanf(EFP,"%s\n",&ver_receiver[0])) != EOF)
   {
      if(strcmp(p_DB->rem_usr_id,&ver_receiver[0]) == 0)
      {
    enroll_flag = 1;
    fflush(EFP);
    fclose(EFP);
    break;
      }
   }
   if( enroll_flag != 1)
   {
    fprintf(stderr,"\nRemote Receiver %s Not Enrolled\n",
          p_DB->rem_usr_id);
    p_DB->FLAG = INVALID;
    return(p_DB);
   }
 }
```

```
}
/* Check that remote machine is valid and pick-up AS-key */

 s_ptr = cat(MACH_DIR,"avail_mach");
 if((EFP = fopen(s_ptr,"r")) == NULL)
 {
 Rec_err("CONNECT: Open error on avail_mach's file\n",OPEN);
  exit(0);
 }
 else
 {
  while((byte = fscanf(EFP,"%s %s\n",&conn_mach_id[0],
     &p_DB->AS_key[0])) != EOF)
  {
   if(strcmp(p_DB->mach_id,&conn_mach_id[0]) == 0)
   {
    fclose(EFP);
    fflush(EFP);
    as_key = 1;
    break;
   }
  }
 }
 if(as_key != 1)
 {
 sprintf(&err[0],"CONNECT: Can Not Find machine: %s  AS_key",
  p_DB->mach_id);
  Rec_err(&err[0],OPEN);
  exit(0);
 }

 s_ptr = cat(USERS_DIR,p_DB->usr_id);
 lock(s_ptr);
 if((FP = fopen(s_ptr,"r+")) == NULL)
 {
 sprintf(&err[0],"CONNECT: Unable to READ %s's file\n",
    p_DB->usr_id);
  Rec_err(&err[0],OPEN);
   p_DB->FLAG = INVALID;
   return(p_DB);
 }
 else
 {
  fscanf( FP,"%s\n",&p_DB->usr_pswd[0] );

  while ((byte = fscanf(FP,"%s %s %s %s\n",
        &Mode_flag[0],&conn_mach_id[0],
        &rec_usr_id[0],&p_DB->C_key[0])) != EOF)
    {
     if( (strcmp(p_DB->mach_id,&conn_mach_id[0]) == 0) &&
      (strcmp(p_DB->rem_usr_id,&rec_usr_id[0]) == 0) &&
      ((strcmp(modType1,&Mode_flag[0])==0)||
      (strcmp(modType2,&Mode_flag[0])==0)))
     {
     if(p_DB->cmd == SEND)
     {
      fclose(FP);
      unlock(s_ptr);
      p_DB->FLAG = VALID;
      return(p_DB);
     }
     fprintf(stderr,
     "A Connection For %s To %s!%s Already Exists\n",
      p_DB->usr_id,p_DB->mach_id,
      p_DB->rem_usr_id);
     fflush(FP);
     fclose(FP);
     unlock(s_ptr);
     p_DB->FLAG = INVALID;
     return(p_DB);
     }
      }
       }

if(p_DB->cmd == SEND)
{
 fclose(FP);
 unlock(s_ptr);
 p_DB->FLAG = INVALID;
 fprintf(stderr,"No Active Connection between %s and %s\n",
 p_DB->usr_id,p_DB->rem_usr_id);
 return(p_DB);
}

if ( sprintf(&p_DB->C_key[0],"%ld",time((long *) 0)) < 0 )
{
 Rec_err("CONNECT:Time function not working\n",OPEN);
 strcpy(&p_DB->C_key[0],"CONVERKEY");
}
     fprintf(FP,"CL %s %s %s\n",
  p_DB->mach_id,p_DB->rem_usr_id,&p_DB->C_key[0]);

fflush(FP);
fclose(FP);
unlock(s_ptr);

if(strcmp(&name[0],p_DB->mach_id) == 0)
{
 s_ptr = cat(USERS_DIR,p_DB->rem_usr_id);
 lock(s_ptr);
 if(( FP = fopen(s_ptr,"a")) == NULL)
 {
```

```
 Rec_err("CONNECT: Open error update receiver file\n",OPEN);
      exit(0);
  }
  else
  {
   if(strcmp(p_DB->usr_id,p_DB->rem_usr_id) !=0)
   {
       fprintf(FP,"CL %s %s %s\n",
   p_DB->mach_id,p_DB->usr_id,&p_DB->C_key[0]);
   fflush(FP);
   fclose(FP);
   unlock(s_ptr);
   }
  }
 }
 unlock(s_ptr);
 p_DB->FLAG = VALID;
 return(p_DB);
}
```

```
/* DATA.BASE  April, 1986 */

/* Declaration of the Name Server and Message Server modules
** Also the data structure that is passed between the Name
** Server and the Authorization Server
**/

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <strings.h>
#include <pwd.h>
#include "constant.h"

static struct DATABASE {
  int FLAG;
  int cmd;
  char *usr_id;
  char usr_pswd[25];
  char *mach_id;
  char *rem_usr_id;
  char C_key[12];
  char AS_key[25];
  };


static struct DATABASE *NS_Request();
static struct DATABASE *P_Request();
static struct DATABASE *E_Request();
static struct DATABASE *C_Request();
static struct DATABASE *D_Request();
static struct DATABASE *ED_Request();
static struct DATABASE *MS_Request();
static struct DATABASE DB;

int d_flag = 0;
int err_flag = 0;
```

```
/*    D_Request.c   April, 1986 */

/* The Disconnect module deactivates a connection between the
 * user and the other member.  The connection record is
 * not removed until all mail that is associated to the
 * conversation key has been read.
*/

#include "database.h"
#include "constant.h"

extern int lock(), unlock();
extern char* cat();
extern int Rec_err();

extern struct DATABASE *
D_Request(pp_DB)
struct DATABASE *pp_DB;
{

  char *s_ptr, *cmd, cline[128], name[25], err[128];
  FILE *EFP, *RFP, *popen();
  int value, child;
  struct DATABASE  *Search();

  cmd = &cline[0];
  if((value = gethostname(name,sizeof(name))) == -1)
   Rec_err("DISCONNECT: error on gethostname\n",OPEN);
  s_ptr = cat(USERS_DIR,pp_DB->usr_id);
  lock(s_ptr);
  if((EFP = fopen(s_ptr,"r+")) == NULL)
  {
   sprintf(&err[0],"DISCONNECT: Can not open %s's files\n",
     pp_DB->usr_id);
   Rec_err(&err[0],OPEN);
   exit(0);
  }
  pp_DB = Search(EFP,pp_DB);
  unlock(s_ptr);
  if(pp_DB->FLAG == INVALID)
  {
  fprintf(stderr,"\nNo Active Connection existed for %s to
    %s!%s\n", pp_DB->usr_id,pp_DB->mach_id,pp_DB->rem_usr_id);
   return(pp_DB);
  }

/* If Disconnect is associated with remote machine
 * pipe a "uux" to start up rdiscon program so
 * remote receiver's connection record will be deactivated
 * If local disconnect deactivation is handled here.
*/
```

```
 if(strcmp(&name[0],pp_DB->mach_id) == 0)
 {
  s_ptr = cat(USERS_DIR,pp_DB->rem_usr_id);
  lock(s_ptr);
  if((EFP = fopen(s_ptr,"r+")) == NULL)
  {
   sprintf(&err[0],"DISCONNECT: Error on users file  %s\n",
    s_ptr);
   Rec_err(&err[0],OPEN);
   exit(0);
  }
  pp_DB->rem_usr_id = pp_DB->usr_id;
  pp_DB = Search(EFP,pp_DB);
 }
 else
 {

/* Fork a child process to handle uux request to remote machine */

  child = fork();
  if(child == 0)
  {
   sprintf(cmd,"/usr/bin/uux - %s!rdiscon %s %s %s",
   pp_DB->mach_id,pp_DB->usr_id,pp_DB->rem_usr_id,
   &name[0]);
   if((RFP = popen(cmd,"w")) == NULL)
   {
    Rec_err("DISCONNECT: Popen pipe error\n",OPEN);
   }
   fflush(RFP);
   pclose(RFP);
   exit(0);
  }
 }
 unlock(s_ptr);
 pp_DB->FLAG = VALID;
 return(pp_DB);
}

/*  The Search function goes into the a user's file
 *  and properly lines up the file pointer so fscanf,
 *  for reading, is possible. It starts at the bottom
 *  of file for faster accesses since the most recent
 *  active record is near the bottom of the file.
 *
*/

struct DATABASE *
Search(SFP, pp_DB)
struct DATABASE *pp_DB;
FILE *SFP;
```

```
{

 int Backup();
 int byte;
 int BUP0 = 15, BUP1 = 19;
 char *mode1 = "CL";
 char *mode2 = "CR";
 char mode_type[25], rem_usr_id[25], conn_mach_id[25];
 char c, *cp;

 cp = &c;
 Backup(cp,SFP,-201,2);
 while((byte = fscanf(SFP,"%s %s %s %s\n",&mode_type[0],
 &conn_mach_id[0],&rem_usr_id[0],&pp_DB->C_key[0])) != EOF)
 {

 /*fprintf(stderr,"%s %s %s %s\n",&mode_type[0],
 **&conn_mach_id[0], &rem_usr_id[0],&pp_DB->C_key[0]);
 **/

  if((strcmp(pp_DB->mach_id,&conn_mach_id[0]) == 0) &&
  (strcmp(pp_DB->rem_usr_id,&rem_usr_id[0]) == 0) &&
  ((strcmp(&mode_type[0],mode1) == 0) ||
    (strcmp(&mode_type[0],mode2) == 0 )))
  {
  byte = BUP0 + strlen(&conn_mach_id[0]) +
                              strlen(&rem_usr_id[0]);
   fseek(SFP,-(long)byte,1);  /*line ptr to status flag*/
   fprintf(SFP,"%s","DL");  /* change status flag */
   fflush(SFP);
   fclose(SFP);
   pp_DB->FLAG = VALID;
   return(pp_DB);
  }
  else
  {
  byte = BUP1 + strlen(&conn_mach_id[0]) +
                              strlen(&rem_usr_id[0]);
   Backup(cp,SFP,-(long)byte,1);
  }
 }
 pp_DB->FLAG = INVALID;
 return(pp_DB);
}

/* Works in conjunction with the Search function.
 * Once a record is read, the pointer is moved up
 * the file to the next new line found.
*/
```

```
int
Backup(bcp,B_FP,offset,where)
char *bcp;
FILE *B_FP;
long offset;
int where;
{
 int byte;
 char c;

 bcp = &c;
 fseek(B_FP,offset,where); /* Move beyond new-line char */
 while((byte = fread(bcp,1,1,B_FP)) == 1)
 {

  if( c == '\n')  /* back up to next new-line char */
  {
   return;
  }
  fseek(B_FP,-2l,1);
 }
 return;
}
```

```
/* ED_Request.c  April, 1986 */
/* This module changes the user's old password
** to the new password.  Works with editx
** program.
**/

#include "database.h"
#include "constant.h"

extern struct DATABASE DB;
extern struct DATABASE *p_DB;
extern char *cat();
extern int Rec_err();

extern struct DATABASE *
ED_Request(p_DB)
struct DATABASE *p_DB;
{
 char err[128];
 char *s_ptr;
 int chmod(), value;
 FILE *FP;
 struct stat stbuf;

 s_ptr = cat(USERS_DIR,p_DB->usr_id);

 if((FP = fopen(s_ptr,"r+")) == NULL)
 {
 sprintf(&err[0],"Unable to READ file for %s\n",
   p_DB->usr_id);
   Rec_err(&err[0],OPEN);
   p_DB->FLAG = INVALID;
   return(p_DB);
 }
 else
 {
   fprintf(FP,"%s\n", &p_DB->usr_pswd[0]);
   fflush(FP);
   fclose(FP);
   p_DB->FLAG = VALID;
   return(p_DB);
 }
}
```

```
/* E_Request.c  April, 1986 */
/* The enroll module creates and checks that the user
** is not already enrolled in the system.
*/

#include "database.h"
#include "constant.h"
#include <fcntl.h>

extern char *cat();
extern int lock(), unlock();
extern int Rec_err();

extern struct DATABASE *
E_Request(p_DB)
struct DATABASE *p_DB;
{
 char file_buf[128], IN_file[128], rem_mach[128],
    line[BUFSIZ];
 char *s_ptr, *cmd, cline[128], name[25], err[128];
 int chmod(), value, gethostname(), byte, access(),
   child;
 FILE *FP, *MFP, *RFP, *popen();
 struct stat stbuf;


 cmd = &cline[0];
 s_ptr = cat(USERS_DIR,p_DB->usr_id);
 if((value = access(s_ptr,0)) == 0)
 {
  fprintf(stderr," %s Already Enrolled \n", p_DB->usr_id);
  return(p_DB);
 }
 else
 {

  if((FP = fopen(s_ptr,"w")) == NULL)
  {
   p_DB->FLAG = INVALID;
  sprintf(&err[0],"\nENROLL: Unable to CREATE Enroll file
   for %s\n", p_DB->usr_id);
   Rec_err(&err[0],OPEN);
   return(p_DB);
  }
  else
  {
   fprintf(FP,"%s\n", &p_DB->usr_pswd[0]);
   fflush(FP);
   fclose(FP);
   if(( value = gethostname(name, sizeof(name))) == -1)
     Rec_err("ENROLL: Error on gethostname\n", OPEN);
  chmod(s_ptr,0600);


  s_ptr = cat(MACH_DIR,&name[0]);
  lock(s_ptr);
  if(( FP = fopen(s_ptr,"a")) == NULL)
  {
   sprintf(&err[0],"Can't open Enroll file %s file \n",
     &file_buf[0]);
   Rec_err(&err[0],OPEN);
     unlock(s_ptr);
     exit(0);
  }
  else
  {
   fprintf(FP,"%s\n",p_DB->usr_id);
   fflush(FP);
   fclose(FP);
   unlock(s_ptr);
  }
s_ptr = cat(MACH_DIR,"avail_mach");
if((FP = fopen(s_ptr,"r")) == NULL)
{
Rec_err("ENROLL: Can not open avail_mach file\n", OPEN);
  p_DB->FLAG = INVALID;
  return(p_DB);
}
else
{
  s_ptr = cat(MACH_DIR,&name[0]);
  strcpy(&file_buf[0],s_ptr);
  child = fork();
  if(child == 0)
  {
  while((byte = fscanf(FP,"%s%*s\n",&rem_mach[0])) != EOF)
  {
   if(strcmp(&name[0],&rem_mach[0]) == 0)
     continue;

  sprintf(cmd,"/usr/bin/uux - %s!renroll %s",
    &rem_mach[0],&name[0]);
  if((RFP = popen(cmd,"w")) == NULL)
  {
   Rec_err("ENROLL: Pipe error\n",OPEN);
  }
  s_ptr = cat(MACH_DIR,&name[0]);
  lock(s_ptr);
  if((MFP = fopen(s_ptr,"r")) == NULL)
  {
   Rec_err("ENROLL: Can't open file for pipe\n",OPEN);
   exit(0);
```

```
    }
  while((byte = fread(&line[0],sizeof(&line[0]),
                  BUFSIZ,MFP)) != 0)
    {
      byte=fwrite(&line[0],strlen(&line[0]),byte,RFP);
      fflush(RFP);
      continue;
    }
      unlock(s_ptr);
      fflush(MFP);
      fclose(MFP);
      fflush(RFP);
      pclose(RFP);
    }
      fflush(FP);
      fclose(FP);
    }
    if(child == 0)
    {
      exit(0);
    }
    p_DB->FLAG = VALID;
    return(p_DB);
   }
   }
  }
 }
```

```
/*  MS_Request.c  April, 1986  */
/*  The message server module, retrieves the conversation
**  key for a particular mail message and also marks and
**  removes deactivated records when the AS reports no more
**  mail is left the user's mail box.
*/

#include "database.h"
#include "constant.h"

extern int lock(), unlock();
extern char *cat();
extern int Rec_err();

char err[128];

extern struct DATABASE *
MS_Request(pp_DB)
struct DATABASE *pp_DB;
{

  char *s_ptr, name[25];
  FILE *EFP;
  int value;
  int gethostname();
  struct DATABASE *MSearch();

  if((value = gethostname(name, sizeof(name))) == -1)
  Rec_err("error on gethostname\n",OPEN);

  /* Bld PATH string to user file*/

  s_ptr = cat(USERS_DIR,pp_DB->usr_id);
  lock(s_ptr); /*  Create user lock file for protection */
  /* open user file to read */

  if((EFP = fopen(s_ptr,"r+")) == NULL)
  {
   sprintf(&err[0],"Can not open %s's database\n",
     pp_DB->usr_id);
   Rec_err(&err[0],OPEN);
   exit(0);
  }
  pp_DB = MSearch(EFP,pp_DB,s_ptr);
  unlock(s_ptr);  /* remove user lock file */
  return(pp_DB);
 }

 struct DATABASE *
 MSearch(SFP, pp_DB,ptr)
```

```
struct DATABASE *pp_DB;
FILE *SFP;
char *ptr;
{

 extern int d_flag;
 int MBackup();
 int byte;
 char *mode3 = "DL";
 char *mode4 = "DR";
 char mode[25], remusrid[25], filemach[25];
 char file_key[25], line[BUFSIZ];
 char c, *cp, ts_ptr[128];
 FILE *TFP, *NFP;

  /** copy user's file to user_tmp file if
  **  there are no more messages and there are
  **  deactivated records in file
  */

 if( pp_DB->FLAG == DELETE )
 {
  fprintf(stderr,"FLAG = %d  d_flag = %d\n",
        pp_DB->FLAG,d_flag);
  if(d_flag == 0)
  {
   pp_DB->FLAG = VALID;
   return(pp_DB);
  }
  strcpy(&ts_ptr[0],USERS_DIR);
  strcat(ts_ptr,pp_DB->usr_id);
  strcat(ts_ptr,"_tmp");
  if((TFP = fopen(&ts_ptr[0],"a")) == NULL)
  {
   Rec_err("MSERVER: TFP error\n",OPEN);
  }
  else
  {
   fseek(SFP,01,0); /* beginning */
  }

  byte = fscanf(SFP,"%s\n",&pp_DB->usr_pswd[0]);

  fprintf(TFP,"%s\n",&pp_DB->usr_pswd[0]);

  while((byte = fscanf(SFP,"%s %s %s %s\n",&mode[0],
    &filemach[0],&remusrid[0],&file_key[0])) != EOF)
  {
   if((strcmp(mode3,&mode[0]) == 0) ||
     (strcmp(mode4,&mode[0]) == 0))
   {
    continue;
   }
      fprintf(TFP,"%s %s %s %s\n",&mode[0],
          &filemach[0],&remusrid[0],&file_key[0]);
   fflush(TFP);
   continue;
 }
      fclose(SFP);  /* close user file */
      unlink(ptr);  /* remove old usr file */
  fflush(TFP);
  fclose(TFP);
  if((TFP = fopen(&ts_ptr[0],"r")) == NULL)
  {
   Rec_err("MSERVER: Error on TFP read\n", OPEN);
   exit(0);
  }
 if((NFP = fopen(ptr,"a")) == NULL) /* create new usr file */
  {
   Rec_err("MSERVER: Error on file change\n",OPEN);
   exit(0);
  }
  chmod(ptr,0600);
 while((byte=fread(&line[0],sizeof(&line[0]),
              BUFSIZ,TFP)) != 0)
  {
    fprintf(NFP,"%s",&line[0]);
    fflush(NFP);
    continue;
  }
  fflush(TFP);
  fclose(TFP);
  fflush(NFP);
  fclose(NFP);
  unlink(&ts_ptr[0]);
  pp_DB->FLAG = VALID;
  return(pp_DB);
} /* end first if statement */

cp = &c;

/* move file ptr to last record in file */

MBackup(cp,SFP,-201,2);

/*  read user's file one record  */

while((byte = fscanf(SFP,"%s %s %s %s\n",&mode[0],
  &filemach[0],&remusrid[0],&file_key[0])) != EOF)
{

  /*** is record deactivated ***/
```

```
       if((strcmp(&mode[0],mode3) == 0)
      || (strcmp(&mode[0],mode4) == 0))
       {
        d_flag++;
       }

   /* check record to see if machine!remote_user match */

      if((strcmp(pp_DB->mach_id,&filemach[0]) == 0) &&
      (strcmp(pp_DB->rem_usr_id,&remusrid[0]) == 0))
      {

/*fprintf(stderr,"file_key = %91d\n",atol(&file_key[0]));*/


   /* type cast character into long type and check time
    * stamp against conversation key.
    *
   */

if( (long) atol(&pp_DB->C_key[0]) >= (long) atol(&file_key[0]) )
   {
     strcpy(&pp_DB->C_key[0],&file_key[0]);

     byte = 19 + strlen(&filemach[0]) + strlen(&remusrid[0]);
     MBackup(cp,SFP,-(long)byte,1);

     while((byte = fscanf(SFP,"%s %s %s %s\n",&mode[0],
      &filemach[0],&remusrid[0],&file_key[0])) != EOF)
    {

        if((strcmp(&mode[0],mode3) == 0)
       || (strcmp(&mode[0],mode4) == 0))
        {
         d_flag++;
       break;
        }

     byte = 19 + strlen(&filemach[0]) + strlen(&remusrid[0]);
     MBackup(cp,SFP,-(long)byte,1);
     continue;

    } /* end while above */

     fprintf(stderr,"d_flag = %d\n", d_flag);
     pp_DB->FLAG = VALID;
     return(pp_DB);
      }
     else
      {
      byte = 19 + strlen(&filemach[0]) + strlen(&remusrid[0]);
```

```
        MBackup(cp,SFP,-(long)byte,1);
        continue;

/** end of if statement on comparison of key and timestamp **/

        }
  }
  else
  {   /* back up one record */
   byte = 19 + strlen(&filemach[0]) + strlen(&remusrid[0]);
   MBackup(cp,SFP,-(long)byte,1);
  } /* end of if statment on comparing sender's address */
  } /* End of While */

  pp_DB->FLAG = INVALID;
  fprintf(stderr,"Could Not Find C_key for %s to %s!%s\n",
      pp_DB->usr_id,pp_DB->mach_id,pp_DB->rem_usr_id);
  sprintf(&err[0],"Could Not Find C_key for %s to %s!%s\n",
      pp_DB->usr_id,pp_DB->mach_id,pp_DB->rem_usr_id);
  d_flag = 0;
  Rec_err(&err[0],OPEN);
  fflush(SFP);
  fflush(SFP);
  fclose(SFP);
  return(pp_DB);
}

int
MBackup(bcp,B_FP,offset,where)
char *bcp;
FILE *B_FP;
long offset;
int where;
{
  int byte;
  char c;
  bcp = &c;
  fseek(B_FP,offset,where); /* Move beyond new-line char */
  while((byte = fread(bcp,1,1,B_FP)) == 1)
  {
    if( c == '\n')   /* back up to next new-line char */
    {
      return;
    }
    fseek(B_FP,-21,1);
  }
  return;
}
```

```
/* P_Request.c  April, 1986 */
/* This module retrieves the user's password */

#include "database.h"
#include "constant.h"

extern struct DATABASE DB;
extern struct DATABASE *p_DB;
extern char *cat();
extern int Rec_err();
extern struct DATABASE *

P_Request(p_DB)
struct DATABASE *p_DB;
{
 char err[128];
 char *s_ptr;
 int chmod(), value;
 FILE *FP;
 struct stat stbuf;

 s_ptr = cat(USERS_DIR,p_DB->usr_id);
 if((value = access(s_ptr,0)) == -1)
 {
  fprintf(stderr," %s Is Not Enrolled \n", p_DB->usr_id);
  p_DB->FLAG = INVALID;
  return(p_DB);
 }
 else
 {
  if((FP = fopen(s_ptr,"r")) == NULL)
  {
  sprintf(&err[0],"Unable to READ file for %s\n",
        p_DB->usr_id);
  Rec_err(&err[0],OPEN);
  p_DB->FLAG = INVALID;
  return(p_DB);
  }
  else
  {
   fscanf(FP,"%s", &p_DB->usr_pswd[0]);
   fclose(FP);
   p_DB->FLAG = VALID;
   return(p_DB);
  }
 }
}
```

```
/** Rec_err() April, 1986 **/
/*  This module records errors detected within
**  the Name Server.  Mostly software errors
**  on the opening and closing of files.
*/

#include <stdio.h>
#include <sys/time.h>
#include "constant.h"

extern int lock(), unlock();
extern char *cat();
extern int err_flag;

extern int
Rec_err(sptr,flag)
char *sptr;
int flag;
{
long time(), clock;
struct tm *localtime();
struct tm *tmptr;
char *ptr, errtime[10];
char errfile[256];
FILE *ERRFILE;

if(flag == OPEN)
{
  ptr = &errfile[0];
  time(&clock);
  tmptr = localtime(&clock);
  strcpy(&errfile[0],USERS_DIR);
  strcat(&errfile[0],"../ERRLOG/err_log");
  sprintf(&errtime[0],"%d.%d",tmptr->tm_mon+1,tmptr->tm_mday);
  strcat(&errfile[0],&errtime[0]);
  if((ERRFILE = fopen(ptr,"a")) == NULL)
    fprintf(stderr,"Can not open errlog file\n");
  fprintf(ERRFILE,"\n%s\n",sptr);
  fflush(ERRFILE);
  fclose(ERRFILE);
}
return;
}
```

```
/* cat.c April. 1986 */
/* This module is used to combine strings */

#include <strings.h>

extern char *
cat(p1,p2)
char *p1, *p2;
{

 static char *string;
 static file[128];

 string = strcpy(&file[0],p1);
 string = strcat(file,p2);
 return(string);
}

/*  constant.h  April. 1986 */
/* Keywords passed between AS and NS */
#define VALID  1
#define  INVALID  0
#define  PASSWORD   1
#define ENROLL  2
#define  CONNECT  3
#define  SEND  4
#define  DISCONNECT  5
#define REMOTE  6
#define MSERVER   7
#define ASKEY  8
#define DELETE  9
#define EDIT  10
#define OPEN 1
#define CLOSE 2
#define MACH_DIR  "/usrb/att/chance/smail/MACHINES/"
#define USERS_DIR  "/usrb/att/chance/smail/USERS/"
#define MAILDIR    "/usrb/att/chance/smail/mail/"
#define TMPFILE    "/usrb/att/chance/smail/tmp/maXXXXXX"
```

```
/* database.h  April, 1986 */
/* data structure passed between AS and NS
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <strings.h>
#include <pwd.h>

extern struct DATABASE {
 int FLAG;
 int cmd;
 char *usr_id;
 char usr_pswd[25];
 char *mach_id;
 char *rem_usr_id;
 char C_key[12];
 char AS_key[25];
 };
extern struct DATABASE *p_DB;
extern struct DATABASE DB;
static int lock();
static int unlock();
static char *cat();
static int Rec_err();

/* editx.c  April, 1986
 *
 * This command allows a user to change his password in the
 * secret mail service. It requires a login id and
 * a passwd that will have to be qouted each time an amail
 * service cmd is invoked. The input info is passed to the
 * Name Server which creates the entry in the registration
 * database.
 */

#include "DATA.BASE" /* user registration database structure */

int getuid();
char *getpass();
char *getlogin();
char *crypt();
struct passwd *getpwuid();

main(argc,argv)
char **argv;
int argc;
{
  struct DATABASE *p_DB;
  char saltc[2];
  char *myname, *mypasswd;
  char *pwd, opwbuf[25], *pw;
```

```
long salt;
int good, i, c, uid, cmd;

p_DB = &DB;
p_DB->usr_id = getlogin();
if(p_DB->usr_id == NULL) {
  uid = getuid();
  p_DB->usr_id = getpwuid(uid)->pw_name;
}
p_DB->cmd = PASSWORD;
NS_Request(p_DB);
if (p_DB->FLAG == INVALID) {
  fprintf(stderr, "connect: database access error");
  exit(1);
}
strcpy(opwbuf, getpass("Enter old password: "));
pwd = p_DB->usr_pswd;
pw = crypt(opwbuf, pwd);
if(strcmp(pw, pwd) != 0) {
  fprintf(stderr, "Sorry.\n");
  exit(1);
}

/*
 * The following code generates a random number
 * by adding the long integer time with the process
 * id.  The number is "processed" and used as the key for
 * the pertubation of the "DES" algorithm within the "crypt"
 * subroutine call which is used to "hide" the user passwd
 *
 */

time(&salt);
salt += getpid();
saltc[0] = salt & 077;
saltc[1] = (salt >> 6) & 077;
for (i=0; i<2; i++) {
  c = saltc[i] + '.';
  if (c>'9') c += 7; if (c>'Z') c += 6;
  saltc[i] = c;
}

mypasswd = crypt((getpass("ENTER new password: ")),saltc);
strcpy(p_DB->usr_pswd, mypasswd);

/*
 * The Name Server is called to enter the user in the
 * registration database.
 */

p_DB->cmd = EDIT;
```

```
  p_DB = ED_Request(p_DB);
  if(p_DB->FLAG == VALID ) {
    printf("You now have a new password\n");
    exit(0);
  }
  else {
  fprintf(stderr, "enroll: registration database error.\n");
  exit(1);
  }
}

/**** helpx.c  March 25, 1986 ****/
/* This command in secret mail is a help message
** consisting of general information  and manual
** page on commands.
*/
/******J.W. Merritt******/

#include <stdio.h>
#include <curses.h>
#include <signal.h>
#define screen 5

char *menu[screen] = {
"\n Help Information on Secret Mail\n",
"\n Type in Item number\n\n",
"1. General Information about Secret-Mail\n",
"2. Manual Pages of Commands\n",
"3. EXIT\n"
    };
int flag;
main()
{
  int cnt, i;
  char *HELP;
  char *s, term[BUFSIZ];
  FILE *FP;
  void int catch();
  void int catchint();

  signal(SIGHUP,catch);
  signal(SIGINT,catchint);
  signal(SIGQUIT,catch);
  initscr();
  for(;;)
  {
  clear();
  refresh();
  s = &term[0];
  for(cnt = 0; cnt < screen; cnt++)
  {
   fprintf(stderr,"%s",menu[cnt]);
  }
  if(gets(s) == NULL)
   exit(9);
  i= atoi(s);
  switch (i)
  {
   case 1: HELP="/usrb/att/chance/smail/general";
     break;
   case 2: HELP="/usrb/att/chance/smail/manual";
     break;
   case 3: exit(0);
  }

  if((FP = fopen(HELP,"r")) == NULL)
  {
   fprintf(stderr," can not open help file\n");
   exit(0);
  }
  clear();
  refresh();
  flag=1;
   while( (flag && fgets(s,BUFSIZ,FP) != NULL))
   {
    fprintf(stderr,"%s",s);
    if (cnt++ >= 17)
    {
     fprintf(stderr,"Hit Return Key to Continue\n");
     gets(s);
     cnt = 0;
     clear();
     refresh();
    }
   }
  }
}
catch()
{
 fprintf(stdout,"Caught Interrupt Signal - BYE!\n");
 exit(0);
}
catchint()
{
 flag=0;
}
```

```
/* lock.c April, 1986 */
/* This module prevents more than one user writing
** to the same file.
*/

#include <stdio.h>
#include <strings.h>

extern int
lock(file)
char *file;
{
 FILE *LFP;
 int i, value;
 char file_lock[128], *p_lock;

   p_lock = &file_lock[0];
   strcpy(file_lock,file);
   strcat(file_lock,"_lock");

   for(i=0;i < 10; i++)
   {
    if(( value = access(p_lock,0)) == -1)
    {
     if((LFP =fopen( p_lock, "a+" )) != NULL)
     {
       fclose(LFP);
       return;
     }
     else
       fprintf(stderr,"can not create lock file\n");
    }
    sleep(2);
   }
   fprintf(stderr,"Timed out to create %s file\n",p_lock);
   return;
}

unlock(file)
char *file;
{
  char file_unlock[128], *p_unlock;

  p_unlock = &file_unlock[0];
  strcpy(file_unlock,file);
  strcat(file_unlock,"_lock");
  unlink(p_unlock);
  return;
}
```

```
#
#The makefile for compiling the Secret Mail software
#
CC= cc
HOME=/usrb/att/chance

all: enroll connect renroll sendx getx rsend rcon rdiscon
        disconnect editx

install: enroll connect renroll sendx getx rsend rcon rdiscon
        disconnect editx

enroll: enroll.c name.a
  ${CC}  enroll.c name.a -o enroll
  chmod 4755 enroll
  cp enroll ../bin/enrollx

editx: editx.c name.a
  ${CC}  editx.c name.a -o editx
  chmod 4755 editx
  cp editx ../bin/editx

sendx: sendx.c name.a libcpc.a
  ${CC} sendx.c name.a libcpc.a -o sendx
  chmod 4755 sendx
  cp sendx ../bin

getx: getx.c name.a libcpc.a
  ${CC} getx.c name.a libcpc.a -o getx
  chmod 4755 getx
  cp getx ../bin

connect: connect.c name.a libcpc.a
  ${CC} connect.c name.a libcpc.a  -o connect
  chmod 4755 connect
  cp connect ../bin

disconnect: disconnect.c name.a libcpc.a
  ${CC} disconnect.c name.a libcpc.a  -o disconnect
  chmod 4755 disconnect
  cp disconnect ../bin

rsend: rsendx.c libcpc.a name.a
  ${CC} rsendx.c libcpc.a name.a -o rsend
  chmod 4755 rsend
  cp rsend ${HOME}/bin

rcon: rcon.c libcpc.a name.a
  ${CC} rcon.c libcpc.a name.a -o rcon
  chmod 4755 rcon
  cp rcon ${HOME}/bin
```

```
renroll: renroll.c name.a
 ${CC} renroll.c name.a -o renroll
 chmod 4755 renroll
 cp renroll ${HOME}/bin

rdiscon: rdiscon.c name.a
 ${CC} rdiscon.c name.a -o rdiscon
 chmod 4755 rdiscon
 cp rdiscon ${HOME}/bin

name.a: P_Request.o E_Request.o C_Request.o NS_Request.o
        cat.o lock.o D_Request.o MS_Request.o Rec_err.o
        ED_Request.o
 ar rv name.a $?
 ranlib name.a

NS_Request.o P_Request.o E_Request.o D_Request.o C_Request.o
        MS_Request.o Rec_err.o ED_Request.o:
                         database.h constant.h

libcpc.a: getem.o des.o
 ar rv libcpc.a $?
 ranlib libcpc.a
```

```
/*  rdiscon.c  April, 1986 */
/* This module removes deactivated records on the remote
** computer.
*/

#include "DATA.BASE"
#include "constant.h"

extern int lock();
extern char *cat();
extern int Rec_err();

char err[128];

main(argc,argv)
int argc;
char *argv[];
{

  struct DATABASE *DD_Request();
  struct DATABASE *p_DB;
  p_DB = &DB;

  p_DB->usr_id = argv[1];
  p_DB->rem_usr_id = argv[2];
  p_DB->mach_id = argv[3];
  p_DB = DD_Request(p_DB);
```

```
/*if( p_DB->FLAG == VALID && p_DB->cmd > ENROLL)
fprintf(stderr,"\nThe structure for %s is:\nPSWD: %s\n
     MACH_ID: %s\n R_USR: %s\nCK:  %s\nASKey: %s\n",
   p_DB->usr_id,&p_DB->usr_pswd[0],p_DB->mach_id,
   p_DB->rem_usr_id,&p_DB->C_key[0],
   &p_DB->AS_key[0]);
*/
exit(0);
}

struct DATABASE *
DD_Request(pp_DB)
struct DATABASE *pp_DB;
{

 int BUP = 19;
 char *s_ptr;
 FILE *EFP;
 char *cat();
 int lock(), unlock(), value;
 struct DATABASE *Search();
 char name[25];

 if((value = gethostname(name,sizeof(name))) == -1)
   Rec_err("RDISCON: error on gethostname()\n",OPEN);
 s_ptr = cat(USERS_DIR,pp_DB->usr_id);
 lock(s_ptr);
 if((EFP = fopen(s_ptr,"r+")) == NULL)
 {
   sprintf(&err[0],"RDISCON: Can not open %s's database\n",
     pp_DB->usr_id);
   Rec_err(&err[0],OPEN);
   exit(0);
 }
 pp_DB = Search(EFP,pp_DB);
 if(pp_DB->FLAG == INVALID)
   return(pp_DB);
 unlock(s_ptr);
 if(strcmp(&name[0],pp_DB->mach_id) == 0)
 {
   s_ptr = cat(USERS_DIR,pp_DB->rem_usr_id);
   lock(s_ptr);
   if((EFP = fopen(s_ptr,"r+")) == NULL)
   {
     sprintf(&err[0],"RDISCON: Error on users file  %s\n",
       s_ptr);
     Rec_err(&err[0],OPEN);
     exit(0);
   }
   pp_DB->rem_usr_id = pp_DB->usr_id;
   pp_DB = Search(EFP,pp_DB);
 }

 unlock(s_ptr);
 return(pp_DB);
}

struct DATABASE *
Search(SFP, pp_DB)
struct DATABASE *pp_DB;
FILE *SFP;
{

 void int Backup();
 int byte, BUP = 19;
 char *mode1 = "CL";
 char *mode2 = "CR";
 char mode_type[25], rem_usr_id[25], conn_mach_id[25];
 char c, *cp;

 cp = &c;
 Backup(cp,SFP,-201,2);
 while((byte = fscanf(SFP,"%s %s %s %s\n",&mode_type[0],
  &conn_mach_id[0],&rem_usr_id[0],&pp_DB->C_key[0])) != EOF)
 {
  if((strcmp(pp_DB->mach_id,&conn_mach_id[0]) == 0) &&
  (strcmp(pp_DB->rem_usr_id,&rem_usr_id[0]) == 0) &&
  ((strcmp(&mode_type[0],mode1) == 0) ||
    (strcmp(&mode_type[0],mode2) == 0 )))
  {
byte = 15 + strlen(&conn_mach_id[0]) + strlen(&rem_usr_id[0]);
    fseek(SFP,-(long)byte,1);
    fprintf(SFP,"%s","DR");
    fflush(SFP);
    fclose(SFP);
    pp_DB->FLAG = VALID;
    return(pp_DB);
  }
  else
  {
byte = BUP + strlen(&conn_mach_id[0]) + strlen(&rem_usr_id[0]);
    Backup(cp,SFP,-(long)byte,1);
  }
 }
 pp_DB->FLAG = INVALID;
 fflush(SFP);
 fclose(SFP);
 return(pp_DB);
}

void int
Backup(bcp,B_FP,offset,where)
```

```
char *bcp;
FILE *B_FP;
long offset;
int where;
{
  int byte;
  char c;

  bcp = &c;
  fseek(B_FP,offset,where);/* Move beyond new-line char */
  while((byte = fread(bcp,1,1,B_FP)) == 1)
  {

    if( c == '\n') /* back up to next new-line char */
    {
      return;
    }
    fseek(B_FP,-2l,1);
  }
  return;
}
```

```
 /* renroll.c   April, 1986 */
/* This program undates all remote computers enrollment
** files.
*/
#include "DATA.BASE"

extern char *cat();
extern int lock();
extern int unlock();
extern int Rec_err();


char err[128];

main(argc,argv)
int argc;
char *argv[];
{

  char line[128], *s_ptr;
  FILE *EFP;
  struct passwd *getpwuid();
  struct passwd *p;

  p = getpwuid(geteuid());
  s_ptr = cat(MACH_DIR,argv[1]);
  lock(s_ptr);
  if((EFP = fopen(s_ptr,"w+")) == NULL)
  {
    Rec_err("RENROLL: Remote enroll file error\n",OPEN);
    exit(0);
  }

  chmod(s_ptr,0600);
  chown(s_ptr,p->pw_uid,p->pw_gid);
  while(fgets(&line[0],sizeof(line),stdin) != NULL)
  {
    fprintf(EFP,"%s",&line[0]);
  }
  fflush(EFP);
  fclose(EFP);
  unlock(s_ptr);
  exit(0);
}
```

DISTRIBUTED FILE SYSTEMS IN AN AUTHENTICATION SYSTEM

by

John W. Merritt

B. S. E. E., University of South Carolina 1979

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

Master of Science

Department of Computer Science

Kansas State University
Manhattan, Kansas

1986

1430-89
CD-68

An Abstract of the Master's Report

The following is an abstract of the secret mail project. The designs of secret mail are to authenticate and provide security and protection of computer messages between UNIX(TM) computers with a minimum of effort.

The philosophy of secret mail is that messages of a sensitive nature should be protected from outside probing or interception. Secret mail is designed to provide this protection with single encryption for local messages and double encryption during the remote transfer of messages. The messages remain encrypted until the intended receiver requests delivery. The user's data file, which contains information pertinent to the encrypting and decrypting of messages, is protected from outside scanning and corruption. Software mechanisms are provided within the file system to ensure data integrity, and the file system was organized so that the user would experience minimum response times.

The secret mail project consists of the three functional areas: the Authorization Server, the Name Server, and the Message Server. The Authorization Server provides the user interface, the authentication of the user, and message encryption and decryption. The Name Server provides the management of the file system and the generation of a unique conversation key. The conversation key is used for encrypting and decrypting messages. The Message Server provides retrieval and decryption of messages that have been sent. The main emphasis of this paper is on the Name Server and the Message Server's interactions with the file system.