FINFS: A FLEXIBLE INTERNET NETWORK FILE SYSTEM

BY

JAYATHIRTHA MOJNIDAR

Bachelor of Engineering

University of Mysore

Mandya, India

1987

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December 1993

FINFS: A FLEXIBLE INTERNET

NETWORK FILE SYSTEM

Thesis Approved :

[illegible]

Thesis Adviser

[illegible]

[illegible]

[illegible]

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to Dr. Neilsen for his continuous guidance throughout the course of my research. His constant support and encouragement has helped me in fulfilling my objective. His valuable suggestions at various points in time have helped me make speedy progress.

I wish to thank Dr. George and Dr. Benjamin, for agreeing to be in my committee and for their constructive criticisms. They were highly cooperative and understanding. They patiently lent their ears to all my discussions.

My parents have been a driving force and a spirit of encouragement throughout my academic studies and other endeavors. I take this opportunity to thank them. I would like to extend my deepest appreciation to a few special friends, Sridhar, Rosy, Nat, John, Sam, Suresh, Shiva, Ravi, who have been a stimulating support and best companions in all my endeavors. I also wish to thank all other friends and people who made my life beautiful and happy.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

Introduction

The need for information is increasing at an explosive rate. In order to increase the availability of information, geographically dispersed information systems should be integrated. In addition to providing information, information systems have the responsibility of collecting, storing, retrieving and distributing the information. Information systems are expected to provide a secure, reliable and efficient environment to manage the information. In a computerized environment, information systems are known as file systems. The Internet provides an integrated environment of geographically dispersed file systems. Systems that manages these geographically dispersed file systems are called *distributed file systems.*

Information can be stored and distributed using two different methods. In the first method, information is stored at one place called a *dedicated server.* This method has some disadvantages. Dedicated servers can become performance bottlenecks and the information will no longer be available if the server fails. In the second method, information is stored in the same file systems in which it is created. This overcomes most of the disadvantages of the first method. This type of file system is called a *network file system.*

In this paper, we present a prototype of a basic network file system called FINFS. We make use of some concepts from the NFS [12, 21] model for FINFS. FINFS provides user mobility and file mobility. In FINFS, access to remote files is given to users on client machines by users on a server machine. FINFS is flexible in the following ways:

1. Users on the client machine can access the remote files from any other machine running FINFS.
2. Any remote file on a server can be transferred to any other machine running FINFS.
3. A user can include any of his files in a remote list or delete a remote file from that list directly, without the intervention of a superuser.

Chapter II discusses the need for a new DFS. Chapter III discusses issues involved in designing a DFS. Chapter IV analyzes some of the existing DFSs with respect to these design issues. Chapter V discusses the overview of the protocols. Chapter VI discusses the implementation issues, and Chapter VII concludes the thesis.

CHAPTER II

Problem Statement

The distributed nature of a distributed file system (DFS) poses some unique challenges in design and implementation. TCP/IP and UDP/IP protocols are used to establish a connection between clients and servers. These communication protocols solve the problems of establishing connections, transmission of data, error checking, etc. Application software built using these protocols, like FINFS, manages the transactions made by clients and servers. Some of the problems tackled by the application software are heterogeneity, consistency of data, fault tolerance and user friendliness. A DFS may contain different types of file systems. This type of DFS is called as heterogeneous file system. Conversion of one format of file to another format is done by these DFSs. IBM's SAA is a heterogeneous distributed file system used in the CICS environment [5]. A new distributed file system presented by Cheng *et al.* discusses a heterogeneous DFS for MS-DOS[3]. Jade discusses a heterogeneous DFS for the Internet [14]. Since more than one client may try to modify a file at the same time, consistency problems may arise. Also, in a DFS environment all information about the client is lost in case of server crash. The problems of consistency and fault tolerance have received much attention from early designers of DFSs like Andrew [10, 13, 16], Coda [15, 16], and Locus [20].

The distributed nature causes special stress on user friendliness and flexibility in a DFS. The problem of user friendliness includes three issues: location transparency, location independence and user mobility.

1. Location Transparency: In any file system, a file is identified by its name and its path from the root. In a DFS, in addition to these, the name of the file system also has to be mentioned. This results in long names for remote files, causing inconvenience to the user. Moreover, a location dependent name may cause mapping problems when the file is renamed or relocated. To tackle this problem in a *global distributed file system* GDFS, a unique name is given to all remote sharable files. Also, renaming of a file locally is possible. Most of the earlier DFSs have achieved location transparency.

2. Location Independence (File Migration): It is always useful to have information as close as possible to the client [9]. Gavish *et al.*, argue that phenomenon of *locality of reference* applies to remote files, too [7]. In fact, Gavish *et al.*, argue that, in business application software there is a regular pattern in which locality of reference varies. They consider an example of an airplane flying from place 'A' to place 'B'. It can be observed that there are more references made to a file containing information about the flight at 'A' at the time of departure and at 'B' at the time of arrival. To improve performance, the file has to migrate from 'A' to 'B'. File migration is allowed in very few DFS. Andrew supports file migration. NFS does not support file migration. FINFS supports file migration.

3. User Mobility: File security is very important in a DFS. To make sure that files are secure, accredition checking must be made for each access request to a file. Servers maintain a list of legitimate clients and their access permissions. A client is normally identified by its machine address. This ties the user on the client machine to the machine. In some DFSs, such as Andrew and Locus, user mobility is provided by identifying the user by his home

directory. In NFS, user mobility is not supported. In FINFS, user mobility is supported.

In popular DFSs such as Andrew, Locus, and NFS, registration of a client, creating a remote file or deleting a remote file are done by the superuser. This helps in proper maintenance and unnecessary inclusion of remote files. For example, a client may be trying to have remote access to one of his own files. In FINFS, a user registers all the clients for his files and inclusion and deletion of remote files in the list is automatically done by the software. Proper care is taken to avoid situations like the one mentioned above. This makes the system user friendly and very much under user control.

In FINFS, providing access to remote directories is not allowed. In a UNIX environment, a directory is a special type of file. Directories contain information about the regular files belonging to the directory. In NFS[21], Andrew [13], Jade[14] and Locus [20], remote access is provided to directories.

CHAPTER III

Design Issues

3.1. Transparency

The aim of a DFS is to provide an environment similar to conventional file systems; that is, the user should be allowed to access a remote file with the same set of commands used to access a local file. For example, if a user is using the command *ls* to list all of the files in a local directory, the same command *ls* should be able to list remote files, too. This feature of a DFS is called *transparency.* If both client and server are running on UNIX-based machines, achieving transparency is much simpler than when either one of them is not. In that case, a proper interface has to be designed for all UNIX commands. Transparency can be achieved by supporting *location transparency* and *location independence.*

Location transparency means that the name of a file should not reveal the whereabouts of the file. For example, suppose that a file named 'file1' is on machine1 and a user on machine2 wants to access that file. Then, if reference to that file is made by referring to 'file1' we can say location transparency is achieved. If the file is referenced as 'machine1/file1' then location transparency is not achieved because the location of the file is mentioned while referencing the file.

To achieve location transparency in a global distributed file system, all sharable files in the network are given unique names. One exception is Prospero, where a sharable file is allowed to be renamed, and that name is valid only locally. In NFS, for each machine to which a user has access, a directory is maintained. All of the files in that machine to which the user has mounting permission can be accessed as if accessing a file in a local directory. FINFS attempts to achieve location transparency by allowing a client to rename the file locally.

Location independence means that files are allowed to be relocated; that is, a sharable file can move around within a network and still be accessed by the users who have mounting permission to that file without being aware of the fact that the file has been relocated. This is also called *file mobility*. The concept of file mobility makes a system very flexible. Among the present DFSs, only Andrew supports the notion of file mobility. FINFS also attempts to attain file mobility.

3.2. Fault Tolerance Issues

A *fault* in a system can be described as an event that reduces the performance of the system and may bring the system to a grinding halt. Failure of either a client or a server, or communication faults are examples of such events. In a DFS, faults are more frequent because of the number of machines in the system and interactions between them. Proper care should be taken to make a DFS *fault tolerant*. Failures of clients and servers pose totally different kinds of problems.

Client Failure: The effect of a client crashing depends on the type of access to remote files that is allowed in the system. If the status of a remote file access is maintained by the server, then it is called *stateful service*. If it is maintained by the client, it is called *stateless service*. For example, say that a remote file is being accessed. Then, the whole file is not accessed at one time. The file may be accessed block by block. In stateful service, after providing access to the first block, the server maintains the offset into the file. In stateless service, the client maintains the offset.

In stateful service, if the client crashes after accessing the first page, and the server learns about the crash, it can reset the status immediately. Usually, in stateful service after accessing a page, the server sets a timer. If no reference is made to the file within that time, the server resets the status automatically. In some DFSs, the server makes a periodic check or checks whenever memory is short. This is called *garbage collection.* If a DFS is large and serves many clients, garbage collection may be expensive.

In stateless service, if a client crashes after accessing a file, no action needs to be taken by the server because the status of the file is maintained by the client. One disadvantage of stateless service is that a client cannot lock the file it is accessing to prevent other users from accessing it. Hence, locking is always stateful. One other disadvantage with the stateless service is that every time an access request is made to a server, accreditation checks have to be performed by the server. This may result in overloading the server. NFS successfully employs a stateless service. FINFS also employ a stateless service. However, accreditation checks are made only the first time when the file is accessed and whenever the access permissions are changed.

Server Failure: Server failures result in non-availability of remote files. This is tackled by replicating files so that if a file server fails there is always another one with the same file, and the client can automatically access the file from some other server. However, maintaining consistency among all copies is a very complicated issue.

Communication faults can be controlled by reducing the number of interactions made to access a file by caching more data with every access. Normally communication faults are controlled by proper hardware design in lower layers of the network protocol.

3.3. Scalability Issues

The ability of a server to withstand fluctuations of load is called *scalability*. Servers handling more than one client may act as a bottleneck. If a machine is a dedicated server, it can be augmented with more than one CPU and more powerful CPUs. But, increasing the number of resources like CPUs may not provide an adequate solution. For example, in a DFS like NFS, a server can also be a client. Moreover, using more resources can result in under utilization of resources. The best way to tackle this problem is by minimizing the amount of interaction between the machines. For example, in a stateful service, a client has to periodically interact with the server just to reset the timer which increases the network traffic. A proper design of a DFS also helps in reducing the load on the server. By providing symmetry to the system (by distributing the servers throughout the network, instead of using a single dedicated server), scalability can be improved. By identifying the set of machines which have a larger number of interactions and providing an exclusive server for that group (known as a cluster), the load can be balanced. For example, all machines in a LAN can be clustered together because they interact more with other machines in the LAN.

3.4. Concurrency Issues

In a DFS there might be more than one access request for a remote file. There are three policies in use to tackle this situation. The first policy is to make all sharable files immutable; that is, provide only *read-only* files. Cedar is an example of such a file system [8]. The second policy is to provide a locking mechanism, so that a client can lock the file as soon as it has been accessed. This results in a queue being formed outside of the file. Each access provides the user at the head of the queue with the most recent version of the file. Since there is a queue formed, there may be delays in accessing the file which makes the system non-transparent. The third policy is to use the UNIX semantics of sharing. Here, any number of users can be provided with a copy of the remote file. Whenever changes made in the copy are saved they are saved in the master copy also. This does not guarantee strong consistency when remote files are accessed concurrently. Most of the DFSs follow UNIX semantics or a variation. Locking is popular among databases, and immutable files are used in sites which are used for sharing information. FINFS also follows the UNIX semantics.

<u>CHAPTER IV</u>

<u>Overview of Existing Systems</u>

This chapter gives an overview of some noteworthy DFSs which are very popular and widely used. All DFSs mentioned here are analyzed with reference to the design issues discussed earlier.

4.1 Andrew

Andrew is a huge DFS developed at Carnegie-Melon University [10, 13, 16]. It consists of dedicated servers serving clusters of clients which are usually workstations. Elaborate techniques are employed to provide location independence to the files and to minimize the network traffic.

Remote files are given network-wide unique names and their location is stored in a database called the *Location Database*. The Location Database is replicated on all servers in the network. This increases the availability of the Location Database. The names of the files are location transparent because they are unique. Each sharable file is given a file id *fid*, similar to a UNIX i-node, which is used as the physical address of the file. Fids have location independence; that is, a file can be relocated and each time it gets relocated, the present location of the file gets updated in the Location Database. If several clients are accessing a remote file concurrently, and if one of them changes the location of the files, then all write-backs after the relocation of the file are sent to the new location.

To minimize the network traffic and improve the scalability, a *whole file caching* technique is used. Whenever a file is accessed, the whole file is cached onto the local machine. Then, during write-back, the whole file is transferred to the server. As a result of whole file caching, clients interact with the servers only during opening and closing of files. If a cached file is not modified, then even that is not necessary during closing of a file. This key feature improves the performance of the whole system dramatically. Also, this makes implementing UNIX semantics of sharing simpler. However, if a file is concurrently accessed by more than one client, then all of the clients are informed about the modification. Remote service is a stateful service because whenever an access is made to a file, the status of the file is logged. This helps in keeping track of all clients who have accessed a particular file and to inform them in case that file is modified.

User mobility is provided. User can access any file from any workstation. Protection of files is provided by providing accreditation lists. The client-server communication is implemented using off-the-shelf packages such as RPC [2]. Hence, Andrew is portable and supports heterogeneity by providing a very good interface.

4.2 Locus

Locus is a DFS developed at UCLA [20]. It consists of dedicated servers serving clients which are on workstations and mainframes connected by an Ethernet. Fault tolerance and performance issues are emphasized in Locus.

Files in Locus are replicated and distributed to servers situated at different locations in the network. This is done mainly to improve the availability of files. Whenever a file is opened, a client sees the local copy. However, when a file is modified, the primary copy is updated, and all servers are informed about the modification. File replication also helps in case of server failure.

To improve performance, the client-server communication is tuned by designing specialized remote action protocols instead of using off-the-shelf packages like RPC [2]. These protocols are built as a part of the kernel instead of providing a layer above the kernel. This hampers portability and heterogeneity in Locus.

By using networkwide unique names to sharable files, location transparency is provided. In Locus, clients are called *Using Sites* (US) and servers are called *Selected Sites* (SS). A third entity which connects a SS to a US is called a *Currently Synchronized Site* (CSS). A typical request from a US for accessing a remote file is made to a CSS which finds the SS using the mount table maintained by the server and establishes communication between the US and the SS. Also, the CSS maintains a list of all remotely assessable files. Hence, at least one CSS should be accessible from each site, and all remote transactions from that site must pass through it. This may result in a CSS becoming a bottleneck. This hampers the scalability of Locus. Communication between the primary copy and the local copy during modification may result in an increase in network traffic.

Locus, in addition to using the UNIX semantics of sharing, provides locking for concurrency control, if required. Fault tolerance issues are emphasized by replication of files. Consistency is maintained by atomic broadcasting of modifications to all copies. Recovery of a server after failure is simple, as it simply has to copy all files from the primary copy to get the most recent version of the files.

4.3 Network File System

NFS is one of the most commercially successful DFS developed by Sun Microsystems Inc. [12, 21]. The best feature of NFS is that any pair of machines is allowed to share their files. In other DFSs, sharing of files is only between a client and a dedicated server. Kernel to kernel communication is implemented by using the TCP/IP protocol.

As explained earlier, remote access of files is performed in three stages: advertising, registering and session. Remote files are accessed transparently by using a *mounting* technique. However, while registering, the transaction is not transparent; that is, while registering, the physical address of the file is used to identify the file. Once registering is over, clients can rename the file locally and any reference to that file is done by using the local name. This is achieved by having a mount table consisting of *mounting points*. Mounting points are nothing but the location and real name of files to which a client has access permission. File protection is given at the user level; that is, the least addressable unit in the network is a user, instead of a workstation, which may be used by more than one user. However, a superuser of the machine maintains the mounting table for all the users who can log onto that workstation. In addition to the local files, a machine can give mounting permission to remote files to which it has mounting permission. This is called *cascading mounts*. However, having access to a remote file system does not imply access to all the other file system to which the other machine has mounting permission. In fact, a user who has mounting permission to a remote file system can see those files to which he has mounting permission and not other files in that file system. In NFS, sharable files are not allowed to be relocated. If they are relocated or renamed, all of the clients are required to obtain new access permissions. This means that location independence of files is not supported.

As mentioned earlier, NFS maintains a stateless service and hence it does not provide any concurrency control. Only the UNIX semantics of sharing are supported. However, instead of caching a whole file, a file is accessed page by page. After every access, the file offset is maintained by the client. Since sharing is allowed between any pair of machines, the system is fairly symmetric. Theoretically, there are no bottlenecks. This feature improves the scalability of the system.

Sun NFS is highly successful, heterogeneous, reliable, and a *de facto* standard commercially.

4.4. Prospero

Prospero is a revolutionary DFS developed at the University of Washington [4]. In addition to transparent access to remote files, emphasis is given to file and information organization. It consists of dedicated servers which are viewed as archives of information, serving a number of clients. The FTP sites in the Internet are integrated to form these archives.

Location transparency is provided by allowing the user to rename a file in an archive locally. A notion of *user centered naming* is used in naming a file. This helps in minimizing the confusion created by the global organization of files. This is achieved by allowing the user to create his own view of an archive. A user can create his own directories and include any file in that directory, and can access it through that directory. For example, user A may want to include a file which has information about a DFS in a directory called "network" whereas user B may include that in a directory called "distributed database". In addition to this to help a user organize files, the user is allowed to see another user's view of the archive. These are achieved by providing virtual links to the archive files. Hence, the name Virtual File System (VFS) is used. This also helps in achieving transparency. Typically, a link specifies the name of the host, the real name of the file and the local name to which it will be renamed. Real names are made networkwide unique by adding closures to the real name which are normally the location of the file. Once the link has been established, a reference to a local file name is mapped onto the real name and a query is made to the Prospero directory server which has the location of all files. Then, contact is made to that location.

Further, to help users in organizing files, some tools called filters are provided. For example, a filter of type *Distribute* helps in distributing files depending on some parameter mentioned by the user and *Union* helps in grouping together the files that are logically related.

Prospero, unlike other DFSs, instead of creating and storing data helps in organizing the available data. Prospero is highly heterogeneous and has a very good interface with Andrew and NFS.

CHAPTER V

Overview

This section briefly describes an overview and the implementation of FINFS. As in NFS, FINFS also has three stages in creating and accessing a remote file:
1. Advertising, 2. Registering, and 3. Session.

5.1. Advertising

As in NFS, a user (server) who has a sharable file, sends a message to the users (clients) with whom he would like to share it. The name of a sharable file is derived from the file creator's address and the time of creation. A user can advertise only those files which are presently resident on his machine. A user cannot advertise those files to which he has access through mounting. An advertisement consists of the following:

1. Sender's machine address,
2. Name of the file he is advertising, and
3. Type of access mode provided.

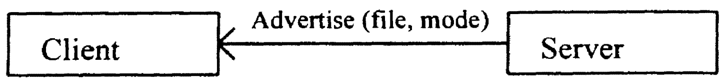

Fig 1. Advertisement

5.2. Registering

Clients who received the message will send a request for registering to the server. A request consists of the following:

1. Client's machine address, and
2. File name of the file he is requesting to access.

Before sending the registering request, the client is requested to rename the file locally. The client will refer to the file only by that local name in the future. At no point during the transaction is the network name of the file revealed either to the user on the server machine or to the user on the client machine. This not only helps in achieving location transparency, but also in hiding the network name of the file through which the file is accessed.

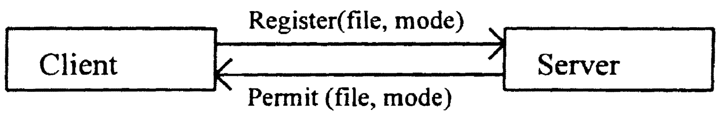

Fig 2. Registration.

After receiving the request, the server creates a mounting point. This includes setting the following parameters:

1. Local name of the file,
2. Client's address,
3. Access mode for the file,
4. Address of file creator, and
5. Network name of the file.

Finally, an acknowledgment of registration is sent to the client. On receipt of the acknowledgment, the client creates its own remote mounting point. This includes setting the following parameters:

1. Local name of the file,
2. Access mode to the file,
3. Present location of the file,
4. Location of the creator, and
5. Network name of the file.

5.3. Session

A reference to a remote file is made by the client. When referring to a remote file, a user uses to the local name. The local name is mapped onto the network name of the file and the present location of the file. A request for a session is made to the server. The server tries to locate the file in its file system. If it fails to locate the file, it returns a *NOFILE* message to the client. If a file is found, the server checks the access request made by the client for that session and the access permission, and provides access if the request is legitimate. If not, the server denies access to the file. The server provides the port number of the file server to the client. For the rest of the session, the client

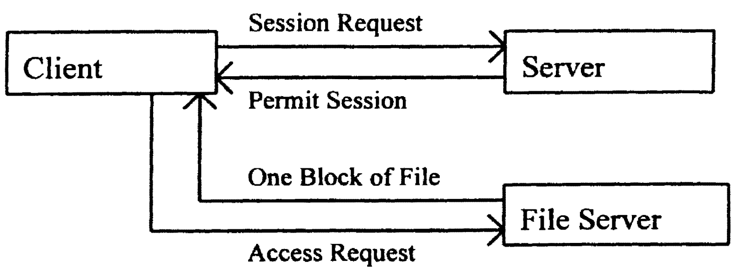

Fig 3. Session

communicates only with the file server. When a request is made, the file server checks the offset sent by the client (which is typically zero on the first request) and copies the file into a buffer from that point onwards until either EOF is reached or the buffer is full. If the buffer is full, it updates the offset, otherwise it sets the offset to -1. Throughout the transaction, the offset is maintained by the client. The file server responds to any number of requests from a client.

5.4 User Mobility

The user mobility is achieved as shown in Figure 4. The client, first contacts his machine and provides the password to his server. In addition to the password, the user has to mention the local name of the file he wishes to access. The client verifies the

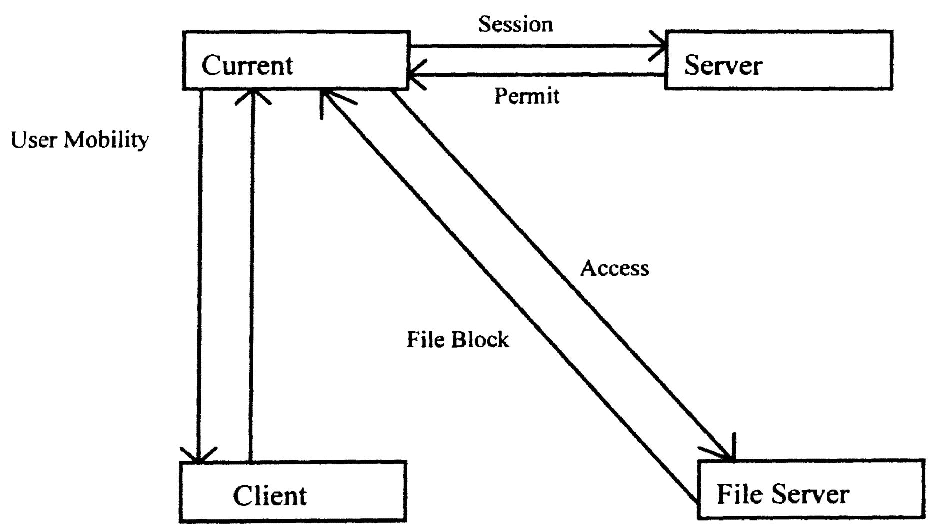

Fig 4. User Mobility

password. If it is legitimate, it provides the network name of the file and its present location to the current machine. Current machine than onwards can contact the present location of the file and access the file as if it has the access permission to that file through out that session. Other than providing the password to his machine, to the user of the client machine, the whole transaction is transparent.

5.5 File Mobility

File mobility or location independence is achieved as shown in the Figure 5. A client can access a file as long as the file resides on the server. Now, say, a server wants to transfer the file to a new server. The server sends a message to the new server about its intention of transferring the file. If the new server accepts, the server transfer the

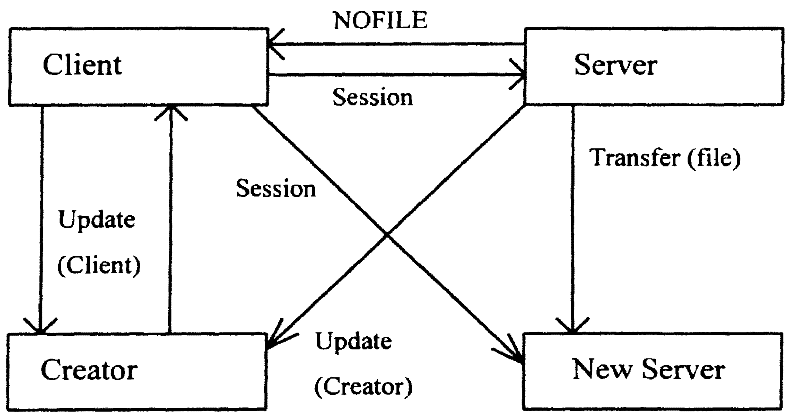

Fig 5. File Mobility

whole file to the new server. Then, the server transfers the part of the mounting table related to that file to the new server. Then, the server updates the whereabouts of the file to the creator of that file. After the file has been transferred, a client may want to access the file. Since the client is unaware of the file being transferred, the client sends an access request to the old server. The server will not be able to map the network name of that file to any of its local files. Hence, it will send back a *NOFILE* message to the client. A client that receives a *NOFILE* message determines the creator of the file, and sends a request for an update of the present location of the file to the creator. The creator looks into its updated table and provides the present location of the file, in this case the new server, to the client. Then, the client can access the file from the new server. Also, the client updates the present location of the file so that it can contact the new server directly to access the file in future. To the users on the creator, server , client, and the new server machines, the whole transaction is transparent.

CHAPTER VI

Implementation Issues

6.1 Platform

The above protocols were implemented on UNIX-based machines connected to the Internet using the Berkley sockets application program interface (API). The UNIX-based machines that were used include a SEQUENT S-81 multiprocessor machine with the Internet address 'a.cs.okstate.edu', and two VAX machines with Internet addresses of 'unx.ucc.okstate.edu' and 'osuunx.ucc.okstate.edu' here at Oklahoma State University. The above protocols could be implemented using other APIs.

6.2 Connection-oriented vs. Connectionless Service

The above protocols were implemented using UDP/IP protocols. Choice of a connectionless service is intentional. Even though, it is common to use a connection-oriented service for file transfer applications [19], a connectionless service is used due to the *virtual* nature of the transactions; that is, instead of transferring a file as one stream, in FINFS it is transferred page by page. However, to maintain reliability, acknowledgments are sent on receipt of a page or message. This helps in reducing the overhead cost of maintaining a permanent connection between the machines. To minimize the file handling and load on the network, access request were made only whenever the client machine doesn't find the required portion of the file in its cache. A user may need only the first page of a large file. In that case, accessing only that page instead of the whole file is more economical. If the user needs the second page, it can be accessed. But the first page is cached in the client machine so that if the user refers to the first page again in the future it will not need to be accessed again over the network. In the same way, a whole file can be cached only if it is required. In short, a server holding the remote file acts as a *virtual memory* once the session starts. This technique of accessing pages of a file only whenever they are required is called *lazy caching*. Since there is no fixed interval of time between the access requests, maintaining a connection between the machines when it is not in use may be less economical.

6.3. Naming

Naming of a file plays an important role in a distributed file system. The whole idea of naming a file is to generate a name for a file which is unique throughout the file system. Having a unique logical name for each file in a file system helps in reducing ambiguity in mapping a logical name to the physical location of the file. The conventional file system residing on a single machine also guarantees a unique file name for each file residing in it. This is achieved by identifying each file by its name and its path. Borrowing the same idea, even in a DFS a unique name can be guaranteed by identifying a file by its name, its path name and the location of the machine; that is, the Internet address of the machine. But the features of location transparency and location independence in a DFS pose new challenges in naming a file.

There are four methods of naming a file. Three methods of naming a file are mentioned in [1].

The first method is a simple method of forming a name of a file with some combination of host name and the local name of the file. Even though this guarantees a networkwide unique name, it is neither location transparent nor location independent [12].

In the second method, to overcome the above problem, in some systems, a networkwide unique name which is not dependent on the file's location is provided. Even though this is helpful in achieving location transparency, generating a unique name for each file is a difficult task [1]. Especially for system files which are present on all the machines, providing an unique name is very difficult.

NFS follows a new approach of creating a separate directory for each remote file, depending on the location of the machine in which the file is actually residing. All remote files residing on the same machine, can be grouped together in one directory. All files in that directory can be accessed by their name and path in that directory. To the user, accessing the remote directory is no different from accessing any of a local directory. This method is location transparent. The disadvantage of using this method is higher administrative cost and less flexibility as a remote file can be accessed only through a particular directory.

The fourth classic method is called *renaming* [4]. In this method, a remote file having a networkwide unique name is allowed to be renamed locally by the user of the local machine. This file is always accessed using the local name only. The machine maintains a table of which local files reside on which remote machine. This method is transparent and flexible as the renamed files can be moved to other local directories like any other local files. The disadvantage of this method is higher administrative cost.

In FINFS, a slightly modified version of the renaming method, proposed by Prospero [4], is employed. The inherent problem with *renaming* is that the files are not location independent. Location independence feature of a DFS pose a new challenge. Say, a file 'A' residing on 'machine A' has a network name of 'A/machine A'. Say, 'machine B' has remote access to this file. Then, 'machine B' can access that file by the name 'A/machine A'. Now suppose that, file 'A' has been transferred to another machine

'machine C' and 'machine A' creates another file with the same name 'A'. The new file have a network name of 'A/machine A'. Now, 'machine B' automatically has remote access to the new file to which it is not supposed to have access. To overcome this problem, in FINFS, in addition to the location of the file being attached to the local name of the file to form the network name, the creation time of the file is also attached. Even Apollo Domain uses time stamping as part of the name of a file [11]. If the above situation arises in FINFS, the network names of the old file 'A' and new file 'A' would be different. The access request for file 'A' from machine 'B' after the transfer of old file 'A' would be invalid. As in Prospero, clients are allowed to rename remote files [4]. A typical network name for a file in FINFS would be 'local name/path name/Internet address/time of creation'. Change of location of a file does not change its network name; that is, the network name always contains the local name it had at the time of its creation and the Internet address of the creator. Renaming of a file only changes the local name to which this network name is mapped onto locally in that particular machine. The 'Internet address' and 'time of creation' part of the network name has no significance, other than making it unique throughout the file system. The present location of a remote file is always saved along with its local name and network name. When a reference to a local name of a remote file is made in addition to the network name its mapped on to the present location also. The disadvantage of this method is a higher administrative cost for keeping track of the present address of each remote file. But the advantages of a flexible, transparent and location independent file server, clearly outweigh the disadvantage.

6.4 Mounting

A mounting technique is used by NFS[21, 12], Andrew [13, 16], Locus [20] file systems. Mounting can be achieved by accessing mounting points. A *mounting point* provides an entry to another file system. A mounting point, should contain at least the local name of the file, the network name of the file and the location of the file. It maps the local name to the network name and to the present location. In addition to these, a mounting point can store other information also. In FINFS, a mounting point stores the mode of accessing, and the creator's location. A list of all such mounting points is called a *mounting table*. In a mounting table, all entries can be grouped either according to the file name or location name. On the server side, where the file resides, a mounting table can be grouped according to the file name. On the client side, all files residing at the same place can be grouped together. This helps to improve the performance of accessing mounting points in large servers and clients. Since FINFS is a network file system, with a separate mounting table provided for each user, it was assumed that the number of files to which each user has remote access is small. Hence, in FINFS the mounting point entries are created when the file is created.

In FINFS the mounting table is stored in a hidden file. Separate files for server side mounting points and client side mounting points are maintained. Saving the mounting points in a file helps in preserving them in case of a crash.

6.5 Stateless Service

A stateless server is one which is not aware of a client being active or not. As mentioned earlier, if a server maintains the offset of a file which has been accessed from a client, then that server is a stateful server. A server can be a stateful server even if the client maintains the offset. For example, if a server creates a separate file server when it provides access permission to a remote client, and terminates it when the transaction is completed, then it is stateful server. In the above example, even though the client maintains the offset, the file server waits for an end of transaction indication from the client to terminate. In case of a client crashing before sending the end of transaction indication, there is no way the file server can terminate. So, the server has to periodically check, and terminate all file servers which are not active. In short, it has to do *garbage collection* as in stateful service. Hence, this type of a server is also a stateful server.

To overcome the above problem, a truly stateless server processes all requests. A file server running in the background processes all file handling requests from all clients. This technique is helpful in off loading the server and eliminating the accredition checking each time a client accesses the file.

6.6 Security

Security is required to prevent either an illegitimate user from accessing a file or a legitimate user accessing a file with an illegitimate mode. This helps in ensuring the consistency and integrity of the file system. In FINFS, the network name is hidden. At no point of time is the network name of a file revealed to the user. The files containing network names of files, like the mounting table and location table are hidden and hence, it can not be tampered. However, if a user wants to know to what files and what mode he has access to, a separate command 'rls' is provided to list remote files. Creation of a network name for a file by a user is also difficult, as networks names are formed by the combining the file creator's location and the file's time of creation which are not revealed to the user.

To prevent users from accessing a file with an illegitimate mode, the server checks the mode of accessing before providing access permission. If its an illegitimate mode, the server sends a DENY message.

CHAPTER VII

Conclusion

7.1 Summary

A DFS is created by integrating file systems on different machines over a geographical area. Presently only the Andrew file system has made an attempt to allow file mobility [13, 16]. FINFS makes an attempt to provide file mobility. Further, to make the system more flexible, an attempt is made to provide user mobility. FINFS identifies a file's owner as the ultimate authority to provide remote access to his files. By bringing the control down to the user level, the system is more flexible and more under the user's control.

The main goal of FINFS is to achieve user mobility and file mobility. User mobility is achieved by identifying a user by his password. File mobility is achieved by transferring a file to other machine and updating the present location of the file with the creator. Transactions other than advertising and registering are transparent.

FINFS is implemented on UNIX-based machines using Berkley sockets. It is a stateless, virtual, fault tolerant and reliable file system, and uses the UNIX semantics of sharing.

Finally, the features of FINFS can be summarized with respect to the following issues:

1. Scalability : FINFS is a symmetric network file system. It prevents any concentration of load at any point.
2. Concurrency: FINFS allows concurrent access of a file.
3. Sharing Semantics : FINFS supports the UNIX semantics of sharing; that is, the most recent modification of a file is stored. This may lead to some consistency problems when files are accessed concurrently. For example, if two users access the first page of a file and if one of them modifies it, the other will only have the older version.
4. Fault Tolerance : FINFS is a stateless server. Performance of the server is not affected if a client crashes. However, in case of server crash, files are not available as no replication of the files is performed. A client tries to contact the server at increasing intervals of time. After trying for certain number of times, the client prompts the user about the unavailability of the server. To minimize the risk of losing the information about the mounting points, in case of a crash, update is made to a temporary file and then that file is renamed. In case of file transfer (file mobility), after the creator is informed about the location of the file, the server updates its mounting points.
5. Transparency : The main goal of FINFS is to achieve transparency and flexibility. Transparency is achieved by allowing the user to rename a remote file. Other than advertising, registering and transfering files, all other transactions are transparent. For user mobility, a user has to provide his password to access his own files or remote files from other than his machine.
6. Flexibility : FINFS is a very flexible file system. All remote transactions are the under the user's control. A user can provide remote access to his files to other users and remove any remote access to others any time he wants. Unlike NFS, intervention by the superuser is not required.
7. Speed and Reliability : Speed of accessing a remote file mainly depends on

the network communication speed on which FINFS does not have any control. The speed also depends on the block size used to fetch the information from the remote machine. In addition to that, since in FINFS, all of the information about the mounting points, registration requests and location table are not cached. This helps in preventing loss of information in case of machine failure. However, this introduces a delay in file seeking every time an access request or registration request made. No advance caching is done by the file server. Every time a file is accessed, the file server seeks that file and copies the required portion of the file. Even though these features causes delay, these are necessary to maintain a fault tolerant and stateless server.

FINFS is a highly reliable file system. It is fault tolerant. No users have access to the mounting table or location table. This prevents users from tampering with it. To prevent misuse, the access mode provided and the access mode a user requesting is checked before providing the access. No user can have two access points to a single file. If a user tries to advertise the same file to the same client a second time, FINFS prompts the user. A file which is not resident cannot be advertised. These features ensure that only legitimate users can access the files and nobody else can access the files without the knowledge of the server.

7.2 Scope of Future Work

Presently, in FINFS access is only allowed for regular files. This can be extended to directories, block special, and character special files. Transparency can be improved by installing the software into a file system. Also, FINFS can be implemented as a utility on any UNIX machine by reserving ports for the file server and distributing the messages received depending upon the login name. Error handling features, such as checking disk space availability before transferring the file to the receiving machine, can be done. Presently, FINFS supports ASCII files only. In the future this can be extended to binary files as well.

REFERENCES

1. Barak, A., Malki, D., Wheeler, R., "AFS, BFS, CFS ... or Distributed File Systems for UNIX", Users Group Conference Proceedings, 461-472, Sept. 1986.
2. Birrel, A.D., Nelson, J.B., "Implementing remote procedure calls", ACM Transactions on Computer Systems, 2(1):39-59, Feb. 1984.
3. Cheng, H., Sheu, J., "Design and implementation of a distributed file system", Software: Practice & Experience, 21(7): 657-675, Jul. 1991.
4. Clifford, N. B., "The Prospero file system based on the virtual system model", Computing Systems, 5(4): 407-430, Fall 1992.
5. Deinhart, K., "SAA distributed file access to the CICS environment", IBM Systems Journal, 31(3): 516-534, 1992.
6. Comer, D. E., Stevens, D. L. "Internetworking with TCP/IP", Prentice Hall Inc., 1993.
7. Gavish, B., Liu, S., Olivia, R., "Dynamic file migration in distributed computer systems", Communications of the ACM, 33(2): 177-189, Feb. 1990.
8. Gifford, D.K., Needham, R.M., Schroeder, M.D., "The Cedar File System", Communications of the ACM, 31(3), 288-98, Mar. 1988
9. Goscinski, A., Beaton, K., "A simple distributed computer system for supporting collaboration in distant and synchronous meetings", Computers in Industry, 12(7): 95-106, May 1989.
10. Kerr, S., "IBM's NFS alternative (Andrew File System)", Datamation, 34(35): 63-65, Jan. 1, 1989.

11. Leach, P. J., Stump, B. L., Hamilton, J. A., Levine, P. H., "UIDs as internal names in a distributed file system", Proceedings of the 1st Symposium on Principles of Distributed Computing of ACM, 34-41, Aug. 1982.

12. Levy, E., Silberschatz, A., "Distributed file systems : Concepts and examples", ACM Computing Surveys, 22(4):321-374, Dec. 1990.

13. Morris, J. H., Sathyanarayanan, M., Conner, M. H., Howard, J. H., Rosenthal, D. S. H., and Smith, F. D., "Andrew: A distributed personal computing environment", Communications of ACM, 29(3):184-201, Mar. 1986.

14. Rao, C. H., Peterson, L. L., "Accessing File in an Internet: Jade File System", IEEE Transactions on Software Engineering, 19 (6): Jun. 1993.

15. Sathyanarayanan, M., "Disconnected operation in the Coda File System", ACM Transactions on Computer Systems, 10(1): 3-25, Feb. 1992.

16. Sathyanarayan, M., "Scalable, secure, and highly available distributed file access (Andrew and Coda)", Computer 23(2): 9-18, May 1990.

17. Stevens, R. W., "UNIX Network Programming", Prentice Hall Inc., 1990.

18. Svobodova, L., "File servers for network-based distributed systems", Computing Surveys, 16(7): 353-398, Dec. 1984.

19. Tanenbaum, S. A., "Computer Networks", second edition, Prentice Hall Inc., 1988.

20. Walker, B., Popek, J., English, R., Kline, C., Thiel, G. "The LOCUS distributed operating system". ACM SIGOPS, operating system review, 17(5):49-70, Oct. 1983.

21. Walsh, D., Lyon, R., Sager, G., "Overview of the Sun Network File System", Proceedings of the Usenix Winter Conference (Dallas Texas), 117-124, 1985.

APPENDIX A

GLOSSARY

A *client* is a process that requests a service from some other process.

A *Distributed File System (DFS)* consists of a number of file systems located on geographically distributed machines. A DFS manages remote sharable files.

A *dedicated server* is a server which is used only for storing and retrieving information.

A *file* is a data structure that is used for long term storage of data.

A *file server* is a server that provides access to requested files.

A *file system* is the part of the operating system that control access to a collection of files.

A *Global Distributed File System (GDFS)* is a type of distributed file system in which a file is identified by a unique name throughout the network.

A *Network File System (NFS)* is a type of distributed file system in which a machine can be both server and client.

A *remote file* for a machine is a file that is not situated on the same machine.

A *server* is a process that provides a service to a client when a request is made.

A *shared file* is a file which can be shared by more than one user.

APPENDIX B

SOURCE CODE

```
/*
*************************************************************************************
*       File:   Defs.h
*       Include file:     limits.h.
*
*       Contains definitions of some of the variables used in FINFS.
*
*************************************************************************************
*/

#include <limits.h>                             /* Limit for number of characters
                                                for password.  */

/* Transaction requests varibles.  */

#define DENY                    -1              /*  for access deny.  */
#define ADVERTISE               0               /*  for advertising.  */
#define REGISTER                1               /*  for registration. */
#define PERMITED                2               /*  for access permission.  */
#define SESSION                 3               /*  for a session request.  */
#define ACCESS                  4               /*  for permitting an access.  */
#define NOFILE                  5               /*  If no file is present.  */
#define CLOSE                   6               /*  To write back a file.  */
#define USER_MOBIL              7               /*  For user mobility request.  */
#define FILE_MOBIL              8               /*  For transfering a file.  */
#define UPDATE_PRESENT          9               /*  To update location of a file.  */
#define DELETE                  90              /*  Deletion of a mounting point.  */
#define SERV_PRESENT            91              /*  Request for present location
                                                of a file.  */

#define ERROR                   -1              /*  To indicate failure.  */
#define SUCCESS                 1               /*  To indicate a success.  */

/*      Message size declarations.  */

#define MAXLINE         512
#define PACKSIZE        500
#define ACKSIZE                 100             /* Max acknowledgement size.  */
#define FILESIZE                3300            /* To transfer a file.  */

#define TIME_INTERVAL           10              /* Time interval between sending
                                                message and receiving ack.  */

#define MAX_RESEND              5

char                            PASWORD[PASS_MAX];
```

```
/*
*********************************************************************************
*       File:  Headers.h
*
*       Contains standard include files which are used in FINFS.
*
*********************************************************************************
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
```

```
/*
*********************************************************************************
*
*       File:    Structures.h.
*
*       Contains declarations of structuress used through out FINFS.
*
*********************************************************************************
*/

        typedef struct Package {
                int                 header;           /* Contains transaction request.  */
                int                 owner_port;       /* Owners port.  */
                int                 toport;           /* Clients port.  */
                int                 present_port;     /* File's present location port.  */
                char                local[30];        /* Local name of the file.  */
                char                rlocal[30];       /* Clients name of the file.  */
                char                network[80];      /* Network name of the file.  */
                char                owner[30];        /* Owner's address.  */
                char                present[30];      /* Present address of the file.  */
                char                toa[30];          /* Client's address.  */
                char                permissions[4];   /* Access permission for a file.  */
                char                passwd[9];        /* Password used during user mobility*/
                char                request[4];       /* Session access mode.  */
                char                login[50];        /* Server's login.  */
                struct Package      *next;
                } Package;


/* Structure for transfering a file.  */

        typedef struct fnode {
                int                 header;
                int                 client_port;
                int                 server_port;
                long                offset;           /* Offset of an accessed file.  */
                int                 index;
                char                request[4];
                char                client[40];
                char                server[40];
                char                network[80];
                char                local[80];
                char                mesg[10][100];
                struct fnode        *prev;
                struct fnode        *next;
                } F_NODE;
```

```
/*
********************************************************************************
*
*       File:    Sys.h.
*
*       Contains system dependant variables.
*
********************************************************************************
*/

#define SERV_UDP_PORT            6464                    /* Address server's port.
*/
#define FILE_CLI_PORT            6664              /* client's file port. */
#define FILE_SERV_PORT           6767              /* File server's port. */
#define FILE_MOBIL_PORT          6868              /* Client's file mobile port. */
#define FILE_MOBIL_SERV_PORT     6768              /* Server's file mobile port. */
#define ACK_PORT                 8887              /* Port for acknowledgements. */

#define SELF_NAME                "a.cs.okstate.edu"  /* Self Internet address. */
#define LOGIN                    "mojnida"         /* Self login name. */
```

```
/*
******************************************************************************
*
*       File:      Util.h.
*       Include files:  Nil.
*       Functions:  prmesg, erro_dump.
*
*       Contains message printing functions.
*
******************************************************************************
*/

/*
******************************************************************************
Function to print a message.
******************************************************************************
*/

prmesg (s, mesg)
char    s[];
char    mesg[];
{
        fprintf (stderr, "\n\n%s\n%s\n\n", s, mesg);
}

/*
******************************************************************************
Function to print message and exit.
******************************************************************************
*/

erro_dump (s)
char    s[];
{
        fprintf(stderr, "%s", s);
        exit(1);
}
```

```
/*
*********************************************************************************
*
*       File:     server.h
*       Include Files:     Nil.
*       Functions:  server.
*
*       Contains function for binding a server to a port.
*********************************************************************************
*/

/*
********************************************************************************
*
*       Function to bind a socket to a port and returns a socket.
*
********************************************************************************
*/

server(SELF_PORT)
int     SELF_PORT;
{
        int                     sock;
        struct sockaddr_in      serv_addr;


        if ((sock = socket (AF_INET, SOCK_DGRAM, 0)) < 0)
                erro_dump("server:can't open datagram socket");

        serv_addr.sin_family            = AF_INET;
        serv_addr.sin_addr.s_addr       = htonl (INADDR_ANY);
        serv_addr.sin_port              = htons (SELF_PORT);

        if (bind (sock, (struct sockaddr *)
                &serv_addr, sizeof (serv_addr)) < 0){
                printf ("Port Number:     %d\n", SELF_PORT);
                erro_dump ("Server: Can't bind local address.");
                }

        return (sock);

}
```

```
/*
********************************************************************************
*                       FILE :  client.c
*                       Include Files :     singal.h
*                       Functions:          client, shakehand,
*                                           resend, send_ack, client_ack.
*
*       Handles all client transactions.
*
********************************************************************************
*/

#include <signal.h>

void    Resend();                                      /* Function for resending the package. */

struct sockaddr_in              serv_addr, cli_addr;    /* Server and client addr.  */
int                             sockc, sockserv;        /* Sockets.  */
char                            sendline [FILESIZE + ACKSIZE]; /* String containing message.  */
int                             Timeout, resend;        /* Variables for inter message time interval
                                                        and number of times to send the message.
*/


/*
********************************************************************************
        Function to format the messge and sending it.
********************************************************************************
*/

client (char        HostName[],  int PortNum,char       cpack[], int size)
{
        struct hostent              *hp, *gethostbyname();


        serv_addr.sin_family        = AF_INET;
        serv_addr.sin_port          = htons(PortNum);
        sethostent(0);
        if ((hp = gethostbyname (HostName)) == 0)
            erro_dump("Client: Can't find host.\n");

        memcpy(&serv_addr.sin_addr, hp->h_addr, hp->h_length);

        /* Socket for sending the message.  */

        if ((sockc = socket (AF_INET, SOCK_DGRAM, 0)) < 0)
            erro_dump ("Client: can't open datagram socket.\n");

        cli_addr.sin_family         = AF_INET;
        cli_addr.sin_addr.s_addr    = htonl(INADDR_ANY);
        cli_addr.sin_port           = htons(0);

        if (bind (sockc, (struct sockaddr *) &cli_addr, sizeof (cli_addr)) < 0)
            erro_dump ("Client: Can't bind local address.\n");
        Timeout = TIME_INTERVAL;  /* TIME_INTERVAL defined in Defs.h.  */
        resend = 0;
        sockserv = server (ACK_PORT); /* Bind a server to recieve ack.  */

        sprintf (sendline, "%d %s %s", ACK_PORT, SELF_NAME, cpack);

        shakehand();  /* Function to send the messge and recieve ack.  */

        close (sockc);
        close (sockserv);

        return (SUCCESS);
}


/*
*********************************************************************************
        Function to send the message and wait for an ack.
*********************************************************************************
*/

shakehand ()
{

        int                     n, clilen;
        int                     size;
        char                    recvline[ACKSIZE + 1];

        size = strlen (sendline) + 1;

        if ((sendto (sockc, sendline, size, 0, (struct sockaddr *)
                &serv_addr, sizeof(serv_addr))) != size)
                        erro_dump ("Error in sending the message.\n");

        signal (SIGALRM, Resend);                       /* Resend is called if
                                                        ack is not recieved.  */

        alarm (Timeout);                                /* Set the alarm.  */

        n = recvfrom (sockserv, recvline, ACKSIZE + 1, 0, NULL, &clilen);

        alarm (0);                                      /* Reset the alarm.  */
        printf ("Message Reached.\n\n\n");

}
```

```
/*
*********************************************************************************
        This function is reached only if ack is not reached.  It resends
the message.
*********************************************************************************
*/

void    Resend()
{
        resend += 1;                        /* Count the number of times the message is sent.  */
        Timeout *= 2;

        /* MAX_RESEND is defined in Defs.h.  */

        if (resend > MAX_RESEND) erro_dump ("Server may not be active.\n");
        printf ("\nResending the package.\n");
        shakehand();
}

/*
*********************************************************************************
Function to send the acknowledgements.
*********************************************************************************
*/

client_ack (char  HostName[], int PortNum,char       cpack[], int size)
{
        struct sockaddr_in          ser_addr, clin_addr;
        int                         sock;
        struct hostent              *hp, *gethostbyname();

        ser_addr.sin_family         = AF_INET;
        ser_addr.sin_port= htons(PortNum);
        sethostent(0);
        if ((hp = gethostbyname (HostName)) == 0){
        printf ("HOSTNAME = %s\n", HostName);
             erro_dump("Client_ack: Can't find host");
             }

        memcpy(&ser_addr.sin_addr, hp->h_addr, hp->h_length);

        if ((sock = socket (AF_INET, SOCK_DGRAM, 0)) < 0)
             erro_dump ("Client_ack: can't open datagram socket.\n");

        clin_addr.sin_family        = AF_INET;
        clin_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        clin_addr.sin_port          = htons(0);

        if (bind (sock, (struct sockaddr *) &clin_addr, sizeof (clin_addr)) < 0)
        erro_dump ("Client: Can't bind local address.\n");
```

```
                if ((sendto (sock, cpack, size, 0,
                        (struct sockaddr *) &ser_addr,
                                sizeof(ser_addr))) != size)
                        erro_dump ("Error in sending acknowledgement.\n");

        close (sock);

        return (SUCCESS);
}

/*
*******************************************************************************
        Function to send the ack and strip off the ack from the message.
*******************************************************************************
*/

send_ack (mesg, cpack, Host)
char    mesg[];
char    cpack[];
{
        char    ack_addr[80];
        int     ack_port, n;



        /*  Set the acknowledgement.   */

        sscanf (mesg, "%d %s %n", &ack_port, ack_addr, &n);

        /* Get the message in cpack.  */

        strcpy (cpack, mesg + n);
        strcpy (Host, ack_addr);
        client_ack (ack_addr, ack_port, "RECIEVED", 8);

}
```

```
/*
********************************************************************************
*
*       File:  mount.h.
*       Include Files:  Headers.h, Sys.h Util.h, Defs.h
*                               Structers.h
*       Functions : create_package, print_pack, unpak_pack
*                       pak_pak, set_pack.
*
*       Contains Package handling functions.
********************************************************************************
*/

/*      Headers files other than the standard header files.  */

#include <stdio.h>
#include "Headers.h"
#include "Sys.h"
#include "Util.h"
#include "Defs.h"
#include "Structures.h"


/*
********************************************************************************
*
*       Function to create a Package and initialize all the fields.
*
********************************************************************************
*/

Package *create_package (pack)
Package *pack;
{

        /* Create a Package.  */

        if ((pack = (Package *) (malloc (sizeof (Package)))) == NULL)
        erro_dump ("mount.h: Error in memory allocation for Package");

        /* Initialise the Package.  */

        strcpy (pack->local, "!");
        strcpy (pack->rlocal, "!");
        strcpy (pack->network, "!");
        strcpy (pack->present, "!");
        strcpy (pack->owner, "!");
        strcpy (pack->passwd, "!");
        strcpy (pack->login, "!");
        strcpy (pack->toa, "!");
        strcpy (pack->permissions, "!");
        strcpy(pack->request, "!");
        pack->header = -2;
        pack->owner_port = -2;
        pack->present_port = -2;
        pack->toport = -2;

        return (pack);
}

/*
********************************************************************************
*
*       Function to print a Package.
*
********************************************************************************
*/

print_pack (pack)
Package *pack;
{

        printf ("Header:                        %d\n", pack->header);
        printf ("owner_port:                    %d\n", pack->owner_port);
        printf ("toport :                       %d\n", pack->toport);
        printf ("present_port:                  %d\n", pack->present_port);
        printf ("local:                         %s\n", pack->local);
        printf ("rlocal:                        %s\n", pack->rlocal);
        printf ("Network:                       %s\n", pack->network);
        printf ("Owner:                         %s\n", pack->owner);
        printf ("Present:                       %s\n", pack->present);
        printf ("To:                            %s\n", pack->toa);
        printf ("permissions:                   %s\n", pack->permissions);
        printf ("request:                       %s\n", pack->request);
        printf ("login:                         %s\n", pack->login);
}


/*
********************************************************************************
*
*       Function to unpack the data from the given Package and
*       arrange the data in message format.
*
********************************************************************************
*/


unpak_pack(pack, cpack)
Package pack;
char    cpack[];
{
        char    temp[40];

        cpack[0] = '\0';

        sprintf (temp, "%d", pack.header);
        strcat (cpack, temp);
        strcat (cpack, " ");
        sprintf (temp, "%d", pack.owner_port);
        strcat (cpack, temp);
        strcat (cpack, " ");
        sprintf (temp, "%d", pack.toport);
        strcat (cpack, temp);
        strcat (cpack, " ");
        sprintf (temp, "%d", pack.present_port);
        strcat (cpack, temp);
        strcat (cpack, " ");
        strcat (cpack, pack.local);
        strcat (cpack, " ");
        strcat (cpack, pack.rlocal);
        strcat (cpack, " ");
        strcat (cpack, pack.network);
        strcat (cpack, " ");
        strcat (cpack, pack.owner);
        strcat (cpack, " ");
        strcat (cpack, pack.present);
        strcat (cpack, " ");
        strcat (cpack, pack.toa);
        strcat (cpack, " ");
        strcat (cpack, pack.permissions);
        strcat (cpack, " ");
        strcat (cpack, pack.passwd);
        strcat (cpack, " ");
        strcat (cpack, pack.request);
        strcat (cpack, " ");
        strcat (cpack, pack.login);
        strcat (cpack, " ");
}

/*
******************************************************************************
*
*       Function to break-up a message and load into a Package.
*
******************************************************************************
*/

pak_pack (pack, cpack)
Package *pack;
char    cpack[];
{
        int                     i = 0, j, num;

        sscanf (cpack + i, "%d %n", &num, &j);
        pack->header = num;
        i = i + j;
        sscanf (cpack + i, "%d %n", &num, &j);
        pack->owner_port = num;
        i = i + j;
        sscanf (cpack + i, "%d %n", &num, &j);
        pack->toport = num;
        i = i + j;
        sscanf (cpack + i, "%d %n", &num, &j);
        pack->present_port = num;
        i = i + j;
        sscanf(cpack + i, "%s %n", pack->local, &j);
        i = i + j;
        sscanf(cpack + i, "%s %n", pack->rlocal, &j);
        i = i + j;
        sscanf(cpack + i, "%s %n", pack->network, &j);
        i = i + j;
        sscanf(cpack + i, "%s %n", pack->owner, &j);
        i = i + j;
        sscanf(cpack + i, "%s %n", pack->present, &j);
        i = i + j;
        sscanf(cpack + i, "%s %n", pack->toa, &j);
        i = i + j;
        sscanf(cpack + i, "%s %n", pack->permissions, &j);
        i = i + j;
        sscanf(cpack + i, "%s %n", pack->passwd, &j);
        i = i + j;
        sscanf(cpack + i, "%s %n", pack->request, &j);
        i = i + j;
        sscanf(cpack + i, "%s %n", pack->login, &j);
        i = i + j;
}

/*
******************************************************************************
*
*       Function to initialise a given Package.
*
******************************************************************************
*/

set_pack (pack)
Package *pack;
{

        pack->header = -2;
        pack->owner_port = -2;
        pack->toport = -2;
        pack->present_port = -2;
        strcpy (pack->local, "!");
        strcpy (pack->rlocal, "!");
        strcpy (pack->network, "!");
        strcpy (pack->owner, "!");
        strcpy (pack->present, "!");
        strcpy (pack->toa, "!");
        strcpy (pack->permissions, "!");
        strcpy (pack->passwd, "!");
        strcpy (pack->request, "!");
        strcpy (pack->login, "!");
}
```

```
/*
********************************************************************************
*
*       File:  fdpak.h.
*       Include files:      Nil.
*       Functions:  unpack_fd, pack_fd, set_fd.
*
*       This file contains functions for unpacking, packing and initialising
*       a given F_NODE.
*
********************************************************************************
*/

/*
********************************************************************************
*
*       Function to unpack a given F_NODE and arrange the data in the
*       message format.
*
********************************************************************************
*/

unpack_fd (fd, cpack)
char    cpack[];
F_NODE          fd;
{
        char    temp[80];
        int     i;

        cpack[0] = '\0';
        sprintf (temp, "%d", fd.header);
        strcat (cpack, temp);
        strcat (cpack, " ");
        sprintf (temp, "%d", fd.client_port);
        strcat (cpack, temp);
        strcat (cpack, " ");
        sprintf (temp, "%d", fd.server_port);
        strcat (cpack, temp);
        strcat (cpack, " ");
        sprintf (temp, "%ld", fd.offset);
        strcat (cpack, temp);
        strcat (cpack, " ");
        sprintf (temp, "%d", fd.index);
        strcat (cpack, temp);
        strcat (cpack, " ");
        strcat (cpack, fd.request);
        strcat (cpack, " ");
        strcat (cpack, fd.client);
        strcat (cpack, " ");
        strcat (cpack, fd.server);
        strcat (cpack, " ");
        strcat (cpack, fd.network);
        strcat (cpack, " ");
        strcat (cpack, fd.local);
        strcat (cpack, " ");

        for (i = 0; i < 10; i++) {
                if (strcmp (fd.mesg[i], "!") == 0) break;
                strcat (cpack, fd.mesg[i]);
                strcat (cpack, " ");
                }
}


/*
********************************************************************************
*
*       Function to break-up the message into a F_NODE.
*
********************************************************************************
*/

pack_fd (fd, cpack)
F_NODE          *fd;
char    cpack[];
{
        int     i = 0, j, k, num, FLAG;
        long    Num;
        char    temp[100];

        sscanf (cpack, "%d %n", &num, &j);
        fd->header = num;
        i = i + j;
        sscanf (cpack + i, "%d %n", &num, &j);
        fd->client_port = num;
        i = i + j;
        sscanf (cpack + i, "%d %n", &num, &j);
        fd->server_port = num;
        i = i + j;
        sscanf (cpack + i, "%ld %n", &Num, &j);
        fd->offset = Num;
        i = i + j;
        sscanf (cpack + i, "%d %n", &num, &j);
        fd->index = num;
        i = i + j;

        sscanf (cpack + i, "%s %n", fd->request, &j);
        i = i + j;
        sscanf (cpack + i, "%s %n", fd->client, &j);
        i = i + j;
        sscanf (cpack + i, "%s %n", fd->server, &j);
        i = i + j;
        sscanf (cpack + i, "%s %n", fd->network, &j);
        i = i + j;
        sscanf (cpack + i, "%s %n", fd->local, &j);
        i = i + j;
        j = 0;
        k = 0;
        FLAG = 1;
        for ( ; cpack[i] != '\0'; i++){
                if ((FLAG == 1) && (cpack[i] == ' ')) FLAG = 0;
                else {
                fd->mesg[j][k] = cpack[i];
                k++;
                FLAG = 0;
                }
                if (cpack[i] == '\n') {
                        fd->mesg[j][k] = '\0';
                        j++;
                        k = 0;
                        FLAG = 1;
                        }
                }
        k++;
        fd->mesg[j][k] = '\0';
}

/*
********************************************************************************
*
*       Function to initialise a F_NODE.
*
********************************************************************************
*/

set_fd (fd)
F_NODE          *fd;
{
        int                     i;

        fd->header = -2;
        fd->server_port = -2;
        fd->client_port = -2;
        fd->offset = -2;
        fd->index = -2;
        strcpy (fd->request,"!" );
        strcpy (fd->client,"!" );
        strcpy (fd->server,"!" );
        strcpy (fd->network,"!");
        strcpy (fd->local, "!");
        for (i = 0; i < 10; i++)
                strcpy (fd->mesg[i], "!");

        fd->prev = NULL;
        fd->next = NULL;
}
```

```
/*
********************************************************************************
*
*       File:  Virtual.h
*       Include files: Nil.
*       Functions:  create_fd, ld_fd, wtransact,
*                   rtransact, etransact, update_mt,
*                   get_prev, get_next, copy_fd, copy_pfd,
*                   wind_up.
*
*       Contains file handling functions which are called by
*       remopen and umopen files.
*
********************************************************************************
*/

F_NODE          *ll_fd;   /*  Pointer to the active F_NODE. */
F_NODE          *ll_first;          /*  Pointer to the first F_NODE. */


/*
********************************************************************************
*
Function to allot memory for a F_NODE and initialising all the fields.
*
********************************************************************************
*/

create_fd (fd1)
F_NODE          **fd1;
{

        F_NODE          *fd;
        int         i;

        /*  Create a F_NODE.  */

        if ((fd = (F_NODE *) (malloc (sizeof (F_NODE)))) == NULL)
             erro_dump ("files.h: Error in memory allocation for fd.\n");


        /*  Initialize all the fields.  */

        fd->header = -2;
        fd->client_port = -2;
        fd->server_port = -2;
        fd->offset = -2;
        fd->index = -2;
        strcpy (fd->request,"!" );
        strcpy (fd->client,"!" );
        strcpy (fd->server,"!" );
        strcpy (fd->network,"!" );
        strcpy (fd->local,"!" );
```

```
          for (i = 0; i < 10; i++) strcpy (fd->mesg[i], "!");
          fd->prev = NULL;
          fd->next = NULL;

          *fd1 = fd;

}

/*
*******************************************************************************
*
*         Function to load relevant data to a fiven F_NODE from the
*         given Package.
*
*******************************************************************************
*/

ld_fd (pack, fd)
F_NODE          *fd;
Package pack;
{

          fd->header = pack.header;
          fd->client_port = FILE_CLI_PORT;
          fd->server_port = pack.present_port;
          fd->offset = 0;
          strcpy (fd->request, pack.request);
          strcpy (fd->client, SELF_NAME);
          strcpy (fd->server, pack.present);
          strcpy (fd->network, pack.network);
          strcpy (fd->local, pack.local);

}


/*
*******************************************************************************
*
*         Function to handle a transaction for write mode on client side.
*         User can write a message from a file or from the screen.
*         User can either append a message at the back or at the front.
*
*******************************************************************************
*/

wtransact (pack, request)
Package pack;
char      request[];
{
          F_NODE                    fd;
          char                      fpack[FILESIZE + 1], File[30];
          int                       sock, clilen, n,i;
          int                       FLAG = 0, flag;
          FILE                      *fp;
```

```
/*  Initialize a F_NODE and load the relevant data.  */

set_fd(&fd);
ld_fd (pack, &fd);

/*  Check whether to append at the front or back.  */

if (strcmp (request, "w") == 0)
        fd.offset = 0;                          /*  Front append.  */
else fd.offset = 1;                             /*  Back append.  */

fd.header = CLOSE;                              /* CLOSE is defined in Defs.h. */


/*  To write from a file or from the screen.
        Get the option from the user.  */

printf ("Include File?\n Y = 1, N = 0\n");
scanf ("%d", &flag);

/*  If the option is to write from a file.  */

if (flag){
        printf ("\nPlese Give The File Name:  ");
        scanf ("%s", File);
        if ((fp = fopen (File, "r")) == NULL)
                erro_dump ("Specified file not available.");
        }


for(;;){
        if (flag) {
                for ( i = 0; i < 10; i++) {

/*  Read the file.  Break from for loop at EOF.  */

                        if(fgets (fd.mesg[i], 99, fp) == NULL){
                                FLAG = 1;
                                strcpy (fd.mesg[i], "!");
                                break;
                                }
                }
        }

/*  If the input is given from the screen.  */

else {
        for ( i=0; i < 10; i++){
                printf ("To end the transaction end with '!q'\n");
                printf ("please give the string.\n");
                gets(fd.mesg[i]);
                 if(strcmp(fd.mesg[i], "!q") == 0){
                        FLAG = 1;
```

```
                                        strcpy (fd.mesg[i], "!");
                                        break;
                                        }
                                strcat(fd.mesg[i], "\n");
                                }
                        }

        fpack[0] = '\0';

        /* Unpack the F_NODE to message format.  */

        unpack_fd(fd, fpack);       /* Function unpack_fd is in fdpak.h.  */

        sock = server (FILE_CLI_PORT);

        /* Send one block of a file.  */

        client (fd.server, fd.server_port, fpack, strlen (fpack + 1));
        close (sock);

        /* If transaction is over  */

        if (FLAG == 1) break;
        fd.offset = 1;
        for (i = 0; i < 10; i++) strcpy (fd.mesg[i] , "");
        }
        if (flag) fclose (fp);
}

/*
*********************************************************************************
*
*       Function to edit a file.  Function copies the whole file and opens
*       local copy for edit.  When editing is over whole file is transferred
*       back to the server.
*
*********************************************************************************
*/

etransact (pack)
Package pack;
{
        FILE                    *fp;
        F_NODE                  *temp, fd;
        char                    fpack[FILESIZE + 1], file[50], local[60];
        int                     i;

        /* Read the whole file.  */

        strcpy (pack.request, "r");
        rtransact (pack);
```

```
while (ll_fd->offset != ERROR) {
        get_next();
        }

/* Copy the whole file into a .ed file. */

sprintf (file, "%s.ed", pack.local);
if ((fp = fopen (file, "w")) == NULL)
        erro_dump ("ERROR: In opening edit file.\n");

temp = ll_first;
while (temp != NULL) {
        for (i = 0; i < 10; i++)
                fprintf (fp, "%s", temp->mesg[i]);
        temp = temp->next;
        }

fprintf (fp, "\n");
fclose (fp);

/* Open the file for edit. */

sprintf (local, "vi %s", file);
system (local);

/* Copy the whole .ed file back to server. */

if ((fp = fopen (file, "r")) == NULL)
        erro_dump ("ERROR: In opening .edit file");

ll_first = NULL;

strcpy (pack.request, "w");
set_fd (&fd);
ld_fd (pack, &fd);
fd.offset = 0;

while (1) {
        for (i = 0; i < 10; i++) {
                if (fgets (fd.mesg[i], 99, fp) == NULL)
                        break;
                }

        fd.header = CLOSE;
        strcpy (fpack, "");
        unpack_fd (fd, fpack);
        if (i != 0)
                client (fd.server, fd.server_port, fpack, strlen (fpack) + 1);
        if (i < 10) break;
        set_fd (&fd);
        ld_fd (pack, &fd);
        fd.offset = 1;
        }
        /* Delete local .ed file. */

        sprintf (local, "rm %s", file);
        system (local);


}


/*
*******************************************************************************
*
*       Function to read the first page of a remote file.
*
*******************************************************************************
*/

rtransact (pack)
Package pack;
{

        F_NODE                  fd;
        char                    fpack[FILESIZE + 1], sfpack[FILESIZE + ACKSIZE + 1];
        char                    Host[50];
        int                     sock,clilen, n;

        set_fd (&fd);

        /* Create a linked list of pages of a remote file. */

        create_fd (&ll_fd);
        ld_fd(pack, &fd);
        fpack[0] = '\0';
        unpack_fd (fd, fpack);
        sock = server (FILE_CLI_PORT);

        /* Send a request to read the first page of a file. */

        client (fd.server,fd.server_port,fpack,strlen(fpack)+1);
        fpack[0] = '\0';
        sfpack[0] = '\0';
        Host[0] = '\0';

        /* Recieve the first page. */

        n = recvfrom(sock, sfpack, FILESIZE + ACKSIZE + 1, 0, NULL, &clilen);
        send_ack (sfpack, fpack, Host); /* Function send_ack is in client.h.*/
        pack_fd(&fd, fpack); /* function pack_fd is in fdpak.h. */
        fd.index = 0;
        close (sock);
        if (fd.header == ERROR)
                erro_dump ("ERROR: In opening file.\n");
        copy_pfd(ll_fd, fd);
        ll_first = ll_fd;
}

/*
*******************************************************************************
*
*       Function update the file location of a remote file.
*
*******************************************************************************
*/

update_MT (file, present, port)
char    file[];
char    present[];
int     port;
{

        FILE                    *rptr, *wptr;
        int                     junk;
        char                    line[200], buf[200], dummy[50];
        char                    request[50];

        sprintf (line, "%s %s %d ", file, present, port);

        /* Open the mounting table. */

        if ((rptr = fopen (".MT", "r")) == NULL)
                erro_dump ("Error in opening .MT file for update.\n");

        /* Open a temp file. */

        if ((wptr = fopen (".temp", "w")) == NULL)
                erro_dump ("Error in opening .MT file for update.\n");

        /* Read the whole mounting table. */

        while ((fgets (buf, 199, rptr)) != NULL) {
                sscanf (buf, "%s %s %d %s", dummy, request, &junk, request);

        /* Check for the referance made in the mounting table
                        for the required file. */

                if ((strcmp (dummy, file)) == 0){

        /* Update the present location. */

                        strcat (line, request);
                        fprintf (wptr, "%s\n", line);
                        }

                else fprintf (wptr, "%s", buf);
                fgets (buf, 199, rptr);
                fprintf (wptr, "%s", buf);
                fgets (buf, 199, rptr);
                fprintf (wptr, "%s", buf);
                }
        fclose (rptr);
        fclose (wptr);

        /*  Use in future the updated mounting table as mounting table.  */

        rename (".temp", ".MT");
}

/*
*******************************************************************************
*
*       Function to move the active F_NODE to its previous node
*       containing the previous page.
*
*******************************************************************************
*/

get_prev ()
{

        /*  If it is not the first node move to the previous node.  */

        if (ll_fd->prev == NULL) {
                printf ("No previous page.\n");
                return;
                }

        ll_fd = ll_fd->prev;
}

/*
*******************************************************************************
*
*       Function to get the next page of the remote file and move the
*       active node to the next page.
*
*******************************************************************************
*/

get_next()
{

        F_NODE          newfd, *dummy;
        int     sock, n, clilen;
        char    fpack[FILESIZE + 1], sfpack[FILESIZE + ACKSIZE + 1];
        char    Host[50];

        /*  If already EOF is reached  */
```

```
    if (ll_fd->offset == ERROR) {
            printf ("\n\nEnd of file Reached.\n");
            return;
            }

    /*  If next page is already cached  */

    if (ll_fd->next != NULL){
            printf ("\nMoving to next Local Page.\n");
            ll_fd = ll_fd->next;
            ll_fd->index = 0;
            return;
            }

    /*  If next page is not already cached  */

    printf ("\nGetting Next page from Remote Machine.\n");

    /*  Create a new node to hold the new page.  */

    create_fd (&dummy);
    ll_fd->next = dummy;;
    copy_fd(ll_fd, &newfd);
    fpack[0] = '\0';
    unpack_fd (newfd, fpack);
    sock = server (FILE_CLI_PORT);

    /*  Send a request for next page.  */

    client (newfd.server, newfd.server_port, fpack, strlen (fpack) + 1);
    fpack[0] = '\0';
    sfpack[0] = '\0';
    Host[0] = '\0';

    /*  Recieve the next page.  */

    n = recvfrom (sock, sfpack, FILESIZE + ACKSIZE + 1, 0, NULL, &clilen);
    send_ack (sfpack, fpack, Host);
    close (sock);
    pack_fd (&newfd, fpack);
    newfd.index = 0;

    /*  Move the active node to the next node.  */

    ll_fd->next->prev = ll_fd;
    ll_fd= ll_fd->next;
    copy_pfd (ll_fd, newfd);
}
```

```
/*
********************************************************************************
*
*       Function to copy relevant data from a given F_NODE to a new F_NODE.
*
********************************************************************************
*/

copy_fd(fd, fd1)
F_NODE  *fd, *fd1;
{

        fd1->header = fd->header;
        fd1->client_port = fd->client_port;
        fd1->server_port = fd->server_port;
        fd1->offset = fd->offset;
        fd1->index = fd->index;
        strcpy (fd1->request, fd->request);
        strcpy (fd1->client, fd->client);
        strcpy (fd1->server, fd->server);
        strcpy (fd1->network, fd->network);
        strcpy (fd1->local, fd->local);
}

/*
********************************************************************************
*
*       Function to replicate a F_NODE.
*
********************************************************************************
*/

copy_pfd(tfd,ffd)
F_NODE          *tfd;
F_NODE          ffd;
{

        int                             i;

        tfd->header = ffd.header;
        tfd->client_port = ffd.client_port;
        tfd->server_port = ffd.server_port;
        tfd->offset = ffd.offset;
        tfd->index = ffd.index;
        strcpy (tfd->request, ffd.request);
        strcpy (tfd->client, ffd.client);
        strcpy (tfd->server, ffd.server);
        strcpy (tfd->network, ffd.network);
        strcpy (tfd->local, ffd.local);

        for (i = 0; i < 10; i++) strcpy (tfd->mesg[i], ffd.mesg[i]);
}
```

```
/*
*******************************************************************************
*
*       Function to destroy a linked list.
*
*******************************************************************************
*/

wind_up()
{

        F_NODE                  *dummy;

         while(ll_first != NULL) {
                dummy = ll_first;
                ll_first = ll_first->next;
                free(dummy);
                }
free (ll_fd);
}
```

```
/*
*********************************************************************************
*
*       File:   server.c
*       Include files:    mount.h, s.h, c.h.
*       Functions:  transact, del_rmt, ser_present, ser_update, fcopy, ser_fm,
*                   sgets, ser_um, ser_sess, serach_RMMT, ser_mt, ser_reg,
*                   ser_ad, ld_um_pack.
*
*       This file contains all the functions required for processing the
*       requests from the clients.
*
*********************************************************************************
*/

/* Header files. */

#include <stdlib.h>
#include "mount.h"
#include "s.h"
#include "c.h"

Package *search_RMMT();

/*
*********************************************************************************
*
*       This program has code for address server. It binds to a well known port.
*       It gets a password from the user. The main program creates a child
*       and exits. The child process recieves messages. The child process
*       creates a child to process each message. This program also maintains
*       mounting tables and location table which are hidden from the user.
*
*********************************************************************************
*/

main()
{
        int             sock, child, proc;
        int             clilen, n;
        char            mesg[FILESIZE + ACKSIZE + 1];
        char            cpack[FILESIZE + 1], Host[80];
        char            dummy[PASS_MAX];


        /* Bind a socket to server port. */

        sock = server (SERV_UDP_PORT);

        /* Register a password for the server. */

        fprintf (stderr, "Please Register The Password Now: ");
        strcpy (dummy, "");
        fflush(stdin);
        strcpy (PASWORD, getpass(dummy));

        /*  Create a daemon server.  */

        child = fork();

        if (child == ERROR){
                close (sock);
                erro_dump ("Can't create child.\n");
                }

        if (child == 0) {
        printf ("Server is created.\n");

        for (;;) {

                /*  Receive a message.  */

                strcpy (cpack, "");
                strcpy (mesg, "");
                strcpy (Host, "");

                n = recvfrom(sock, mesg, FILESIZE + ACKSIZE + 1,0,NULL,&clilen);

                if (n < 0)
                erro_dump ("Error:  In receiving messge.\n");

                /*  Send an acknowledgement.  */

                send_ack(mesg, cpack, Host);
                proc = fork();
                if (proc == ERROR) printf ("Can't process a request.\n");
                if (proc == 0) {
                        transact (cpack, Host);
                        exit (0);
                        }
                }
        }
}


/*
**********************************************************************************
*
*       Function to process a message recieved in main.
*
**********************************************************************************
*/

transact (cpack, Host)
char    cpack[];
char    Host[];
{
```

```
Package                 *packi, pack;
int                     n;
char                    mesg[FILESIZE + 1];
char                    ack_addr[80];
char                    HostName[50], Local[50];
int                     Port, ack_port;
int                     sock1, clilen;


set_pack (&pack);


strcpy (mesg, cpack);

/*  If the message is of Package type */

if ((cpack[0] != '8') || (cpack[0] != '9'))
        pak_pack (&pack, mesg);
else
        sscanf (cpack, "%d", &pack.header);

/*  process the request.  */

switch (pack.header) {
        case ADVERTISE:                     /* Recieving an advertisement.  */
                ser_ad (&pack);
                break;

        case REGISTER :                     /* Recieving a register request.  */
                if(ser_reg (&pack) != SUCCESS)
                        prmesg("Error in registering", pack.local);

                strcpy (cpack, "");
                unpak_pack(pack, cpack);
                client (pack.toa, pack.toport, cpack, strlen (cpack)+1);
                break;

        case PERMITTED :                    /*  Getting a permission in response
                                            to registration request.  */
                ser_mt (&pack);
                break;

        case SESSION :                      /*  Session requests from clients.  */
                ser_sess(&pack);
                strcpy (mesg, "");
                unpak_pack(pack, mesg);
                client (Host,pack.toport,mesg,strlen(mesg)+1);
                break;

        case USER_MOBIL :           /*  Request from owner from remote machine.  */
                ser_um (&pack);
                strcpy (HostName, pack.toa);
                strcpy (mesg, "");
                unpak_pack (pack, mesg);
                client (Host,pack.toport,mesg,strlen(mesg)+1);
                break;

        case FILE_MOBIL :                        /* Recieving a file. */
                strcpy (Local, "");
                strcpy (HostName, "");
                Port = -1;
                if((ser_fm (cpack, Local, HostName, &Port)) == SUCCESS){
                        sock1 = server (FILE_MOBIL_SERV_PORT);
                        sprintf (cpack, "%d %d ",
                                        FILE_MOBIL, FILE_MOBIL_SERV_PORT);
                        client (HostName, Port, cpack, strlen (cpack) + 1);
                        n = 1;
                        printf ("Recieving File. .");
                        while (n != ERROR) {
                                strcpy (cpack, "");
                                strcpy (mesg, "");
                                strcpy (Host, "");
                                n = recvfrom(sock1, mesg, FILESIZE + 1,
                                                0, NULL, &clilen);
                                send_ack (mesg, cpack, Host);
                                n = fcopy (cpack, Local);
                                printf (" .");
                                }
                        printf ("\nFile Recieved.\n");
                        close (sock1);
                        }
                break;

        case UPDATE_PRESENT :                    /* Update the location table. */
                ser_update (cpack);
                break;

        case SERV_PRESENT :                      /* Update a client about the present
                                                    location of a file. */
                ser_present (cpack, HostName, &Port);
                client (HostName, Port, cpack, strlen (cpack) + 1);
                break;

        case DELETE :                            /* Delete a client. */
                del_rmt (cpack);
                break;

        default :
                erro_dump ("Ambiguous Command.\n");
                break;
        }
}
```

```
/*
********************************************************************************
*
*       Function to delete an access given to a client for a file.
*
********************************************************************************
*/

del_rmt (cpack)
char    cpack[];
{
        FILE                    *rptr, *wptr;
        char                    line[100], nline [100], network[100], oline [100], dumch[30];
        char                    log [100], logcli[100], clien [100], cli[100];
        int                     dummy;

        if((rptr = fopen (".RMMT", "r")) == NULL)
                return;
        if ((wptr = fopen (".temp", "w")) == NULL)
                return;

        sscanf (cpack, "%d %s %s %s", &dummy, network, clien, logcli);

        /* Search for the mounitng points and delete them. */

        while (fgets (line, 99, rptr) != NULL) {
                fgets (oline, 99, rptr);
                fgets (nline, 99, rptr);
                if(nline[strlen(nline) - 1] == '\n')
                        nline[strlen (nline) - 1] = '\0';
                if (strcmp (nline, network) == 0) {
                        sscanf (line, "%s %s %d %s", dumch, cli, &dummy, log);
                        if ((strcmp (cli, clien) == 0) &&
                                (strcmp (log, logcli) == 0));
                        else {
                                fprintf (wptr, "%s", line);
                                fprintf (wptr, "%s", oline);
                                fprintf (wptr, "%s\n", nline);
                                }
                        }
                else {
                        fprintf (wptr, "%s", line);
                        fprintf (wptr, "%s", oline);
                        fprintf (wptr, "%s\n", nline);
                        }
                }

        fclose (rptr);
        fclose (wptr);

        rename (".temp", ".RMMT");
}
```

```
/*
*********************************************************************************
*
*         Function to update the clients about the location of the file created
*         by the local machine.
*
*********************************************************************************
*/

ser_present (cpack, toa, Port)
char   cpack[];
char   toa[];
int    *Port;
{
          char                      dummy[100], network[100], line [201];
          int                       header, port;
          FILE                      *fp;

          sscanf (cpack, "%d %s %s %d",&header, dummy, toa, &port);
          *Port = port;

          /* Update from the location table. */

          if ((fp = fopen (".ofile", "r")) == NULL)
               erro_dump ("Can't open .ofile.\n");

          while ((fgets (line, 200, fp)) != NULL) {
                    sscanf (line, "%s", network);
                    if ((strcmp (network, dummy)) == 0)
                    sprintf (cpack, "%d %s", SERV_PRESENT, line);
                    }

          fclose (fp);
}

/*
*********************************************************************************
*
*         Function to update the location of the file which has been created
*         and then have been transferred other machines.
*
*********************************************************************************
*/

ser_update (cpack)
char      cpack[];
{
          int                       i;
          char                      dummy[3];
          FILE                      *fp;
```

```
        /*      Update the location table.  */

        if ((fp = fopen (".ofile", "a")) == NULL)
                erro_dump ("Can't update the location table.\n");

        sscanf (cpack, "%s %n", dummy, &i);
        fprintf (fp, "%s\n", cpack + i);
        fclose(fp);
}

/*
*******************************************************************************
*
*       Function to copy a block of file.
*
*******************************************************************************
*/

fcopy (cpack, Local)
char   cpack[];
char   Local[];
{
        FILE                    *fp;
        char                    dummy[20];
        int                     i,j;

        if ((fp = fopen (Local, "a")) == NULL){
            sprintf (cpack,"Can't open %s for writing.\n", Local);
            erro_dump (cpack);
            }

        fseek (fp, 0, 2);

        sscanf (cpack, "%s %n", dummy, &i);
        sscanf (cpack + i, "%s %n", dummy, &j);
        i = i + j;
        sscanf (cpack + i, "%s %n", dummy, &j);
        i = i + j;

        fprintf (fp, "%s", cpack + i);
        fclose(fp);
        if (strcmp (dummy, "-1") == 0) return (ERROR);
        return (SUCCESS);
}

/*
*******************************************************************************
*
*       Function to create new mounting points when a new shared file has
*       transferred to the server.
*
*******************************************************************************
*/
```

```
ser_fm (cpack, Local, HostName, Port)
char    cpack[];
char    Local[];
char    HostName[];
int     *Port;
{
        char                    network[100], owner[50];
        char                    line[100], oline[100], dummy[125];
        int                     i, j, port;
        FILE                    *fp;


        i = 0;
        sscanf (cpack, "%d %n", &j, &i);
        sscanf (cpack + i, "%s %n", HostName, &j);
        i = i + j;
        sscanf (cpack + i, "%d %n",&port, &j);
        i = i + j;
        *Port = port;
        sscanf (cpack + i, "%s %n", Local, &j);
        i = i + j;
        sscanf (cpack + i, "%s %n", network, &j);
        i = i + j;

        /* Get file creator's name. */

        sscanf (cpack + i, "%s %n", oline, &j);
        i = i + j;
        sscanf (cpack + i, "%s %n", dummy, &j);
        i = i + j;

        strcat (oline, " ");
        strcat (oline, dummy);

        if ((fp = fopen (".RMMT", "a")) == NULL)
                erro_dump ("Error in opening .RMMT file.\n");

        /* Get all client's names. */

        while (sgets (cpack, line, &i) != -1){
                strcpy (dummy, Local);
                strcat (dummy, " ");
                strcat (dummy, line);
                fprintf (fp, "%s\n", dummy);
                fprintf (fp, "%s\n", oline);
                fprintf (fp, "%s\n", network);
                }


        fclose (fp);
        return (SUCCESS);
}

/*
*********************************************************************************
*
*        This function gets a line terminated with a '\n' from a given string.
*
*********************************************************************************
*/

sgets (cpack, line, i)
char     cpack[], line[];
int      *i;
{
         int                          j, k;

         j = *i;

         for (k = 0 ; (cpack[j] != '\n') && (cpack[j] != '\0'); j++, k++)
         line[k] = cpack[j];
         line[k] = '\0';
         *i = j + 1;
         if (cpack[j] == '\0')
                  return (ERROR);
         return (SUCCESS);
}

/*
*********************************************************************************
*
*        Function to compare the password provided by the user with the
*        password he has registered and call the function which handles
*        the user mobility transactions.
*
*********************************************************************************
*/

ser_um (pack)
Package *pack;
{

         if ((strcmp (pack->passwd, PASWORD)) != 0) {
                  pack->header = DENY;
                  return (ERROR);
                  }

         ld_um_pack (pack->rlocal, pack->request, &pack);

}
```

```
/*
*******************************************************************************
*
*       Function to check the access acredations.
*
*******************************************************************************
*/

ser_sess (pack)
Package *pack;
{
        Package *temp;
        char    tpermit[4], ppermit[4];

        /* Get the mounting point. */

        temp = search_RMMT (pack->network, pack->toa, pack->login);
                if(temp->header == NOFILE) {
                        pack->header = NOFILE;
                        return (SUCCESS);
                        }

        /* Check the file requested and the client who made the request. */

                if((strcmp(pack->network, temp->network) == 0) &&
                        (strcmp (pack->login, temp->login) == 0)){
                        strcpy (ppermit, pack->request);

        /* Allow append and edit requests also for write permission. */

                        if ((strcmp (ppermit, "a") == 0)
                                || (strcmp (ppermit, "e") == 0))
                                        strcpy (ppermit, "w");
                        strcpy (tpermit, temp->permissions);

                /* Accredition checking for 'READ' request. */

                        if ((ppermit[0] == 'r') && (tpermit[0] == 'r')){
                                pack->header = ACCESS;
                                strcpy(pack->permissions, "r");
                                pack->present_port = FILE_SERV_PORT;
                                strcpy (pack->local, temp->local);
                                printf("Success in accessing.\n\n");
                                return (SUCCESS);
                                }

                /* Accredition checking for 'WRITE' request. */

                        else if ((ppermit[0] == 'w') && ((tpermit[0] == 'w') ||
                                (tpermit[1] == 'w'))) {
                                pack->header = ACCESS;
                                pack->present_port = FILE_SERV_PORT;
                                strcpy (pack->permissions, "w");
```

```
                                        strcpy (pack->local, temp->local);

                                        printf("Success in accessing for write.\n");
                                        return (SUCCESS);
                                        }

                  /* Deny access for an illegitimate access request.  */

                                else {
                                        pack->header = DENY;
                                        strcpy(pack->permissions, "");
                                        printf("Access permission denied.\n");
                                        return (ERROR);
                                        }
                                }

        /* If the client is not a legitimate user DENY access.  */

        else {
                pack->header = DENY;
                strcpy(pack->permissions, NULL);
                printf("access permission denied.\n");
                return (ERROR);
                }
}

/*
*******************************************************************************
*
*       Function to get the mounting point of the requested file.
*
*******************************************************************************
*/

Package *search_RMMT (network, toa, Login)
char    network[];
char    toa[];
char    Login[];
{
        Package                 temp;
        char                    line[200], pline[200], nline[200];
        int                     dummy;
        FILE                    *fp;

        if((fp = fopen(".RMMT", "r")) == NULL) {
                temp.header = NOFILE;
                return (&temp);
                }


        /* Search for the mounting point.  */

        while ((fgets (line, 200, fp)) != NULL) {
                fgets (pline, 200, fp);
                fgets (nline, 200, fp);
                if (nline[strlen (nline) - 1] == '\n')
                        nline[strlen (nline) - 1] = '\0';
                if ((strcmp (nline, network)) == 0){
                        set_pack(&temp);
                        if (line[strlen (line) - 1] == '\n')
                                line[strlen (line) - 1] = '\0';

        /* Get all the info available about the mounting point. */

                        sscanf (line, "%s %s %d %s %s",
                                        temp.local, temp.toa, &dummy,
                                        temp.login, temp.permissions);
                        if ((strcmp (toa, temp.toa) == 0) &&
                                (strcmp (Login, temp.login) == 0)){
                                fclose (fp);
                                strcpy (temp.network, nline);
                                return (&temp);
                                }
                        }
                }
        fclose (fp);
        temp.header = NOFILE;
        return (&temp);
}


/*
*************************************************************************************
*
*       Function to add a mounting point to the mounting table.
*
*************************************************************************************
*/

ser_mt (pack)
Package *pack;
{
        FILE                    *ptr;

        if((ptr = fopen (".MT", "a")) == NULL)
        erro_dump("Error in updating mounting table. Mounting aborted.\n");

        fprintf (ptr, "%s %s %d %s\n", pack->rlocal, pack->present,
                                pack->present_port, pack->permissions);
        fprintf (ptr, "%s %d\n", pack->owner, pack->owner_port);
        fprintf (ptr, "%s\n", pack->network);

        fclose (ptr);
        return (SUCCESS);
}
```

```
/*
*******************************************************************************
*
*        Function to add a mounting point *server side* for a file to a client.
*
*******************************************************************************
*/

ser_reg (pack)
Package *pack;
{

        FILE                    *fp;


        pack->header = PERMITED;

        if ((fp = fopen (".RMMT", "a")) == NULL)
                prmesg("s.c:", "Error in opening .RMMT\n");

        else {
                fprintf (fp, "%s %s %d %s %s\n",
                                pack->local, pack->toa, pack->toport,
                                pack->login, pack->permissions);
                fprintf (fp, "%s %d\n", pack->owner, pack->owner_port);
                fprintf (fp, "%s\n", pack->network);
                fclose (fp);
                }

        return (SUCCESS);
}

/*
*******************************************************************************
*
*        Function to save all the advertisements recieved.
*
*******************************************************************************
*/

ser_ad (pack)
Package *pack;
{

        FILE                    *ptr;


        if ((ptr = fopen (".regs", "a")) == NULL)
                erro_dump ("Server:  Can't write messages.\n");

        fprintf (ptr, "%d %s %s %s %d %d %s\n", pack->header, pack->owner,
                        pack->local, pack->present, pack->owner_port,
                        pack->present_port, pack->permissions);

        fprintf (ptr, "%s\n", pack->network);
        fclose (ptr);

}

/*
*********************************************************************************
*
*       Function to handle request user mobility request.
*
*********************************************************************************
*/

ld_um_pack(file, request, um_pack)
char    file[];
char    request[];
Package **um_pack;
{

        FILE                    *ptr, *fptr;
        char                    buf[91], cmd[15], rlocal[30];
        int                     j,i = 0;
        Package                 *pack;

        /* Get the mounting point. */

        pack = *um_pack;
        if((ptr = fopen(".MT", "r")) == NULL)
                erro_dump("files.h: Error in opening .MT");

        if ((fgets (buf, 90, ptr)) == NULL){
                pack->header = NOFILE;
                return (0);
                }

        sscanf(buf, "%s %n", rlocal, &j);
        while (strcmp(rlocal, file) != 0) {
                fgets(buf, 90, ptr);
                fgets(buf, 90, ptr);
                if (fgets(buf, 90, ptr) == NULL){
                        pack->header = NOFILE;
                        return (0);
                        }
                sscanf(buf, "%s %n", rlocal, &j);
                }

        /* Load the relevant info about the file. */

        i = i+j;
        sscanf(buf + i, "%s %n", pack->present, &j);
        i = i+j;
        sscanf (buf + i, "%d %n", &pack->present_port, &j);
        i = i+j;
        sscanf (buf + i, "%s", pack->permissions);

        fgets (buf, 90, ptr);
        sscanf (buf, "%s %d", pack->owner, &pack->owner_port);

        fgets (buf, 90, ptr);
        strcpy(pack->network, buf);
        if (pack->network[strlen (pack->network) - 1] == '\n')
                pack->network[strlen(pack->network) -1] = '\0';

        pack->header = SESSION;

        fclose (ptr);

        *um_pack = pack;
}
```

```
/*
*******************************************************************************
*
*        File:  FileServer.c
*        Include files:  mount.h, c.h, s.h, fdpak.h.
*        Functions:  Get_file, Put_file, serv, transact.
*
*        This file contains file handling functions.
*
*******************************************************************************
*


#include "mount.h"
#include "c.h"
#include "s.h"
#include "fdpak.h"


/*
*******************************************************************************
*
*        Function to read a block of file.
*
*******************************************************************************
*/

Get_file (fd)
F_NODE          *fd;
{

        FILE    *fp;
        int     i = 0;

        fp = NULL;
        if ((fp = fopen (fd->local, fd->request)) == NULL){
                fd->header = ERROR;
                return;
                }

        fseek (fp, fd->offset, 0);

        while ((fgets (fd->mesg[i], 99, fp) != NULL) && (i < 9)) i++;

        if (i < 9) fd->offset = -1;

        else fd->offset = ftell(fp);

        fclose (fp);

}
```

```
/*
*********************************************************************************
*
*       Function to write to a file.
*
*********************************************************************************
*/

Put_file (fd)
F_NODE          fd;
{

        FILE                    *fp;
        int                     i;

        if (fd.offset == 1)
                if ((fp = fopen (fd.local, "a")) == NULL)
                                return(ERROR);
        if (fd.offset == 0)
                if ((fp = fopen (fd.local, "w")) == NULL)
                        return (ERROR);

        for (i = 0; i < 10; i++) {
                if (strcmp (fd.mesg[i], "!") != 0)
                        fprintf (fp, "%s", fd.mesg[i]);
                else break;
                }
        fclose (fp);

        return (SUCCESS);

}

/*
*********************************************************************************
*
*       This program creates a file server.  It binds a socket.  It creates
*       a child to recieve messages.  The child create a child to process all
*       the requests made by the client.  Typical requests are read a block
*       of file, write to a file.
*
*********************************************************************************
*/

main()
{
        int         child, sock;

        /* Bind the server to a well known port. */

        sock = server (FILE_SERV_PORT);
        /* Run the server in the background. */

        child = fork();

        if (child == ERROR) erro_dump ("Can't run the file server.\n");

        if (child == 0){
                serv (sock);
                exit(0);
                }
}


/*
*******************************************************************************
*
*       This function recieves a message and creates a child to handle
*       the messge.
*
*******************************************************************************
*/

serv (sock)
int     sock;
{

        F_NODE                  fd;
        int                     n, clilen, child;
        struct sockaddr         cli_addr;
        char                    sfpack[FILESIZE + ACKSIZE + 1], fpack[FILESIZE + 1];
        char                    Host[50];

        clilen = sizeof (cli_addr);
        set_fd(&fd);

        for (;;) {

            /* Reset the variables. */

            fpack[0] = '\0';
            sfpack[0] = '\0';
            Host[0] = '\0';
            set_fd (&fd);

            /* Recieve a message. */

            n = recvfrom (sock, sfpack, FILESIZE + ACKSIZE + 1,
                            0, &cli_addr, &clilen);

            /* Send an acknowledgement. */

            send_ack (sfpack, fpack, Host);

            /* Create a child to process the message. */
            child = fork();
            if (child == ERROR) printf ("Can't process a message.\n");
            if (child == 0) {
                transact (fpack);
                exit(0);
                }
            }
}

/*
*********************************************************************************
*
*       Function to process the message.
*
*********************************************************************************
*/

transact (fpack)
char    fpack[];
{

        F_NODE                  fd;

        pack_fd (&fd, fpack);
        switch (fd.header) {

                /* Request to read a block of file. */

                case ACCESS:
                        Get_file (&fd);
                        fpack[0] = '\0';
                        unpack_fd (fd, fpack);
                        client(fd.client, fd.client_port, fpack, strlen (fpack) + 1);
                        break;

                /* Request to write a file. */

                case CLOSE :
                        Put_file (fd);
                        break;
                        }
}
```

```
/*
********************************************************************************
*
*       File:  advertise.c
*       Include files:  mount.h, c.h, s.h.
*       Functions : get_ld_pak, get_net_name.
*
*       This file contains code for advertising a file.
*
********************************************************************************
*/

/* Header files. */

#include "mount.h"
#include "s.h"
#include "c.h"

/*
********************************************************************************
*
*       This program advertises a file for sharing.
*       It gets name of the file to be shared, the access permissions
*       provided, and to what user from the user.  Then, the program checks
*       whether the file is present in the local machine or not.  Then,
*       if the file is already shared with some other client it uses the
*       same network name.  Hence, guarenteeing a unique network name for
*       each file.  Also, if the client already have the access to the same
*       file, this program prompts the user.  If necessary, this programs
*       creates a network name of a file, by attaching location, path of the
*       file and the time of creation, to the local name of the file.  Finally,
*       this program sends a message to the client about the advertisement.
*
********************************************************************************
*/

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

        char                HostName[100];
        char                cpack[PACKSIZE], cack[ACKSIZE + 1];
        Package             pack;
        int                 PortNum, num, n, clilen;

        set_pack (&pack);
        pack.toport = 6464;                              /* Address server port. */

        /* Check for wrong modes. */

        if (argc == 5) {
                strcpy (pack.local, argv[1]);
        if ((strcmp (argv[2], "r") != 0) &&
              (strcmp (argv[2], "w") != 0) &&
              (strcmp (argv[2], "rw") != 0)) {
              printf ("Wrong mode : %s\n", argv[2]);
              erro_dump ("Please use 'r' or 'w' or 'rw' modes.\n");
              }
        strcpy (pack.permissions, argv[2]);
        strcpy (pack.toa, argv[3]);
        strcpy (pack.login, argv[4]);
        }

 /* If required, get the data interactively.  */

 else if (argc == 1) {
         printf ("\n Please give the FILE name.\n");
         fflush(stdin);
         scanf ("%s",pack.local);

         printf ("\nPlease give the ACCESS permissions.\n");
         printf ("\nUse following notations for access permissions.\n");
         printf ("\n\n For READ ONLY use r.\n");
         printf ("\n For READ & WRITE use w.\n");

         fflush(stdin);
         scanf("%s", pack.permissions);
         if ((strcmp (pack.permissions, "r") != 0) &&
                 (strcmp (pack.permissions, "w") != 0) &&
                         (strcmp (pack.permissions, "rw") != 0))
                 erro_dump ("Select proper mode.\n");

         printf ("Please give the Client's address.\n");
         fflush(stdin);
         scanf ("%s", pack.toa);
         printf ("Please give the Client's login.\n");
         fflush(stdin);
         scanf ("%s", pack.login);
         }

else erro_dump
         ("FORMAT: 'File' 'Mode' 'Client's Address' 'Client's Login'\n");

/* load the Package.  */

if (get_ld_pk(&pack) != ERROR){
         unpak_pack(pack, cpack);

         /* Send message to the client.  */

         client (pack.toa, pack.toport, cpack, strlen (cpack) + 1);
         }
```

```
/*
******************************************************************************
*
*       Function load a Package with the relevant data about the file.
*       This function also checks whether the client has access to the
*       file being advertised.  If so, it prompts the user.  If the file is
*       already shared by some other client, same network name is used.
*       Otherwise, a new network name for the file is created.
*
******************************************************************************
*/

int     get_ld_pk (pack)
Package *pack;
{
        char                    cmd[100], file[20];
        FILE                    *ptr, *rptr;
        int                     n = 101,i, j, FLAG = 0;
        char                    buf[100], line[100];
        long                    offset = -1;

        sprintf (line, "%s %s %d %s %s\n", pack->local, pack->toa,
                        pack->toport, pack->login,  pack->permissions);

        /*  Check the mount table for network name of the file.  */

        rptr = NULL;
        if ((rptr = fopen (".RMMT", "r")) != NULL){
                while ((fgets (buf, 100, rptr)) != NULL) {
                        sscanf (buf, "%s %n", file, &j);
                        if ((strcmp (file, pack->local)) != 0) {
                                fgets(buf, 100, rptr);
                                fgets(buf, 100, rptr);
                                }

                /* If the client already has the access  */

                        else if ((strcmp (file, pack->local)) == 0) {
                                if ((strcmp (line, buf)) == 0) {
                                sprintf (line,
                                "%s %s already have access to %s.\n",
                                        pack->toa, pack->login, pack->local);
                                        erro_dump (line);
                                        }


                /* If file has a network name use the same.  */

                                if (!FLAG) {
                                        strcpy (pack->present, SELF_NAME);
                                        pack->present_port = SERV_UDP_PORT;

                                        fgets (buf, 100, rptr);
                                        sscanf (buf, "%s %d",
                                                pack->owner, &pack->owner_port);
                                        fgets(pack->network, 80, rptr);
                                        pack->header = ADVERTISE;
                                        FLAG = 1;
                                        }
                                return (SUCCESS);
                                }
                        }
                }

        /*  Check whether the file is resident or not.  */

        strcpy (cmd, "ls ");
        strcat (cmd, pack->local);
        if((ptr = popen (cmd, "r")) == NULL)
                erro_dump("Error: In checking residense of the file. \n");


        /* If file is not present in the machine. */

        fgets (buf,n,ptr);
        buf[strlen(buf) - 1] = '\0';
        if(strcmp (buf, pack->local) != 0)
                return (ERROR);


        strcpy (pack->owner, SELF_NAME);
        strcpy (pack->present, SELF_NAME);
        pack->present_port = SERV_UDP_PORT;
        pack->owner_port = SERV_UDP_PORT;

        /* If no network name exist create one.  */

        get_net_name (&pack);


        pack->header = ADVERTISE;

        if(pack->network[strlen(pack->network) - 1] == '\n')
                pack->network[strlen(pack->network) - 1]  = '\0';

        return (SUCCESS);
}
```

```
/*
********************************************************************************
*
*       Function to generate a network name for a given file.
*
********************************************************************************
*/


get_net_name(pack)
Package **pack;
{

        char    buf[100], cmd[40];
        FILE    *ptr;
        int     n = 101;


        strcpy ((*pack)->network, (*pack)->local);
        strcat ((*pack)->network, ":");

        /* Attach path of the file. */

        strcpy(cmd, "pwd");
        if((ptr = popen (cmd, "r")) != NULL)
                fgets (buf, n, ptr);
        else erro_dump ("Error in advertising.\n");
        buf[strlen(buf) - 1] = '\0';
        pclose (ptr);

        strcat ((*pack)->network, buf);

        /*  Attach the time of creation.  */

        strcpy (cmd, "date '+%H:%M:%S:%m:%d:%y:%Z'");
        if((ptr = popen (cmd, "r")) != NULL)
                fgets (buf, n,ptr);
        else erro_dump ("Error in advertising.\n");
        buf[strlen(buf) - 1] = '\0';
        pclose (ptr);

        strcat ((*pack)->network, buf);

}
```

```
/*
*********************************************************************************
*
*       File:      register.c
*       Include files : mount.h, s.h, c.h.
*       Functions : regi.
*
*       This file contains all the functions required for sending a
*       registration request.
*
*********************************************************************************
*/

/* List of header files. */

#include <stdio.h>
#include "mount.h"                          /* Package related functions. */
#include "s.h"                              /* Server related functions. */
#include "c.h"                              /* Client related functions. */

/*
*********************************************************************************
*
*       This program handles registration transaction. First, it creates
*       a list of requests waiting for registration. Then, user is allowed
*       to opt for any of valid requests. If the option is for registration
*       user is requested to name the file locally. Then a request for
*       registration is made to server. User can delete a request without
*       registering it. Program before exiting, saves all the requests not
*       registered or deleted. Same requests can be processed along with
*       the new requests if any later when the program is executed again.
*
*********************************************************************************
*/

main ()
{
        FILE                *ptr;
        char                buf[100], temp[3];
        Package             *pack, *first, *last, pck;
        int                 i = 0, j, k, ind;
        char                cpack[PACKSIZE];

        first = NULL;
        last =  NULL;

        /* Handle registration requests if any. */

        if ((ptr = fopen (".regs", "r")) == NULL)
                erro_dump ("No new requests.\n");


        /* Create a list of all the registration requests. */
        while ((fgets (buf, 80, ptr)) != NULL){
                pack = create_package();

                sscanf (buf, "%d %n", &pack->header, &j);
                sscanf (buf + j, "%s %n", pack->owner, &k);
                j = j + k;
                sscanf (buf + j, "%s %n", pack->local, &k);
                j = j + k;
                sscanf (buf + j, "%s %n", pack->present, &k);
                j = j + k;
                sscanf (buf + j, "%d %n", &pack->owner_port, &k);
                j = j + k;
                sscanf (buf + j, "%d %n", &pack->present_port, &k);
                j = j + k;
                sscanf (buf + j, "%s %n", pack->permissions, &k);

                fgets (buf, 80, ptr);
                strcpy (pack->network, buf);

                if (first == NULL) {
                        first = pack;
                        last = pack;
                        }

                else {
                        last->next = pack;
                        last = pack;
                        }

        /* Count the number of registration requests. */

                i++;
                printf(".");
                }

        if ( i == 0) erro_dump ("No new requests.\n");

        printf("Totally  %d registration requests.\n\n", i);
        fclose (ptr);

        /* Process user's requests. */

    for (;;) {
        printf("->  ");
        gets (buf);

        /* Error checks for proper requests. */

        sscanf (buf, "%s %n", temp, &j);
        sscanf (buf + j, "%d", &ind);
        if ((strcmp (temp, "q") != 0) && (strcmp (temp, "r") != 0) &&
          (strcmp (temp, "t") != 0) && (strcmp (temp, "d") != 0) &&
                (strcmp (temp, "h") != 0))
                printf ("Invalid Command.\nType 'h' for help.\n");
```

```
else if ((ind > i) && (strcmp (temp, "q" ) != 0) &&
                    (strcmp (temp, "h") != 0))
                     printf ("Not that many number of requests.\n");
else if ((ind < 1) && (strcmp (temp, "q" ) != 0) &&
                    (strcmp (temp, "h") != 0))
                     printf ("Invalid request number.\n");
else {

/* If the option is quit, save the remaining registration requests. */

pack = first;
if(strcmp(temp, "q") == 0) {
          if ((ptr = fopen(".regs", "w")) == NULL)
                    erro_dump ("Error: in registration.\n");

          for (ind = 0; ind < i; ind++){
                    if (pack->header != ERROR){
                    fprintf(ptr,"%d %s %s %s %d %d %s\n", pack->header, pack->owner,
                    pack->local, pack->present, pack->owner_port,
                    pack->present_port, pack->permissions);
                    fprintf (ptr,"%s", pack->network);
                    }
                    pack = pack->next;
                    }
          fclose (ptr);
          break;
          }

/* If the option is for registration  */

else if (strcmp(temp, "r") == 0) {
          for (j = 1; j < ind; j++) pack = pack->next;
          pck = *pack;

/*  Get all the info about the request.  */

          if (regi(&pck) == SUCCESS) {
          if (pck.network[strlen(pck.network) -1] == '\n')
                    pck.network[strlen(pck.network) - 1] = '\0';
          cpack[0] = '\0';
          unpak_pack (pck, cpack);

/*  Send the registration request to the server.  */

          client(pck.present,pck.present_port,cpack,strlen (cpack) + 1);
                    pack->header = ERROR;
                    printf("SUCCESS\n");
                    }
          else printf("Error: Request is either already deleted or registered.\n");
          }
```

```
        /* If the option is for deletion of a request. */

        else if (strcmp (temp, "d") == 0) {
                for (j = 1; j < ind; j++) pack = pack->next;
                pack->header = ERROR;
                }
        else if (strcmp (temp, "t") == 0) {
                for (j = 1; j < ind; j++) pack = pack->next;
                if (pack->header != ERROR) {
                        printf ("File Name:         %s\n\n", pack->local);
                        printf ("Permissions:       %s\n\n", pack->permissions);
                        printf ("From:              %s\n\n", pack->present);
                        }
                else printf ("Request is deleted.\n");
                }

        /* List all the valid options. */

        else if (strcmp (temp, "h") == 0) {
                        system ("clear");
                        printf ("\n\n\n\n");
                        printf("\nTo type request on stdout: t\n\n");
                        printf("To register a request:              r\n\n");
                        printf("To delete a request:                        d\n\n");
                        printf("To quit the process:                q\n\n");
                        }
                }
        temp[0] = '\0';
        ind = -1;
        }
}

/*
******************************************************************************
*
*       Function to rename the remote file locally.
*
******************************************************************************
*/

regi(pack)
Package *pack;
{

        FILE                    *fptr;
        char                    buf[20], cmd[10];

        if (pack->header == ERROR) return (ERROR);

        strcpy (pack->login, LOGIN);

        /* Get the local name for a remote file. */
        while (1) {
                printf ("Local Name:  ");
                fflush (stdin);
                gets (pack->rlocal);
                strcpy (cmd, "rls");

        /* Ensure that there is no other remote file refered by same
                local name.  */

                if ((fptr = popen (cmd, "r")) != NULL) {
                        while ((fgets (buf, 80, fptr)) != NULL) {
                                buf[strlen (buf) - 1] = '\0';
                                if ((strcmp (buf , "No mounted files.")) != 0)
                                if((strcmp (buf, pack->rlocal)) == 0){
                                        printf ("File  %s  already exists.\n",
                                                pack->rlocal);
                                        strcpy (pack->rlocal, NULL);
                                        break;
                                        }
                                }
                        }

        /* If it is valid local name  */

        if ((strcmp (pack->rlocal, NULL)) != 0)
                        break;
        }

        pclose (fptr);

        pack->header = REGISTER;

        strcpy (pack->toa, SELF_NAME);
        pack->toport = SERV_UDP_PORT;
        return (SUCCESS);
}
```

```
/*
*******************************************************************************
*
*       File: RemoteOpen.c
*       Include files: mount.h, c.h, s.h, fdpak.h, Virtual.h.
*       Functions: r_open, ld_pack.
*
*       This file contains functions for opening a remote file for a session.
*
*******************************************************************************
*/

/* Header files. */

#include "mount.h"
#include "c.h"
#include "s.h"
#include "fdpak.h"
#include "Virtual.h"

F_NODE                  *ll_fd;                 /* Pointer to a active node. */
F_NODE                  *ll_first;              /* Pointer to the first node. */


/*
*******************************************************************************
*
*       Function to open a remote file in the required mode.
*       Given the name of the file from the user, get the network name, location
*       of the file and make a request to the well known port of the server.
*       If the session permission is obtained, get the file from file server.
*       If the file is not in the server get the updation from the creator
*       of the file and make a fresh session request.
*
*******************************************************************************
*/

r_open (file, request)
char    file[];
char    request[];
{
        Package                 pack;
        F_NODE                  *fd;
        int                     n, clilen, port;
        int                     sock;
        struct sockaddr         cli_addr;
        char                    cpack[PACKSIZE + 1], scpack[PACKSIZE + ACKSIZE + 1];
        char                    dumreq[3], Host[50], dummy[100];


        set_pack(&pack);
```

```
/* Bind a socket for session. */

sock = server (FILE_CLI_PORT);

/*  Get the info about the file. */

ld_pack (file, request, &pack);
if (pack.header == NOFILE) return (NULL);
unpak_pack(pack, cpack);

/* make a session request. */

client (pack.present, pack.present_port, cpack, strlen (cpack) + 1);
cpack[0] = '\0';
scpack[0] = '\0';
Host[0] = '\0';

/* Recieve a message. */

n = recvfrom(sock, scpack, PACKSIZE + ACKSIZE + 1, 0, NULL, &clilen);
send_ack (scpack, cpack, Host);
if (n < 0) erro_dump ("rp.c : Error in recieving pack.");
pak_pack(&pack, cpack);

/*  If access permission is not obtained */

if (pack.header == NOFILE) {
        sprintf (cpack, "%d %s %s %d",
                        SERV_PRESENT, pack.network, SELF_NAME,
                        FILE_CLI_PORT);

/*  Send a request for updation of location to the creator. */

        client (pack.owner, pack.owner_port, cpack, strlen (cpack) + 1);
        strcpy (cpack, "");
        scpack[0] = '\0';
        Host[0] = '\0';
        n = recvfrom(sock, scpack, PACKSIZE + ACKSIZE + 1, 0, NULL, &clilen);
        send_ack (scpack, cpack, Host);
        if (n < 0) erro_dump ("Error in opening the remote file.\n");
        sscanf (cpack, "%d", &n);
        if (n == -1){
                close (sock);
                erro_dump ("File not found.\n");
                }
        else {

        /*  Recieve the updation from the creator. */

                sscanf (cpack, "%d %s %s %d",
                        &n, dummy, pack.present, &(pack.present_port));
                pack.header = SESSION;
                unpak_pack (pack, cpack);
                port = pack.present_port;
```

```
                    /* Make a fresh request to the new location. */

                              client (pack.present, pack.present_port,
                                             cpack, strlen (cpack) + 1);
                              cpack[0] = '\0';
                              scpack[0] = '\0';
                              Host[0] = '\0';
                              n = recvfrom(sock, scpack, PACKSIZE + ACKSIZE + 1,
                                             0, NULL, &clilen);
                              send_ack (scpack, cpack, Host);
                              if (n < 0) erro_dump ("rp.c : Error in recieving pack.");
                              pak_pack(&pack, cpack);
                              if (pack.header == DENY){
                                        close (sock);
                                        erro_dump ("File access permission denied.\n");
                                        }

                    /* Update the mounting table. */

                              update_MT (file, pack.present, port);
                              }
                    close (sock);
                    }

          /* Open the file in the required mode. */

          if (pack.header == ACCESS){
                    close (sock);
                    if (strcmp (request, "r") == 0)
                              rtransact (pack);
                    else if ((strcmp (request, "w") == 0) ||
                                        (strcmp (request, "a") == 0))
                              wtransact(pack, request);
                    else if ((strcmp (request, "e")) == 0)
                              etransact (pack);
                    else erro_dump ("Ambiguous mode.\n");
                    }
          else      {
                    close (sock);
                    erro_dump ("Remote file access denied.\n");
                    }
          return (SUCCESS);
}

/*
*****************************************************************************
*
*         Function to load all the details about the request and file in a
*         Package.
*
*****************************************************************************
*/
```

```
ld_pack(file, request, pack)
char     file[];
char     request[];
Package *pack;
{

         FILE                       *ptr, *fptr;
         char                       buf[91], cmd[15];
         int                        j,i = 0;


         if((ptr = fopen(".MT", "r")) == NULL)
                  erro_dump("files.h: Error in opening .MT");

         if ((fgets (buf, 90, ptr)) == NULL){
                  pack->header = NOFILE;
                  return (0);
                  }

         sscanf(buf, "%s %n", pack->rlocal, &j);
         while (strcmp(pack->rlocal, file) != 0) {
                  fgets(buf, 90, ptr);
                  fgets(buf, 90, ptr);
                  if (fgets(buf, 90, ptr) == NULL){
                           pack->header = NOFILE;
                           return (0);
                           }
                  sscanf(buf, "%s %n", pack->rlocal, &j);
                  }
         i = i+j;
         sscanf(buf + i, "%s %n", pack->present, &j);
         i = i+j;
         sscanf (buf + i, "%d %n", &pack->present_port, &j);
         i = i+j;
         sscanf (buf + i, "%s", pack->permissions);

         fgets (buf, 90, ptr);

         sscanf (buf, "%s %d", pack->owner, &pack->owner_port);

         fgets (buf, 90, ptr);
         strcpy(pack->network, buf);
         if (pack->network[strlen (pack->network) - 1] == '\n')
                  pack->network[strlen(pack->network) -1] = '\0';

         pack->header = SESSION;
         strcpy(pack->request, request);
         strcpy (pack->toa, SELF_NAME);
         pack->toport = FILE_CLI_PORT;
         strcpy (pack->login, LOGIN);
}
```

```
/*
********************************************************************************
*
*       This program opens a remote file for read, write or editing.  It takes
*       two other  command line arguments; one for local name of a remote file.
*       Two mode of opening.  If the opening is for 'READ', the remote file is
*       opened block by block.  If the mode is for 'WRITE' a file or input from
*       the screen is written to remote file block by block.  If the opening
*       is for edit whole file is copied and the whole file is written back.
*
********************************************************************************
*/

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

        F_NODE                  *fd;
        char                    buf[100], opt [4];
        int                     i, num;

        ll_fd = NULL;

        /*  Open the remote file for required mode.  */

        if (r_open (argv[1], argv[2]) != SUCCESS) {
                strcat(argv[1], " : No such mounted file.");
                erro_dump (argv[1]);
                }

        if ((strcmp (argv[2], "w") == 0) ||
                (strcmp (argv[2], "a") == 0) ||
                (strcmp (argv[2], "e") == 0))
                erro_dump ("SUCCESS.\n");

        /*  If the mode is for 'READ' read the file block by block.  */

        fd = ll_fd;
        strcpy (opt, "!");
        while (opt[0] != 'c') {
                for (i = 0; i < 10; i++) {
                        if (fd->mesg[i][1] != '!')
                        printf ("%s", fd->mesg[i]);
                        }
                while (1) {
                        printf ("\n\n*****************************************");
                        printf ("*************************************\n\n");
                        printf ("\n\nClose choose 'c'.\n");
                        printf ("Previous Page Choose 'p'.\n");
                        printf ("Next page Choose 'n'.\n");
                        printf ("\n\n*****************************************");
                        printf ("*************************************\n\n");
                        fflush (stdin);
```

```
                    scanf("%s",opt);
                    if ((opt[0] == 'c') || (opt[0] == 'p') ||
                            (opt[0] == 'n'))
                            break;
                    printf ("Choose correct menu.\n");
                    }
            if (opt[0] == 'p')
                    get_prev();
            else if (opt[0] == 'n'){
                    get_next ();
                    }
            fd = ll_fd;
            }

    /*  Free all the memory allocated.  */

    wind_up();

}
```

```
/*
********************************************************************************
*
*        File: UserMobil.c
*        Include files: mount.h, c.h, s.h, fdpak.h, Virtual.h.
*        Functions: rum_open, ld_um_pack.
*
*        This file contains functions for a user to access remote files from
*        remote machines.
*
********************************************************************************
*/

/* Header files. */

#include "mount.h"
#include "c.h"
#include "s.h"
#include "fdpak.h"
#include "Virtual.h"


F_NODE                  *ll_fd;             /* Pointer to the active node. */
F_NODE                  *ll_first;          /* Pointer to the first node. */

/*
********************************************************************************
*
*        Function to open a remote file by a remote user.
*
********************************************************************************
*/

rum_open (pack)
Package pack;
{
        int                     sock, n, clilen, port;
        struct sockaddr         cli_addr;
        char                    cpack[PACKSIZE + 1], scpack[PACKSIZE + ACKSIZE + 1];
        char                    dumreq[3], HostName[80], Host[50], dummy[100];
        int                     Port;

        strcpy (HostName, pack.toa);
        Port = 6464;

        /* Create a socket. */

        sock = server (FILE_CLI_PORT);
        unpak_pack (pack, cpack);
        client (HostName, Port, cpack, strlen(cpack) + 1);
        cpack[0] = '\0';
        scpack[0] = '\0';
        Host[0] = '\0';
        n = recvfrom(sock, scpack, PACKSIZE + ACKSIZE + 1, 0, NULL, &clilen);
```

```
if (n < 0) erro_dump ("Error in recieving file details from client.\n");
send_ack (scpack, cpack, Host);
pak_pack(&pack, cpack);

if ((pack.header == NOFILE) || (pack.header == DENY))
        erro_dump ("Access denied.\nPassword may not be matching.\n");
printf ("SUCCESS\n");

unpak_pack(pack, cpack);

/*  Send a session request to server.  */

client (pack.present, pack.present_port, cpack, strlen (cpack) + 1);
cpack[0] = '\0';
scpack[0] = '\0';
Host[0] = '\0';
n = recvfrom(sock, scpack, PACKSIZE + ACKSIZE + 1, 0, NULL, &clilen);
if (n < 0) erro_dump ("Error in opening remote file.\n");
send_ack (scpack, cpack, Host);
pak_pack(&pack, cpack);

/*  If file is not in server find new location.  */

if (pack.header == NOFILE) {
          sprintf (cpack, "%d %s %s %d",
                         SERV_PRESENT, pack.network, SELF_NAME, FILE_CLI_PORT);
        client (pack.owner, pack.owner_port, cpack, strlen (cpack) + 1);
        strcpy (cpack, "");
        scpack[0] = '\0';
        Host[0] = '\0';
        n = recvfrom(sock, scpack, PACKSIZE + ACKSIZE + 1, 0, NULL, &clilen);
        send_ack (scpack, cpack, Host);
        if (n < 0) erro_dump ("rp.c : Error in recieving pack.");
        sscanf (cpack, "%d", &n);
        if (n == -1){
        close (sock);
        erro_dump ("File not found.\n");
        }
else {
        sscanf (cpack, "%d %s %s %d",
                &n, dummy, pack.present, &(pack.present_port));
        pack.header = SESSION;
        unpak_pack (pack, cpack);
        port = pack.present_port;
        client (pack.present, pack.present_port, cpack,strlen (cpack) + 1);
        cpack[0] = '\0';
        scpack[0] = '\0';
        n = recvfrom(sock, scpack, PACKSIZE + ACKSIZE+1, 0, NULL, &clilen);
        send_ack (scpack, cpack, Host);
        if (n < 0) erro_dump ("Error in opening remote file.\n");
        pak_pack(&pack, cpack);
```

```
                if (pack.header == DENY){
                        close (sock);
                        erro_dump ("File access permission denied.\n");
                        }
                close (sock);
                }

        /*      Open the file in required mode.  */

        if (pack.header == ACCESS) {
                close (sock);
                if (strcmp (pack.request, "r") == 0)
                        rtransact (pack);
                else if ((strcmp (pack.request, "w") == 0) ||
                        (strcmp (pack.request, "a") == 0))
                        wtransact(pack, pack.request);
                }

        else    {
                printf ("Package Header:  %d\n", pack.header);
                close (sock);
                erro_dump ("Access permission denied.\nFile may be deleted.\n");
                }
        return (SUCCESS);

}

/*
********************************************************************************
*
*       This program opens a remote file for a remote user.
*
********************************************************************************
*/

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

        F_NODE                  *fd;
        char                    buf[100], opt [4];
        int                     i, num;
        Package                 pack;


        if (argc != 3) erro_dump ("FORMAT: 'File', 'Mode'.\n");
        ll_fd = NULL;
        set_pack (&pack);
        ld_um_pack (&pack, argv[1], argv[2]);

        if((rum_open (pack)) != SUCCESS){
                erro_dump ("No such Mounted Files.");
                }
        if ((strcmp (pack.request, "w") == 0) ||
                    (strcmp (pack.request, "a") == 0))
                        erro_dump ("SUCCESS\n");

        fd = ll_fd;
        while (opt[0] != 'c') {
                for (i = 0; i < 10; i++) {
                        if ((fd->mesg[i][1] != '!') ||
                                (strcmp (fd->mesg[i], "") != 0))
                                printf ("%s", fd->mesg[i]);
                                }
                while (1) {
                        printf ("\n\n****************************************");
                        printf ("******************************\n\n");
                        printf ("\n\nClose choose 'c'.\n");
                        printf ("Previous Page Choose 'p'.\n");
                        printf ("Next page Choose 'n'.\n");
                        printf ("\n\n****************************************");
                        printf ("******************************\n\n");
                        fflush (stdin);
                        scanf("%s",opt);
                        if ((opt[0] == 'c') || (opt[0] == 'p') || (opt[0] == 'n'))
                                break;
                        printf ("Choose correct menu.\n");
                        }
                if (opt[0] == 'p')
                get_prev();
                else if (opt[0] == 'n')
                get_next ();

                fd = ll_fd;
                }

        wind_up();
}


/*
*************************************************************************************
*
*       Function to get file information and password for client's server.
*
*************************************************************************************
*/

ld_um_pack (pack, file, mode)
Package *pack;
char    file[];
char    mode[];
{

        char    dummy[9];
        strcpy (pack->rlocal, file);
        strcpy (pack->request, mode);
        printf ("\n\nPlease give the Clients Address.\n");
        scanf ("%s", pack->toa);
        printf ("\n\nPlease give the login name.\n");
        scanf ("%s", pack->login);
        pack->toport = FILE_CLI_PORT;
        pack->header = USER_MOBIL;
        printf ("\nPlease give the password: ");
        strcpy (pack->passwd, getpass(dummy));
}
```

```
/*
********************************************************************************
*
*       File: FileTransfer.c
*       Include Files: unistd.h, mount.h, s.h, c.h.
*       Functions: update_rmmt, get_file, get_mv_pack.
*
*       This file contains all the functions for transferring a shared file.
*
********************************************************************************
*/

/* Header files. */

#include <unistd.h>
#include "mount.h"
#include "s.h"
#include "c.h"


/*
********************************************************************************
*
*       This program transfers a shared file to a specified remote machine.
*       Send a message to the other machine about the intention of transferring
*       a file. Transfer list of clients of that file. Transfer the file.
*       Update the local mounting table. Send a message to the creator of the
*       file about the change of location.
*
********************************************************************************
*/

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

        char                    cmv_pack[FILESIZE + 1];
        int                     sock;
        struct sockaddr         cli_addr;
        int                     clilen;
        char                    cack[ACKSIZE + 1], scack[ACKSIZE + 1], network[100];
        char                    fpack[FILESIZE + 1], sfpack[FILESIZE + 1];
        char                    owner[50], Host[50];
        int                     owner_port;
        int                     n, port;
        long                    offset = 0;

        if (argc != 3) erro_dump ("File and Destination required.\n");

        scack[0] = '\0';
        cack[0] = '\0';
        Host[0] = '\0';
        /* Bind a socket. */

        sock = server (FILE_MOBIL_PORT);

        strcpy (cmv_pack, "");

        /* Send the list of clients. */

        get_mv_pack (argv[1], cmv_pack, owner, &owner_port, network);
        client (argv[2], SERV_UDP_PORT, cmv_pack, strlen (cmv_pack) + 1);
        n = recvfrom(sock, scack, ACKSIZE + 1, 0, NULL, &clilen);
        send_ack (scack, cack, Host);
        if (n < 0) erro_dump ("Error in transferring file.\n");

        /*      If the request is okayed, */

        if (cack[0] == '8') {                           /* If it is FILE_MOBIL. */
                sscanf (cack + 2, "%d", &port);
                strcpy (fpack, "");
                system ("clear");
                printf ("\n\nTransfering file .");
                while (offset != -1) {
                        offset = get_file (argv[1], fpack, offset);
                        client (argv[2], port, fpack, strlen (fpack) + 1);
                        printf (" .");
                        }
                printf ("\nTransferred.\n");

                /* Update the creator. */

                sprintf (cack, "%d %s %s %d", UPDATE_PRESENT,
                                network, argv[2], SERV_UDP_PORT);
                client (owner, owner_port, cack, strlen (cack) + 1);

                /* Update the mounting table. */

                update_rmmt (argv[1]);
                }

        /* If the file transfer request is refused. */

        else erro_dump ("File transfer request refused by the client.\n");

}
```

```
/*
**************************************************************************************
*
*       Function to update the mounting table once the file is transferred.
*
**************************************************************************************
*/

update_rmmt (local)
char    local[];
{

        FILE    *rptr, *wptr;
        char    line[100], pline[100], oline[100], nline[100];

        if ((rptr = fopen (".RMMT", "r")) == NULL)
                erro_dump("rmv.c : Error in opening .RMMT.\n");

        if ((wptr = fopen (".temp", "w")) == NULL)
                erro_dump ("rmv.c : Error in opening .temp.\n");

        while (fgets (line, 99, rptr) != NULL){
                fgets (oline, 99, rptr);
                fgets (nline, 99, rptr);

                sscanf (line, "%s", pline);
                if ((strcmp (pline, local)) != 0){
                        fprintf (wptr, "%s", line);
                        fprintf (wptr, "%s", oline);
                        fprintf (wptr, "%s", nline);
                        }
                }

        fclose (rptr);
        fclose (wptr);

        rename (".temp", ".RMMT");
}

/*
**************************************************************************************
*
*       Function to copy a block of a file.
*
**************************************************************************************
*/

get_file (file, fpack, n)
char    file[];
char    fpack[];
long    n;
{
        FILE                    *fp;
        int                     i = 0;
        char                    buf[100], dummy[2000];

        if((fp = fopen (file, "r")) == NULL)
                erro_dump ("Error in opening requested file.\n");

        strcpy (dummy, "");
        fseek (fp, n, 0);
        while ((fgets (buf, 99, fp) != NULL) && (i < 9)) {
                strcat (dummy, buf);
                i++;
                }
        n = ftell(fp);
        if (i < 9) n = -1;
        fclose (fp);

        sprintf (fpack, "%d %s %ld ", FILE_MOBIL, file, n);
        strcat (fpack, dummy);
        return (n);
}

/*
*******************************************************************************
*
*       Function to get all the info about the clients who has access to the
*       file being transferred.
*
*******************************************************************************
*/

get_mv_pack (local, cmv_pack, owner, port, network)
char    local[];
char    cmv_pack[];
char    owner[];
int     *port;
char    network[];
{

        FILE                    *fptr;
        int                     i, j,k;
        char                    line[200], nline[200], oline[200], dummy[50], FLAG = '0';

        if ((fptr = fopen (".RMMT", "r")) == NULL)
                erro_dump ("Error in opening mounting table.\n");

        sprintf(cmv_pack,"%d %s %d %s.rmv ",
                        FILE_MOBIL,SELF_NAME,FILE_MOBIL_PORT, local);

        while ((fgets (line, 200, fptr)) != NULL) {
                sscanf (line, "%s %n", dummy, &j);
                if((strcmp (dummy, local)) == 0) {
                    if (FLAG == '1') {
                              fgets(nline, 200, fptr);
                              fgets (nline, 200, fptr);
                              }
                    else if (FLAG == '0') {
                              fgets (oline, 200, fptr);
                              fgets (nline, 200, fptr);
                              if (nline[strlen (nline) -1] == '\n')
                                        nline[strlen (nline) - 1] = '\0';
                              strcat (cmv_pack, nline);
                              strcpy (network, nline);
                              strcat (cmv_pack, " ");

                              if (oline[strlen (oline) -1] == '\n')
                                        oline[strlen (oline) - 1] = '\0';

                              strcat (cmv_pack, oline);
                              strcat (cmv_pack, " ");
                              sscanf (oline, "%s %n", owner, &i);
                              sscanf (oline + i, "%d", &i);
                              *port = i;
                              FLAG = '1';
                              }
                    strcat (cmv_pack, line + j);
                    strcat (cmv_pack, " ");
                    }
          else {
                    fgets(line, 200, fptr);
                    fgets (line, 200, fptr);
                    }
          }

     fclose (fptr);
}
```

```
/*
******************************************************************************
*
*       File:  de.c
*       Include Files:  mount.h, s.h, c.h.
*
*       This file has code for deleting a mounting point by the client.
*
******************************************************************************
*/


/*      Header files.  */

#include "mount.h"
#include "s.h"
#include "c.h"


main(argc,argv)
int     argc;
char    *argv[];
{
        FILE                    *rptr, *wptr;
        char                    sendline[PACKSIZE + 1], line[100], nline[100];
        char                     local[30], present[50], mode[5];
        int                     j, port, FLAG = 0;

        if (argc < 2) erro_dump ("File Name Not Given!");
        if ((rptr = fopen (".MT", "r")) == NULL)
                erro_dump ("ERROR: No Such File Name.\n");
        if ((wptr = fopen ("temp", "w")) == NULL)
                erro_dump ("ERROR: Creating a temp file.\n");

        /*  Search for a mounnting table and delete it.  */

        while (fgets (line, 99, rptr) != NULL) {
                sscanf (line, "%s %n", local, &j);
                if (strcmp (local, argv[1]) == 0) {
                        sscanf (line + j, "%s %d %s", present, &port, mode);
                        fgets(line, 99, rptr);
                        fgets (nline, 99, rptr);

        /*  Inform server about the deletion.  */

                        sprintf (sendline, "%d %s %s %s",
                        DELETE, nline, SELF_NAME, LOGIN);
                        FLAG = 1;
                        client (present, port, sendline, strlen (sendline) + 1);
                        }
                else {
                        fprintf (wptr, "%s", line);
                        fgets (line, 99, rptr);
                        fprintf (wptr, "%s", line);
```

```
                    fgets (line, 99, rptr);
                    fprintf (wptr, "%s", line);
                    }
          }

     fclose (rptr);
     fclose (wptr);
     if (FLAG == 1) rename ("temp", ".MT");
     else printf ("\nERROR:  %s, No such file.\n", argv[1]);
}
```

```
/*
*********************************************************************************
*
*       File:  rde.c
*
*       This file has code for deleting mounting points when a shared file is
*       deleted.
*
*********************************************************************************
*/

#include <stdio.h>

main(argc,argv)
int     argc;
char    *argv[];
{
        FILE                    *rptr, *wptr;
        char                    line[100], nline[100], local[30], present[50], mode[5];
        int                     j, port, FLAG = 0;

        if (argc < 2) {
                printf ("File Name Not Given!");
                exit(0);
                }
        if ((rptr = fopen (".RMMT", "r")) == NULL) {
                printf   ("ERROR: No Such File Name.\n");
                exit(0);
                }
        if ((wptr = fopen ("temp", "w")) == NULL) {
                printf ("ERROR: Creating a temp file.\n");
                exit (0);
                }

        /*  Search for the required mounitng point, delete and update.  */

        while (fgets (line, 99, rptr) != NULL) {
                sscanf (line, "%s %n", local, &j);
                if (strcmp (local, argv[1]) == 0) {
                        fgets(line, 99, rptr);
                        fgets (nline, 99, rptr);
                        FLAG = 1;
                        }
                else {
                        fprintf (wptr, "%s", line);
                        fgets (line, 99, rptr);
                        fprintf (wptr, "%s", line);
                        fgets (line, 99, rptr);
                        fprintf (wptr, "%s", line);
                        }
                }

        fclose (rptr);
        fclose (wptr);
        if (FLAG == 1) rename ("temp", ".RMMT");
        else printf ("\nERROR:  %s, No such file.\n", argv[1]);
}
```

```
/*
********************************************************************************
*
*       File: rdec.c
*
*       This file contains code for deleting a mounting point for a specified
*       client, for a specified file.
*
********************************************************************************
*/

#include <stdio.h>

main(argc, argv)
int     argc;
char    *argv[];
{
        FILE    *rptr, *wptr;
        char    line[100], local[50];
        char    logcli[100], clien [100];
        int     FLAG = 0, dummy;

        if (argc != 4) {
                printf ("Four arguments needed.\n");
                exit(0);
                }

        if((rptr = fopen (".RMMT", "r")) == NULL){
                printf ("ERROR: In opening Mounting Table.\n");
                exit (0);
                }

        if ((wptr = fopen (".temp", "w")) == NULL)
                return;


        while (fgets (line, 99, rptr) != NULL) {
                sscanf (line, "%s %s %d %s",
                                local, clien, &dummy, logcli);
                if ((strcmp (argv[1], local) == 0) &&
                        (strcmp (argv[2], clien) == 0) &&
                        (strcmp (argv[3], logcli) == 0)){
                        FLAG = 1;
                        fgets (line, 99, rptr);
                        fgets (line, 99, rptr);
                        }
                else {
                        fprintf (wptr, "%s", line);
                        fgets (line, 99, rptr);
                        fprintf (wptr, "%s", line);
                        fgets (line, 99, rptr);
                        fprintf (wptr, "%s", line);
                        }
                }
        fclose (rptr);
        fclose (wptr);

        if (FLAG == 1)
              rename (".temp", ".RMMT");
        else printf ("ERROR: %s@%s doesnot have remote access to %s\n",
                    argv[2], argv[3], argv[1]);
}
```

```
/*
**********************************************************************************
*
*       File:    list.c
*
*       This file has code for listing remote files.  The object code of
*       this file is 'rls' which is used inside programs.
*
**********************************************************************************
*/

#include <stdio.h>

main()
{

        FILE                    *fp;
        char                    buf[80], file[40];


        if ((fp = fopen (".MT", "r")) == NULL) {
                printf ("No mounted files.\n");
                exit(0);
                }

        while ((fgets(buf, 80, fp)) != NULL){
                sscanf(buf, "%s", file);
                printf ("%s\n", file);
                fgets(buf, 80, fp);
                fgets(buf, 80, fp);
                }

}
```

VITA

Jayathirtha N. Mojnidar

Candidate for the Degree of

Master of Science

Thesis: FINFS: A FLEXIBLE INTERNET NETWORK FILE SYSTEM

Major Field: Computer Science

Biographical:

Personal Data: Born in Bangalore, India, April 6, 1964, the son of Narasinga Rao M.N. and Vijaya.

Education: Graduated Pre-university Certificate at Bangalore, India, in June 1982; Received Bachelor of Engineering Degree in Mechanical Engineering from University of Mysore, at Mandya, India in January 1987; completed requirements for Master of Science degree at Oklahoma State University in December, 1993.