Adding Remote File Access to Berkeley UNIX† 4.2BSD Through Remote Mount‡

Edward Hunter

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

This report describes the design and implementation of a remote file access mechanism that has been added to Berkeley UNIX 4.2BSD. The mechanism allows for access to remote files in a manner transparent to the user. This is accomplished through the use of remote mounts. A remote mount allows portions of a foreign file system to appear locally accessible to a user. This mechanism allows the user to do any operation on the file as if the file were local to the user's machine. Some of the issues concerning remote file access are presented, and areas for future work are discussed.

1. Introduction

Falling hardware costs allow computer resources to be distributed within a user community. This distribution is causing a change in the way computing resources are managed and allocated. Local area networks are slowly breaking down the hardware barriers of communication that existed between isolated machines. However, there still exist, in many cases, software barriers that prevent easy sharing of the resources controlled by these machines. Over time, various methods have been proposed to minimize or eliminate these impediment. Initially, the methods were simple, allowing only the transfer of data between machines located on the same network. These transfers were normally initiated by a user and required complete knowledge of the location of the data to be moved. This usually meant knowing the name of the device on which the file resided in addition to the name of the host. Work is now being done to provide transparent access to objects located on a different machine than the user's. Transparent access means that the user accesses all files with the same set of commands. The same commands that work for local objects will also work for

† UNIX is a trademark of AT&T Bell Laboratories

‡ This work was sponsored by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4031, monitored by the Naval Electronics Systems Command under contract No. N00039-C-0235. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

remote objects. The goal is to provide a uniform view of the file system to the user of a collection of machines in a network. The collection of file systems should appear, as much as possible, as a single file system.

However, each machine should have the ability to maintain a certain degree of autonomy. By autonomy we mean that the system administrator should be able to control access to its resources by outside users. Transparent access simplifies the user's view of his environment, and allows for modification of the underlying system structure to provide better service. In other words, the implementation of the access mechanism may be changed without affecting those using it. Additionally, this collection of machines should not necessarily behave like a single machine (as in Locus[1]). It should be possible to limit the sharing of resources through administrative access control. This could mean that certain resources may only be available to particular hosts on the network.

This report will describe the design and implementation of two methods for providing remote file access for UNIX. The rest of Section 1 will supply some background information about the requirements and objectives for remote access. Section 2 will present a design for the first method. This is an early version of the LucasFilm extended file system (EFS) for Berkeley UNIX 4.2BSD, which follows the super root methodology. Section 3 will present a design for the second method. This is a remote mount method developed for Berkeley UNIX 4.2BSD. Section 4 will describe future improvements and modifications.

1.1. Issues Surrounding Remote File Access

Remote access provides a means of reaching objects that are not part of a machine's physical domain. The physical domain of a machine is considered to include only the storage devices directly connected to the processor of the machine. A remote object is geographically located at some place other than the local machine. A remote object's location may be across the room or across the country. As with any mechanism which expands the horizons reachable by a user, there are several issues that must be dealt with correctly. This section will briefly touch on some of these issues as they apply to any remote access method. In general the issues boil down to administrative decisions about the autonomy of, and privileges available to, each user or machine in a community. In addition, different administrative domains must interact to iron out differences between various local policies. One can foresee a time when there might be a United Nations type body to settle disputes between various administrative computer domains around the country or the world.

The two major areas of administrative concern when dealing with file system access are authentication and security. Also, there are several design issues that should be addressed in addition to the administrative concerns. They include: reliability, performance, ability to incorporate a wide class of machine architectures, functioning in a range of networks environments, and the image presented by the system to an end user.

Whereas some amount of physical security may be achieved in the case of a single machine, a network is an entirely different matter, especially if it traverses a continent or the globe. There needs to be means to insure that the requests

being responded to by the file system are valid requests from real users, and not attempts to extract information by unknown parties. An authentication server should be used by every file server to verify requests. Many researchers have discussed the problems of providing security in a network environment, and their possible solutions.[2] From their work several ideas which may be applied to a remote file access mechanism can be taken. However, the issue of security affects all aspects of an operating system, and should therefore be examined in that light. Such an examination is beyond the scope of this report. It should be noted, though, that the remote file access mechanism should attempt to verify the identity of the users for whom it is providing service, but it should not attempt to be the final arbiter. In other words, the mechanism should perform some checks to insure that the request format is valid, and the request is for a file that exists on the server machine. The local file access mechanism on the server, however, should determine whether or not the operation is to be allowed.

A remote file access mechanism should provide reliable access to remote objects in the community. It can do this by use of robust protocols for carrying out its accesses. Reliable protocols and issues dealing with reliable transmission of data have been addressed numerous times before, and several protocols have been developed which provide this reliable service. [3,4] Therefore, the issue of designing reliable data delivery is not crucial to our remote access design. In fact, the mechanism should work with any reliable transport protocol provided by a system. The mechanism will depend on having a reliable transport protocol. It should be noted, however, that the remote access mechanism should be robust with respect to failures at the access protocol level. It should be prepared to handle improperly constructed requests and improper requests in a robust fashion.

A concern with any new mechanism is its performance. A new file access mechanism is no exception to this rule. For the file system to be usable by the general community, it must present as small a performance degradation as possible for that community. Those users who do not require this facility should experience no change in performance for their applications. Those users who do require this facility should see, what they consider a minimal impact on their applications. Performance aspects of the mechanism should be considered at each step of the design. Efforts should be made to maximize the performance wherever possible. Once a prototype has been constructed, it should be tested to determine where any bottlenecks might be and to eliminate them.

A remote file access mechanism ideally should be usable across any machine architecture currently in use or planned for the future. This is not a realistic goal, however. A mechanism designed to work across all architectures will probably have numerous ad hoc mechanisms in it to handle special cases for certain architectures. This can only degrade the performance for all the architectures. Because of this observation, it is best to make the mechanism as general as possible but not completely general. This should allow the mechanism to function well within most environments, and be easily modifiable in those for which it was not designed originally. Some design decisions about the Berkeley remote file system have been dictated by these considerations.. First, the system will be optimized to communicate with systems using eight bit bytes (or octets). Several architectures have different byte sizes which will not be handled by the

standard implementation, for example the DEC-20 and the BBN C70. Second, the system will be designed to communicate with other UNIX systems. This will eliminate the need for converting between different file system representations when doing accesses. It should be possible for any system which wishes to join a file sharing community to create a server to handle the translation from a standard network file access format to its own internal format. For instance, a pathname could include a file name for a VMS file system. This file name would be sent to the VMS system for translation but all operations on the file would have to be converted by the VMS server to UNIX representation before the results are returned to the client machine. This may not be easy to do for certain operations. For example, in the current UNIX file system it is possible to seek past the end of a file. The file system will automatically grow the file for the user. This operation may not have an equivalent operation in all operating systems. In that case the operation might fail for a file on another system. At present the problem is not a concern of this project. In future releases, however, it should be possible to support other file system architectures other than that of UNIX.

Besides communicating with a wide range of architectures, the mechanism should be able to handle a range of network topologies. This means that the implementation will not be specific to one type of network hardware or topology. There are several standards for communication between geographically separated hosts on a network, and we will initially be relying on the DOD specified internet protocols.[3,5] The DOD protocols are widely used in the research community of which we are a part, and therefore thoroughly tested. The access mechanism should be designed in such a way that the network transport mechanism may easily be replaced without affecting any of the higher layers that depend on it.

The last major concern for a file access mechanism is how the system will appear to the end user and to the administrators who must manage the facility. If the mechanism were obscure or hard to manage, then no one would use it. Also, if it did not fit in well with the user's view of the system, then people would be reluctant to use it. A user should be able to use the system with as little training as possible. In summary, the mechanism should have minimal impact on the end users of the system. Current applications should be able to take advantage of the new facility with little or no modification. From the administrative point of view, the mechanism should require little maintenance and fit smoothly into the day to day operations of the installation.

1.2. Previous Work in Distributed File Systems

Previous work in remote file access has been done at several different research institutions. Various designs have been tried, each providing a different level of functionality and service. For non-UNIX systems there is the V kernel,[6] and the Xerox distributed file system.[7] Notable among those for UNIX systems are the Newcastle Connection,[8] the S/F UNIX distributed file system,[9] the Locus distributed file system,[1] and the IBIS distributed file system.[10]

Each of the distributed file systems developed has been concerned with the research problem of providing access to files not directly accessible to the local machine through attached storage devices. One early effort was the Distributed

File System (DFS) developed at Xerox PARC. This file system used transactions to communicate changes in files to file servers. It did not, however, provide a directory structure for the file system. A directory structure could be constructed on the DFS as a service but was not part of the underlying design. This system demonstrates that one approach to providing a distributed file system is through a transaction based mechanism. It also shows that the concept of having servers to handle file system requests appears to give reasonable performance. The Distributed V Kernel is a more recent effort whose goal is to connect diskless workstations together in a distributed system. The V kernel uses the message passing paradigm to handle file accesses to a server machine. This demonstrates another approach to the problem of handling remote file access in a distributed system. In addition, performance figures indicate that the overhead from providing remote access may not be significant.

The Newcastle Connection provides remote access by inserting a new access mechanism layer between the user and the operating system kernel. The goal was to avoid changing either the kernel or any user programs. This system provides access transparency but not location transparency. Access transparency means that the files may be accessed using the same system calls used for local file. Location transparency means that the location of the file need not be known in order to find the file. In other words, in a location transparent file system, there is nothing in the file name to indicate its location in the network. The Newcastle Connection has a host name as part of the file name. The host name is the first element of the pathname (I.E. immediately following the first slash).

The Newcastle Connection used a non-existent global root directory to link the file systems together. The global root is non-existent because it does not exist as a data structure on any of the machines but is merely a naming convention. In essence the global root is used to connect together all of the separate name spaces of each individual UNIX system to form a larger name space encompassing them all.

The S/F UNIX system developed at Bell Laboratories uses heterogeneous operating systems to provide a distributed file system with a single global name space. There are two types of systems, the S-UNIX which provides computing service to the user, and the F-UNIX which provides file server service. The two types of systems are connected together by a virtual circuit network. The Locus distributed file system allows a collection of machines to appear as one machine, with each machine in the group sharing resources with the others. This system provides both access transparency and location transparency. The file system will automatically place the file on a storage device and replicate it if needed. The amount of replication can be specified by the user, but the location of each copy of the file is determined by the system. The last system to be mentioned here is the IBIS distributed file system. In addition to providing all the features of Locus in terms of transparency and replication, it also plans to allow file migration. File migration allows the location of a file to be changed after it has been created. The change may be done with or without user intervention. Moving the file does not change its name. These systems span the range of distributed file system methodologies investigated to date.

We felt that the previous research had shown that distributed file systems are a viable idea. The enhanced functionalities for the user are worthwhile achieving. The next step was to implement a remote file access mechanism suitable for our environment. Berkeley's environment consists of VAX 11/780 and 11/750 processors running Berkeley UNIX 4.2BSD, and connected by multiple three and ten megabit Ethernets. These machines are used both for research, development, and class work.

1.3. Remote File Access Versus Distributed Files

Two types of file access methodologies are prevalent in the area of extended file systems or file systems which expand across multiple machines. The first may be called remote file access, and the second distributed file systems. Both methodologies are generally grouped under the heading of distributed file systems. However, we would like to distinguish between the two classes because each has different constraints and goals. Both methods have similarities; for instance both allow files to be placed anywhere in the community, and allow for files to be accessed transparently. The major difference between the two methodologies is basically in their approach to naming. Distributed file systems provide a single global naming space, whereas remote file access provides a collection of several individual name spaces and a mechanism to connect them together in an arbitrary fashion.

Generally, in remote file access the placement of the files is under the control of the user or of the system administrator. A file is created in a specific location and from then on it may be found at that location unless explicitly moved. In other words, its name will indicate its location in the file system. In a distributed file systems, the system will decide where to store the file. This decision may be made with some help from the user, but, there is the additional possibility for files to automatically migrate from one storage site to another without user intervention. The first type of file access is best exemplified by the Newcastle Connection system developed at the University of Newcastle.[8]

The second methodology is best shown by the Locus file system done at UCLA.[1] The Locus distributed file system presents a single machine view to the user, with all machines having access to the files on all other machines. It allows for both access and location transparency. Also, it supports automatic replication of files.

The distributed file system case is the more difficult to implement. For one thing, if it allows replication of files, it must be more robust, in the face of network failures and other problems, to maintain file integrity. In case of a network failure, such as an internetwork partition, it is possible for separate copies of a file to be independently modified. Once the network problems are corrected, these independent copies must somehow be merged to form a single coherent instance of the file. If replication is not allowed the bookkeeping becomes simpler and this problem can be ignored at the expense of availability of the files in the system. More importantly, in many environments, a system such as Locus would present administrative problems due to its severe limitations to autonomy. Locus provides for a group of loosely coupled processors that share all

resources. All these processors form a single logical machine. To make this work, some amount of global information must be kept. For instance, there is a global mount table which is maintained by all the machines. As the number of mounts grow, so does the size of this table. As more processors are added to the collection, every machine's mount table must grow even if the machines are not all interested in the resources provided by the new system. We feel that a system should allow more control over the sharing of resources, and that a remote file access mechanism can provide the desired amount of control.

1.4. Super Root Versus Remote Mount

The remote file access methodologies examined for this project fell into two classes. The first was the 'super root' approached used in the Newcastle Connection or the LucasFilm EFS, and the second was remote mounts. In a super root scheme there is a root directory that is global to all the machines available through the remote file system, which connects the root directories of these machines. In essence, pathnames are extended to include a host identifier which specifies where the file may be found. To specify a remote file, its full pathname, including the remote machine identifier, must be given. Using certain aliasing facilities, such as symbolic links, most of the details of the naming convention can be hidden from the user. Super roots provide a convenient means of naming remote objects. They do not, however, present a view of the extended file system name space in keeping with the goal to make the remote file system appear as an extension of the local file system, since they require full pathnames to access remote objects. Additionally, since access on the remote machine starts at the root directory of that machine, the entire file tree is available for access. This may not be a desirable attribute in some environments. With this in mind, we have chosen a different semantic representation, called *remote mount*.

In UNIX it is possible to link various local file systems together to form an arbitrary tree structure using the *mount* command. The *mount* command will inform the system that a removable file system may be accessed through a certain directory. This directory is referred to as the 'mounted on directory'. If one of these directories is encountered during pathname translation, the translation continues at the root directory of the new file system.

For our file system this concept has been extended to include remote file (sub)trees. A portion of a remote file tree will be made to appear local by mounting it on a local directory. As in the local case, if a remote directory is encountered, the translation will continue at the root directory of the new file tree.

We feel that remote mount has several advantages over super roots. First, it allows arbitrary structuring of the file tree. Since the remote mount will allow an arbitrary portion of the remote file tree to appear local to a user, it is possible to control the amount of resource sharing by mounting only a restricted subtree of a remote file system. This allows files not appearing in the subtree to be invisible to users not on the same machine as the file system that includes that subtree. Second, remote mount appears more transparent than the super root. A remote mount provides some degree of location transparency. Once the mount is created,

there is no information in a file name about the physical location of the file. This is the way a local mount currently work. The number of people who must now know the location of the file is reduced to the person who has created the mount. This seems more reasonable than requiring all users of the system to know a file's location. Also, it allows the location of a file to be changed arbitrarily as long as its name does not change. By name we mean the full pathname which uniquely identifies the file. For example:

/rc/progres/edh/file.txt

is a pathname pointing to the file named "file.txt". For these reasons we have chosen a remote mount methodology for the Berkeley file system.

1.5. Approach Taken in the project

The project was divided into five tasks. The first task involved bringing up the LucasFilm Extended File System (EFS). The second task involved putting the first version of the Berkeley remote mounts into the kernel. The third task involved moving the file server process from user space to kernel space. The fourth task involved converting the underlying file system protocol from TCP to a datagram service, and the last task involved optimizing the implementation of the Berkeley remote mounts for maximum performance. This report will discuss the completion of the first two tasks.

Initially, a remote file access mechanism developed outside Berkeley was brought up on some of our research machines. This served two purposes. The first was to provide us with some experience in doing kernel development work. The second was to obtain a remote access mechanism which future designs and implementations could be tested against. This will help assess the performance impact of different remote file access methods. In addition, certain problem areas might become evident from this first implementation, and would be avoided the second time around. The LucasFilm Extended File System was felt to be close enough to the general idea of the Berkeley remote file access method to provide a useful testbed. In addition, the sources for large sections of the code were available for modification. Unfortunately, the version we had was not the one currently being run at LucasFilm, but was deemed to be suitable for our comparison purposes.

Design work for the first version of the Berkeley file system would proceed in parallel with the implementation of the modified EFS file system. This would allow modifications to be made to the design of the Berkeley file access method as problems were discovered with the EFS method. Even though the two methods required different implementations there were enough issues which were similar to allow solutions found for EFS to be applied to the RFS. Once the EFS access method was reasonably stable, work could proceed on the remote mount system.

The first two tasks have now been completed, and the Berkeley system is being tested. Once it appears to perform satisfactorily, a datagram service may be used as the transport mechanism. This would be done to hopefully improve the performance of the system. Also, the server will be converted from a user process to a kernel process. One other project within CSRG is involved with providing a datagram based remote procedure call mechanism in the kernel. This

effort should yield a connectionless method of sending file system requests from one machine to another. The last task involves performance analysis of the implementation to determine bottlenecks. An extensive series of benchmarks should be run, and the kernel should be profiled to determine the areas of possible bottlenecks in the system. These would be removed and the tests repeated. Before these benchmarks would be run, some amount of tuning between the local file system and the network would be done. Buffer sizes would be adjusted to a value which would be optimum for both. Also, changes would be made to minimize the amount of data copying done between the file system and the network. For instance, the file system and the network may swap pages of the virtual memory space through the use of reference counts on the pages to eliminate some data copying overhead. Also scatter-gather input/output could be used to reduce the amount of copying during the assembly of request packet headers.

2. LucasFilm Extended File System

2.1. System Design

As mentioned above, the first task required the bringing up of the LucasFilm Extended File System (EFS). This system of remote access was developed at LucasFilm for Version 7 UNIX. A version that had been partially converted to 4.2BSD was available at Berkeley, and this was completed and improved upon.

The LucasFilm EFS is based on the concept of a super root for the remote file systems. In a super root design, there is a special directory, or file, in the file system hierarchy, which is a gateway to the file systems of other machines. To access a file on a remote machine, a pathname that includes this special directory is given to the system. The mechanism that converts pathnames to object pointers for the file system will recognize this special directory when scanning a pathname. When the special directory is encountered in the pathname translation, the rest of the path is sent off to the remote machine for resolution. The resolution is done in a remote procedure call fashion, where the operation which caused the file system lookup and the pathname are sent to the remote site for servicing by some server process. The results of this operation are then returned to the caller. A good discussion of remote procedure calls may be found in a paper by Birrell and Nelson.[11]

In our version of EFS, the special directory is called 'efs', and is a file in the root directory of the system. The efs directory is in reality a character special device whose device number is recognized as the EFS device. A character special device is used to access and/or control real or pseudo devices in the system. All foreign file references then are given by: '/efs/<host name>/<absolute pathname>'. For example:

/efs/arpa/rc/progres/edh

With symbolic links, much of this verboseness may be hidden from the user. However, the user is still required to know what machine the files reside on to initially set up these links. EFS also has a special EFS device (called 'efsdev' in our system) for use in communicating with the kernel based part of the system. This communication is accomplished through the *ioctl* system call. Its details will

be described later.

EFS uses TCP[3] as the underlying transport mechanism. TCP provides most of the features desirable for a file access mechanism. These include reliable delivery of messages, sequencing of packets, and data consistency checking. In addition, it already existed in the system, thereby eliminating the need to develop yet another reliable transmission protocol. TCP insures the validity of the data being transferred between hosts, but requires that a connection be established between the two hosts before any data transfer takes place. This means that the state of the connection needs to be preserved at each end, thereby tying up system resources. Consequently, an upper limit is placed on the number of connections and therefore on the number of requests for service that can be handled by any single machine. In addition, our initial version of EFS allocated one TCP connection per open remote file or remote system call. This required the connection setup process every time a file was opened remotely or a system call was made. This scheme made remote system calls expensive both in resources and packet exchanges per call.

The version of EFS used at Berkeley provides a means of accessing remote files that is transparent to the user at the access level, but has no location transparency. Each user of the file must still know the location of the desired files, and specify that location exactly through a full pathname. This is one drawback with systems involving the super root concept.

2.2. Implementation details

The heart of EFS is three data structures: the host table, the patchboard, and the request packet. The host table contains a list of the hosts which allow remote access to their files, and their internet addresses. The host table provides almost the same functionality as the system's mount table. It connects the ASCII representation of a host name with its internet address.

The patchboard, as it name implies, contains the current connection information for each TCP connection. It is used by the access system to route requests for service to the appropriate host. Initially, there was one patchboard entry for each open file or system call. This patchboard contained information about the file being held open, such as its size and whether the file was a directory. In later versions of the system, there is one patchboard entry per user process per remote system. This allows one TCP connection to be multiplexed among many requests to a remote system. Each patchboard entry contains the user id of the user of this entry, a pointer to the socket to be used for sending the request, and some information about the files currently held open remotely on this connection.

A request packet asks for service from a remote server. It contains all the information needed by the server for any particular request. This information includes authentication information, the operation and the arguments of the system call. The request packet is constructed as a data structure with as many common fields as possible to maximize the reuse of space in the packet and keep the amount of data transmitted per request at a minimum.

2.3. How it works

When the operating system is initially booted, the various internal tables used by EFS must be initialized. This is done through the *ioctl* system call. *Ioctl's* allow a user process to manipulate certain aspects of the devices on the system, for instance setting terminal parameters. Several *ioctl* function codes are allocated to functions for the EFS. An initialization program uses these codes to set the patchboard and the host table. Each host is added in a separate operation. This allows hosts to be added to the host table at any time. Currently, this is a privileged operation, although this is just an administrative decision and does not affect the design. The initialization program provides a standard interface through which the system administrator may deal with EFS.

For the purposes of EFS, the system calls in the system are divided into three classes. The first class includes operations which deal with pathnames, the second class consists of operations which deal with file descriptors, and the third class contains operations which deal with moving blocks of data (i.e. read and write). Each class has a separate routine to handle the system calls in that class. The servicing of a pathname request will be described in detail below.

When a system call is made with a pathname as an argument, the first action performed is to convert the pathname to an inode pointer. This is done by a call to *namei*. The *namei* routine scans the pathname looking up each component. A component of the pathname is either a directory or a file name. In the case of a directory, the directory is opened and scanned for the next component in the pathname. If the next component is a file name (at the end of the pathname), the search is completed and a pointer to information relating to that file is returned. This information is contained in an inode, and the value returned by *namei* is called an inode pointer. To support remote files, *namei* has been modified to recognize the remote file device when looking for the next component of a pathname. If it encounters a pathname element that corresponds to that device, it stops scanning the path name and returns the remote device inode pointer.

Each of the system calls has been modified to check the return value from *namei* to determine if it matches the remote device inode. If there is no match, then *namei* has returned a pointer to a local inode. In the case where there is a match, then the pathname operation routine is called. This routine opens a connection to the appropriate host, creates a request packet to be sent to the foreign server, and sends it. If the the request can be serviced, the server will handle all the details and return any results. Some details of the connection establishment are described below.

The operation for file descriptor system calls is the same, except that the pathname scanning step is skipped. The file pointer is used by the server to index a table of open files which it maintains, and the operation is performed on the file accessed in this way. The operations for reading and writing are similar, except that the data buffers to be sent on a write are copied directly from user space into the socket, and conversely on a read.

The first working version of EFS used one TCP connection per user per open file or system call. Therefore, each path operation would generate a TCP connection, that might be short lived, for operations on a file. Because of this,

operations such as listing a directory took much time owing to the opening and closing of a TCP connection for each file in the directory. An obvious performance enhancement was allowing only one connection per user-machine pair. The next working version of EFS included this change. In this version a connection was opened whenever a user performed any operation involving a given machine. Instead of closing the connection at the end of the operation, it was kept open, and a timer was started. If the timer expired, the connection was closed. If another operation was performed to the same machine, the currently open connection was used and the timer reset. This allowed for idle connections to eventually go away, thereby freeing up system resources, while allowing active connections to remain around. In addition, it eliminated the start up time required to open the initial connection by causing it to be amortized over all later operations. For the listing of directories this savings was significant. Also, a reference counter was maintained with each open connection. This counter was incremented each time a file was opened and decremented each time a file was closed. Any connection with a non-zero reference count was not timed out. This prevented connections from closing during a file read/write operation.

2.4. The EFS Server

The EFS file server on a remote machine is not part of the kernel, and therefore executes as a user process. It performs all the operations needed for remote file access. Since the server runs as a user process, it must be started by the super user so that it may obtain all the privileges required to do file access. In particular, each child must assume the identity of the user whose request it is servicing. This allows the standard file protection mechanisms to be used on the server machine to provide the usual UNIX level of access control.

When started, the server forks off a child process to handle file system requests. A new server is spawned for each user requesting service. This allows the server to act as the agent for that user on the server system by assuming the user's user ID. As a result, user ids must be the same across machines for the protection mechanisms to work. This is normally not a safe assumption, but, until an authentication server becomes available, it is a simple and viable method. An alternative would be to pass the user name with the request, but in addition to increasing the size of the request packet, this would again require that user names be the same across all machines.

The operation of the server is simple. When a connection for service is requested and accepted, a new server process is spawned to service that request and any later requests by that user. This new server assumes the identity of the user requesting service through a 'set user identification' system call. It performs the system call as specified in the request packet with the arguments supplied. The results are then returned to the requester, and the server is ready to handle a new request. Because of the way system calls are executed by the kernel, only one request per user process may be outstanding at a time. This restriction simplifies the operation of the server since it does not have to be re-entrant. The server remains in existence until the connection is closed by the requesting site or the system crashes. This model of service is similar to the remote procedure call model developed at Xerox PARC.[11] When the connection to the server is closed,

the server exits, and the dead process is reaped by the parent. At this point the parent can collect statistics about the operation of the child; these statistics are returned by the operating system and stored in a file for later examination.

The only optimization employed by the server is in the reading of directories. When a directory is first read, the entire directory is read into the memory of the server to minimize the number file system accesses. Since most directories are read sequentially, during file listing for instance, this appears to be a reasonable optimization. Also, as mentioned above, only one server exists per user per system, thereby minimizing the process creation time for a server. This is one deviation from the model as proposed by Xerox PARC. In the Xerox model there were several idle servers waiting for requests. New ones were created as the supply of ready servers became depleted. In the EFS model, a server is only created when service is required by a client. Therefore, the first request may take somewhat longer, but no subsequent request must pay this penalty.

3. Remote File System

3.1. System Design

When designing the access method for the Berkeley Remote File System (RFS), the first major design decision to be made was about the level of abstraction in the file system at which the changes for remote file access should be inserted. There were four possible choices: the system call level, the file descriptor level, the inode level, and the device driver level. Each level had its advantages and disadvantages, which will be discussed below. Starting at the top level and working down, each will be examined in turn.

The first, possibly obvious choice, is to insert the remote access mechanism at the system call level. In this scenario, the code executed for each system call is modified to recognize a reference to a remote file. One simple way to do this is to have *namei* recognize remote files and return a special inode pointer which indicates that the file is not on the local machine. The name of the machine containing the file is included in the pathname for the file. When a remote file operation is detected, the arguments to the system call along with the pathname are sent to the remote machine for processing. The results from this processing are returned to the caller. This method is straight forward to implement. Each operation can be converted separately, and the semantics are generally easy to preserve since the remote system call will get all of the arguments that the local system call would have gotten. A disadvantage of this approach, however, is that all the information needed to complete a system call is not readily available at the system call level. For instance, the pathname is used by *namei* and discarded. For a remote system call to work, the pathname buffer used by *namei* must not be discarded. Each system call must release it after the remote call has been completed. Also, certain internal operations circumvent the system call interface and work at a lower level in the system. An example of this is the generation of *core* files for aborted execution. These files are created without using the *open* system call. A similar operation is performed to read directories during file name translation. Each of these cases would have to be handled in a special fashion.

Another possibility is the file descriptor level. The system maintains a system wide descriptor file table and uses it to decide which routines to call to do an input/output operation on a file object. The system file descriptor table already has the ability to handle different types of files, and a remote file simply is another type of file. At this level, all operations on open files can be intercepted by marking remote files in the file descriptor table. Whenever one of these remote files is encountered, it is handled by some remote file handling routine. The advantage of this method is the centralization of the remote mechanism at the file descriptor table. The number of operations that would have to be implemented at this level is small: *open, close, read, write, ioctl and stat.* The disadvantage is that not all the file operations need to open a file to perform operations on it. An example is the *change directory (chdir)* command, which changes the current working directory for a user. Commands such as *chdir* would have to be handled in a non-standard fashion. The cleanness of the abstraction quickly disappears at that point. Secondly, since the virtual memory loader does not go through the file descriptor table to execute programs, additional changes would have to be made to support remote file execution. A modification of this level handles some of the problems well but others not at all.

An inode is the handle by which all system code refers to file system objects. Whenever a pathname is resolved, the result is an inode. There already exist several types of inodes, and this scheme would involve adding an inode of type 'remote' to this list. Each system call would treat this inode in the same way as it treats any other inode. The difference would come at a lower level, when the modified inodes are to be written to disk or an inode is used to access a file. At this point the routine involved would recognize the remote inode and would send them across the network to the remote site for storage. In the case of reading or writing a file the requests to read or write the file would also be sent across the network. The advantages of this method come because the inode is the center of all file system operations in the system. All system routines use an inode to access files. Therefore, it appears the proper place for inserting a remote access mechanism. The disadvantages become evident when the use of an inode is examined. First, to perform operations on an inode, it must be locked to prevent it from being changed by other processes in the system. For remote accesses, the inode would be locked at both the client and server. This would be required so that the client may use the inode like any other inode. At the server end the inode would have have to be locked to prevent any other client from changing it while it was being used. If the client crashed while holding locked inodes this would prevent the server from using them again. If one of the inodes was a directory inode, this would effectively prevent anyone from accessing any files in that directory since *namei* must lock a directory before it can read it. The locking is necessary to prevent the directory from changing if the process reading it must release the processor, while waiting for a disk operations to complete. Additionally, because of the central nature of the inode some of its fields are modified as a side effect of certain operations. Reading a block from a file, for example, causes the access time of the file to be updated in the inode as a side effect.

Because of the nature of the operations on an inode, more packet exchanges than are strictly necessary would have to be done to complete an operation. The sequence would be: request an inode, receive the inode and modify it, return the modified inode, and an acknowledgement of the completion of the operation, for a total of four packet exchanges. The operation could be simplified by sending a request to do the change without ever bringing the inode to the client machine, that does not need the inode information for anything. In this case, the cost of the operation would be two packet exchanges (one request and one acknowledgement).

The lowest level of abstraction is the device driver level. This is similar in concept to the SUN† Microsystem network disk. Device drivers handle the physical records of the device they manage. A device driver would have to be added to handle the remote file system blocks. When a driver detects a request for a remote file system block, it sends this request off to the remote machine. The advantage of this method is the simplicity of the operations. However, since the device driver is only one step above the physical device, it need only understand how to manage file system blocks. This would in theory make it simple.

There are several disadvantages, however. Since the device drivers handle physical blocks, there is the problem of mapping blocks with identical physical block addresses across systems. The physical device address are only unique within a single system. Also, this modification is on the wrong side of the buffer cache. There is the possibility that the system will cache disk blocks for some amount of time before sending them to the device driver. This leads to a multiple cache inconsistency problem between the caches of two systems sharing blocks of the same file. Such problems occur if a client believes that a disk block has been written across the network to disk when in reality it is still in the local disk block cache. If this block also exists in the disk block cache of the server or another client, there is a cache inconsistency. The client will always see the updated block in its local cache but anyone else accessing the file will see the old block which the server still holds. This problem could be solved by changing the way the disk block cache is accessed. Currently, blocks are found by device and block number. This would have to be changed to allow for remote disk blocks. A disk block would now have to be addressed by some type of logical block number which was unique across all systems.

Lastly, the remote device driver may not be able to take advantage of file system optimizations on the server machine since it would not have any knowledge of the disk block queue on that machine. Therefore, it might not be able to order its requests to take advantage of the position of the disk heads. The server machine would still be able to manage its queues intelligently but the timing of the requests being received from the clients may not arrive be optimal for the local queue management routines of the server.

After examining each of these options we chose a combination of the inode level and the system call level. Certain modifications were done at the system call level, while other operations were done at the inode level. The determination of

---

† SUN is a trademark of SUN Microsystems

the location of a file, for instance, is done by examining the inode. The scheme chosen provides the desired functionality without making extensive changes to the system. In addition, the scheme makes certain operations, such as *change directory*, easier to implement. It was decided to insert the mechanism at the inode level by allowing for a new type of inode which we called remote. A remote inode could be treated exactly like a local inode, and would contain all of the information a normal inode would contain except for the physical disk block addresses of the file. These addresses would be useless to the user of the remote file. Instead, the space normally reserved for disk block addresses was used to contain information pertinent to the remote instance of the file. At the system call level a remote inode would be recognized and an appropriate routine called to handle the remote request. A remote inode can be identified in one of two ways. The first is a flag bit marking the inode as remote. The second is a device number which is greater than the maximum device number on the client system. If the device number does not exist then the file is assumed to be remote. By using a combination of both levels it was possible to avoid the locking problem described above and still install the mechanism at a level which was common to all areas of the operating system.

The next decision was the choice of a transport mechanism for the file system requests. The standard transport protocols available in Berkeley UNIX 4.2BSD are the Internet Protocol (IP), the User Datagram Protocol (UDP), and the Transmission Control Protocol (TCP). The initial version of RFS is built using TCP as a transport mechanism between hosts. Figure 1 shows the layering of the various portions of the file access method. On initial inspection, it would seem that TCP would not be a suitable protocol on which to build a file system due to its connections and stream nature. However, closer examination will show that this is not the case. Some people already believe that TCP is an acceptable protocol to use on a local area networks.[12] Looking at some of the functions needed for a remote file system will show that TCP already provides those functions. First, there needs to be some assurance that the data sent between hosts has arrived in the order it was sent. This is necessary because file system operations in UNIX are not idempotent, and UNIX will allow file operations with data buffers bigger than a single network packet. This would cause fragmentation of the data as it is sent through the network. UDP or IP will not guarantee ordered delivery in this case. TCP provides this through an acknowledgement mechanism. Next, there needs to be some assurance that the data sent has arrived intact. TCP provides this through checksumming of the data packets at both ends of the connection. Lastly, there needs to be some assurance that the data being received has really been sent by the party that it claims to be from, and TCP also provides this.

A drawback of TCP over some other protocols is the requirement that a connection be established before any data transfer can take place. However, one connection's cost can be amortized over a long period if it is used for multiple requests. The byte stream nature of the TCP protocol is not really a desirable one for file system operations, but not a major impediment. For a file system, the concern is whether the whole request got to the server. Therefore, a packet protocol of some sort would be more desirable. TCP allows for the partial

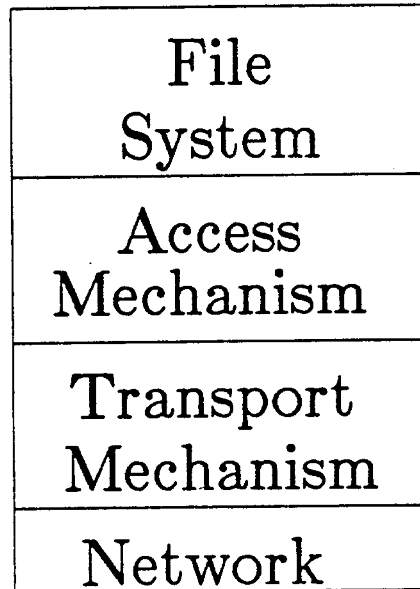

Figure 1: Protocol layering for remote file access

delivery of a packet. In other words, TCP allows some bytes to be delivered to the user before all of the bytes sent have been received. This requires essentially another protocol layer above TCP to mark packet boundaries, if an application expects them.

Performance measurements have been made of the Berkeley 4.2BSD TCP, which show that it performs reasonably well when used in its optimum configuration.[13] In addition, measurements have been made of the local file systems.[14] These show that the file system is faster than the network when running at maximum speed on an unloaded system. However, this will rarely be the case in a production environment, except when large amounts of data are being moved. These measurements point o the network as the possible bottleneck in any remote access mechanism. Therefore, an effort should be may to use the network as efficiently as possible.

3.2. Implementation details

Internally, the heart of the access method is the mount table. The standard system mount table has been modified to show that a mount is either local or remote. This is done through the addition of a *type of mount* field. Local mounts are handled in the same way they always were. The table entry for the remote mount contains a pointer to an RFS protocol control block (rfspcb). This control block contains the information necessary to maintain the mount, which includes the address of the host to which the mount is connected, a sequence number to be used with all requests involving this mount, the name of a socket to be used for communication, and a capability that may be used to control access through the mount. Additionally, information that may be used by the server to locate the mount point is also stored: the inode pointer, the device, and the inode number. The mount table also contains a pointer to an inode which is the local representation of the root of the remote file tree to which this mount corresponds.

This inode is used by the unmount command and to flag remote accesses.

To support remote inodes, the 'in-core' inode structure had to be modified slightly. These modifications involved inserting a pointer to the rfspcb for remote inodes and a pointer to the mount table for all inodes. This second pointer was added for two reasons. The first was to help the operation of the remote unmount command. The second was general system efficiency. The system currently searches the mount table linearly whenever a mount is encountered. The new pointer will allow an inode to point directly to the mount table entry for the root of the mount point. The addition of remote mounts will increase the size of the mount table, since more mounts per system will be possible. Therefore, a linear search will become more costly. This additional pointer allows this search to be avoided. Also, a remote flag bit was added to the flag bits for an inode. Lastly, space was allocated in the area normally used for disk block pointers to hold information pertinent to the remote inode. This information is described in detail below. Figure 2 shows the relationship between the remote inode and the mount table.

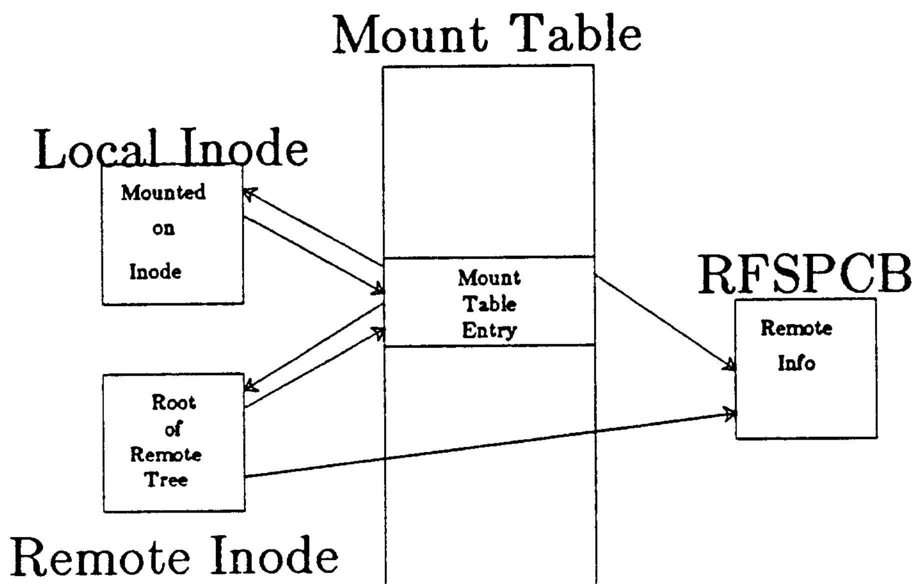

Figure 2: Mount Table for Remote Mounts

For communication between two installations sharing file systems, some unique way of identifying inodes and mount points was required. The information needed to do this is contained in two structures. The first structure contains information about the mount, while the second contains information about the inode. They are called rfsdata1 and rfsdata2, respectively. Figures 3 and 4 show the contents of both structures.

The first structure contains information returned by the server in response to the initial request for the mount. This is the information that the server will use to identify all future requests as coming from this mount. The structure contains a unique sequence number which is used to match the mount request to the current incarnation of the server. This is similar to the sequence number in the

Xerox remote procedure call mechanism. The structure also contains some information that the server may use to find the root inode of the mount point on its local file system. This includes the server's inode pointer and the server's device and inode number for the root. The device, inode number pair is used to find the inode on disk if it has been flushed from the server's memory. Lastly, there is a capability, whose exact use has not been specified at this time, although it is envisioned that it will be distributed by the server to control access on a machine by machine basis.

The second structure contains information returned with a remote inode by the server. This information is used by the server to identify the inode locally whenever a request is made to access it. This is the handle assigned to the inode by the server and has no meaning to the client. This information includes the local server device and inode number, so that the inode may be retrieved if it has been flushed from memory.

```
struct rfsdata1 {          /* mount info */
    short rfs_cook_len;     /* length of info */
    dev_t rfs_dev;          /* rfs device */
    long rfs_inonum;        /* inode number */
    long rfs_inode;         /* inode pointer */
    long rfs_seqno;         /* sequence number to use */
    short rfs_c_size;       /* capability size */
    long rfs_cap[2];        /* capability */
};
```

Figure 3: Mount table RFS information

```
struct rfsdata2 {          /* inode info */
    daddr_t rfs_idb;        /* device # for char/block devices */
    short rfs_cook2_len;    /* length of info */
    dev_t rfs_dev;          /* rfs device */
    long rfs_inonum;        /* inode number */
    long rfs_inode;         /* inode pointer */
};
```

Figure 4: *namei* RFS information

3.3. How it works

To set up a remote mount, the *remote mount* system call is used. This call takes as arguments a remote host address, a remote mount point, and a local mount point. First, the local mount point is checked for existence. Next, a connection to the remote site is opened, and a request for a mount is sent. Initially, only the super user will be allowed to request remote mounts, however this is only an administrative decision. If the mount point exists on the remote system, then the remote server may grant the mount request. If the request is granted, then a packet containing an rfsdata1 structure is returned. The information from this structure is copied into an rfspcb, and a pointer to the rfspcb is placed in the mount table. The mount table entry is marked *remote* so that it may be recognized later. Following the response message is a remote inode

structure, which contains the information in the inode on disk at the server end. This information is copied into a locally allocated inode. The local inode is allocated, held in core, and marked *remote*. This local inode represents the root of the remote file (sub)tree on the client. At this point, the mount operation is complete, and files on the remote system may be accessed. The local inode is used only as a place holder in the client system for determining that the mount is remote. The information it contains is not considered to reliably represent the root information at the server.

To unmount a remote file (sub)tree, a *remote unmount* system call has been added. This call finds via the *namei* routine the remote inode locked in core. It then frees that inode. Next, it tries to communicate to the remote system that the mount has been removed. Whether or not this succeeds, the mount table entry is cleared and the connection closed. No retries are attempted in case the server has crashed. By clearing the mount table entries for the mount, it will not be possible for the client to use it again. If the server has crashed, it will not have any knowledge of the mount when it is rebooted.

3.4. Operations on Remote Inodes

Since all file operations are handled at the inode level, there is no need to distinguish between pathname functions and file descriptor operations. Consequently, the two routines which perform these functions under EFS were merged into one. Also, since only one connection per mount is maintained, no searching must be done to find the proper connection to use.

When a *namei* request is made, it is processed normally. The difference comes when remote mounts are encountered by *iget*. *iget* is the routine which actually reads the next inode from the disk or the inode cache if it has been used recently. *iget* also performs the lookup in the mount table to resolve pathnames which traverse mounts. For a remote mount, the inode returned by *iget* will be marked as remote. *namei* will detect this, and call a routine to resolve the remote name. This routine will form the remote *namei* request, and send it followed by the rest of the path. If the remote translation succeeds, the result will be a remote inode. If the translation cannot be completed, the result will be the remainder of the pathname, which has not been scanned, for processing on the client system. If an error occurs during translation, an error code will be returned. In the case where another remote mount is encounter while the server is translating the pathname enough information will be returned to the client for it to communicate with the server possessing the new mount. Eventually, the client will be communicating directly with the server responsible for the file it wishes to access. That server may decide whether to allow the access or not based on the capability it receives. This allows a server to decide whether or not to allow the capabilities it distributes to be copied. It could, for instance, encrypt the capability with the address of the host it distributes it to as the key. This would prevent other hosts from using it. This method of resolving multiple remote mounts allows each server to determine what access rights it wishes to allow for the file systems it controls.

All input/output operations for an inode are handled by a single routine, *rwip*. This routine examines the inode structure to determine what type of inode is being operated on, and calls the appropriate routine to handle it. A remote inode is merely another type of inode for the system to deal with. When a remote inode is detected, the routine for reading and writing remote inodes is called. This routine determines what type of operation is to be performed, and constructs an appropriate request packet. For a read operation, the response to the request will be an acknowledgement packet followed by data. The acknowledgement packet indicates the actual amount of data being returned for the read operation. For a write packet, the data accompanies the request and the acknowledgement packet indicates that the write operation has been completed. The data for either operation is copied directly between user space and the TCP socket to minimize the amount of data copying at the client end.

One difficulty encountered with remote inodes is the execution of programs located on a remote machine. The programs need to be copied to the client machine before execution. Unfortunately, the virtual memory system is designed to perform the mapping from memory pages to disk blocks before reading the program into memory. It then uses this mapping to read the pages directly into memory using the raw disk interface. Consequently, it bypasses the inode when reading pages in from the disk. This will not work across a network since the client does not know the location of the disk blocks on the remote system. There are two possible solutions to this problem. The first involves rewriting the virtual memory system to refer to logical disk blocks that may exist anywhere. The second is arranging for the virtual memory system to read the entire execution image into local memory when the program starts, and later swap out whatever is not needed. The second alternative was the one chosen for this project. This approach was deemed satisfactory for our current needs, and should only cause noticeable performance problems for large programs. Large programs will take a longer time to start because of the delay while waiting for all the pages of the program to be read. Normally, a program can start as soon as the first page has been read into memory. This will not be the case with remote execution. Eventually, the virtual memory system will be redone to allow for remote file blocks, and it should then be possible to page in files over the network.

For the remote mounts to appear as transparent as possible, all operations should be supported. This includes changing directories to a remote directory and can be done by sending, to the server, a current directory inode reference with the *namei* requests when the search does not start at the root of the mount point.

3.5. The RFS Server

The RFS server process is similar in some respects to the EFS server process. Both processes run in user space. This is mainly done for debugging purposes, as eventually the server will be made a kernel process similar to the page daemon. Like the EFS server, the RFS server forks a copy of itself to handle each mount request. There is one child per mount on the server system. Once the child has accepted the mount, it then waits for requests involving that mount.

There are three types of requests that the child can handle. The first is *mount*, the second is *namei*, and the third are all other file system requests. When a mount request is made, the child checks the local pathname for the mount to see if it exists. If it does and access is allowed, then the mount will succeed. The child changes to that directory so that it may make all accesses relative to that point. It then returns a remote inode to the sender along with the information needed to identify the mount.

When a *namei* request is made, it contains the information returned by the server to identify the mount, as well as the authentication information for this user. This is followed by a pathname to be looked up on the server system. Also, if the current directory is on the server machine, this information is included in the request. The server uses the request information, including the location of the mount point, to search for the file locally. If the file is found, a response packet is constructed and sent back to the client. The locally allocated pathname of the inode is kept by the server in case of future requests involving that inode. Note, however, that this is only necessary for the user-mode server. Currently, the only time this information is released is on the closing of an open file or through an LRU aging of the inodes maintained by the server. An alternative to this approach would be to release the inode through a remote *iput* call. However, it was decided to minimize network traffic in exchange for speed and memory space. Memory at the server end is effectively free of charge for a user process, through the use of virtual memory. Therefore, the server keeps full pathnames to be used in locating the inodes on the server machine whenever they are needed. This scheme will not be used for a kernel based server. For that type of server the device and inode number may be used to fetch the inode from the disk, thereby eliminating the need to convert pathnames to disk addresses on each access. The device number and inode number will identify the file as well as a pathname.

For file system requests, the server examines the request packet to determine whether the information it contains corresponds to this version of the server. This is done by comparing the sequence number contained in the packet with the sequence number of this incarnation of the server. If they match, the server looks up the inode supplied with the request and performs it. If the request generates any results, these are returned to the requesting machine. In any case, a response packet is returned to show that the operation either completed successfully or failed. In the case of failure, the local error code is returned so that it may be passed to the user. The sequence number mentioned above is provided as a check against system crashes. The sequence number is changed each time the server is restarted. It therefore provides a check that the client is still communicating with the same version of the server for which the mount was created.

For handling the reading and writing of files, the server will maintain an LRU cache of recently opened files. The reason for this is the limitation of twenty open file descriptors per process in UNIX. Because of this limitation, the server must be prepared to cache several open files, and re-open files that have been closed because they were not used recently. If a request to read or write an inode is received and there are no free file descriptors, the file at the bottom of the cache will be closed, and its file descriptor recycled. Scenarios can be constructed where this scheme will be inefficient, but in general it should be sufficient for the

user mode server. However, it will not be used in a production environment. For a kernel based server, which is the one planned for the final version, the number of open file descriptors will not be so severly limited.

3.6. The Hard Problems

There are several problems that the above method of providing remote file access does not adequately address. This section will briefly outline some of those problems and the solutions adopted or proposed but still under consideration. It should be noted that some of these problems are a result of the semantics of the UNIX file system. Therefore, they may not have suitable solutions under the given semantics. This may mean that the UNIX semantics is not suitable for a distributed system, but such a discussion is beyond the scope of this report.

When doing name translation in *namei*, symbolic links must be followed. Since a symbolic link is just a character string, it may begin with a slash. This shows that the link points to the root of the file tree. The question with a remote file is: which root should be used? That of the client or that of the server? We have chosen to have the slash indicate the root of the client machine. Therefore, if a symbolic link beginning with a slash is encountered, the translation will stop and be completed on the client machine. We felt that allowing symbolic links starting with slash to be processed on the server machine presented a breach in security for remote mounts. If the translation were allowed to continue on the server machine it would be possible to change the current directory through the symbolic link and have access to portions of the remote file tree which would normally be inaccessible.

UNIX provides a means of locking files for cooperating processes. These locks are on a file by file basis. The information about these locks is kept in the 'in-core' inode structure. In the local case this is not a problem. When the system crashes the lock information is lost and all the locks removed. The remote case must now be able to deal with crashes involving either the client or the server. Since the server will be managing the files, it will be maintaining the lock information. If the client, unbeknownst to the server, should crash after creating a lock, there would be no way to remove the lock.

One possible solution would be to have the system time out locks on files. This would require a dirty bit for the inode to indicate whether the file has been touched since the last time the timeout interval was checked. The next question would be how long should a lock be held for. Clearly, no matter what duration is chosen, an example can be constructed which will show this scheme to behave incorrectly.

In the case where the server crashes and comes back up again, the client must be prepared to reestablish the locks when communication is restarted. In summary, file locking is a problem which has not been completely solved at this point.

The single-site file system attempts to use the disk in an intelligent fashion by reading disk blocks ahead when an access is done in anticipation of future sequential accesses. This algorithm must be reexamined for remote file access, so that the file system and the network may be used together to provide maximum

throughput to the system. The question here is whether read ahead should be supported across the network connection or only done at the server end. If read ahead across the network is not allowed, then the performance penalty of retrieving individual blocks across the network must be paid. If read ahead is allowed and the disk block cache is used, there a possibility for inconsistency it the files.

The UNIX kernel maintains a buffer cache of recently read disk blocks. If copies of the same disk blocks for a file are allowed to exist on more than one system, a cache inconsistency problem arises. A block may be modified on the server without the knowledge of any of the clients. Or worse, a file being shared by two clients may be modified by one of them with neither the server nor the other client being aware of the fact. Unfortunately, the buffer cache is designed to know local disk blocks, and therefore is not easily modified to take into account blocks that may not be part of the local machine. To solve these problems in the near future, remote disk blocks will not be cached at any of the clients, and read ahead will not be allowed. This will avoid the cache inconsistency problem in exchange for some loss of performance in reading ahead. However, the system will maintain the semantics of files. It might be desirable to allow only the client to cache disk blocks locally. Only when the blocks were to be flushed would they be sent back to the server. In this case, the server would need a way to retrieve outstanding blocks if some other process also tried to open the file for writing. It is evident that there are several schemes to chose from. Some of them interact with other areas of the system, and therefore need to be examined in that light before a decision is made.

Since the file system is being allowed to extend across multiple machines, network routing now becomes an issue for file access. Network topologies can be constructed which allow multiple access paths from one machine to another, using either multi-homed hosts or dual ported disks. It would be desirable for the client/server pair to be able to take advantage of this information to make routing decisions if there is a network failure. This requires some knowledge of the topology of the file storage devices with respect to the network. This is not a straight forward routing decision similar to those to be made in a gateway, since the path to a file system may be through a host with which no mount is currently established. For example, with a dual ported disk each interface may be on a different machine. If one of the two machines crashes, it will still be possible to access the disk through the second host. This information however, cannot normally be determined through gateway routing messages alone. Ideally, the client should be aware that there are two paths to the remote file and establish a new mount through the second machine without user intervention. An examination of the various routing failures modes needs to be made to determine what cases need to be handled and in what way.

4. Future research

The file access method as it exists can be used to do remote access to different objects. There are performance enhancements still to be made. The first calls for moving the server into kernel space. The next involves tuning the mechanism. Another requires converting from TCP to some connectionless

protocol. The last involves moving to a standard kernel based RPC mechanism to handle remote system calls and data transfers. Each of these points will be covered in turn below.

Currently, the server process runs as a user process. This is desirable during debugging, since it is easier to debug user processes, but for a production environment a kernel based server would be more appropriate. A kernel based server has a number of advantages, the main one being the ability to directly call *namei*. Currently, the server has to duplicate the functionality of *namei* in user space. Each component of the pathname must be examined, using the *stat* call, to check for symbolic links. Each *stat* call results in one *namei* call at the server. This basically multiplies the overhead on pathname lookup by a factor proportional to the number of components in the pathname. Given the amount of system time already spent in *namei*, doubling it can only have a disastrous effect on performance. Secondly, a kernel based server will have an easier time handling certain file operations, since it may take advantage of the strategy built into the current file system with regard to allocating blocks on the disk.

Next, the access mechanism should be optimized to work in conjunction with the file system of the machine on which the server is running. The network software is optimized to handle data in a manner which is suitable for local area networks. The file system on the other hand is optimized to handle data in a manner which is suitable for the storage devices used on each machine. Some middle ground should be found where the file system is not constantly waiting for the network or vice versa. Perhaps a read ahead scheme or some more intelligent scheme of data management for data destined to go to another machine should be devised. Currently, the file system makes no distinction between data to be used locally and data to be sent to another host. Perhaps such knowledge should be available to the file system to assist in making file accesses. Other optimizations might include allowing flexible IPC socket sizes, so that the size of a socket used by the file system might be increased to be the same size as the basic read or write unit of the file system. Also, the file system, the network protocols and the virtual memory system may be changed to interact better with each other allowing the copying of page-aligned data by the passing of page pointers. Currently, the data must be copied from file system buffers to network buffers before being sent. Memory to memory copying is always expensive.

Another modification which will be made to the RFS is the replacement of the underlying TCP protocol with some connectionless protocol. TCP was initially chosen as a transport mechanism because it provided all the features needed for remote file access in terms of reliability. However, as it is a connection based protocol, it requires that the state of the connection be preserved at both ends of the connection. System resources are required for each connection thereby putting an upper limit on the number of connections which may be in use at one time. Generally, the amount of resources being used is not a problem for the requesting site, unless multiple servers are being accessed at once. The server, however, may be servicing requests from multiple hosts simultaneously.

In a connectionless protocol much, if not all, of this state information could be eliminated, thereby allowing more requests to be serviced concurrently. Some

minimal amount of state information (mostly datagrams waiting for retransmission) must be maintained, but this would be required no matter how much other information is to be kept for reliability.

In addition, some other means of communication besides a byte stream protocol might be tried; for instance, a sequenced packet protocol similar to that developed by Xerox.[4] Since the data being sent between each host is basically atomic in nature (i.e. request-response), such a protocol might be desirable.

Another possible change would be the addition of a kernel based remote procedure call mechanism. This would allow for a standardized interface for any new system calls which might be added to the system, and provide hooks for user calls to be executed at remote sites. In addition, a standard RPC mechanism would allow different file systems and operating systems to be used, as long as they used the same protocol. Work at Xerox PARC has been done on the use of a remote procedure call paradigm for distributed computation. That work shows that the paradigm is a useful one. There is a current project at Berkeley to provide such a service for Berkeley UNIX 4.2BSD.

5. Conclusion

This report has presented some issues considered when adding remote file access to Berkeley UNIX 4.2BSD. It has discussed some of the issues involved in deciding what type of mechanism to provide, and where in the operating system to insert this mechanism. Having examined these issues, we have proposed a scheme using remote mounts as the naming structure for file access. The design details of two different methods for file naming and access were described. These were the LucasFilm Extended File System (EFS), and the Berkeley Remote File System (RFS). The LucasFilm EFS uses a super root naming mechanism while the Berkeley RFS uses remote mounts. Finally, areas for future development were outlined. These included changing from a user process file server to a kernel based file server, and optimizing the system to handle remote file accesses. These areas can be addressed once the current system is more stable.

At the time of the writing of this report, the first system (EFS) was functional but not thoroughly tested. Since there is no current intention of placing it into production in our environment, no effort was expended testing the system once the basic functionality was obtained. The second system (RFS) was working but not completely tested. A number of system calls functioned correctly for remote files. It was also possible to change the current directory into a remote directory, and to execute programs located on a remote machine. The entire system is now being exercised to discover any strange failure modes and to stress sections of it. As mentioned above, the file server still executes in user space. Because of this, it was decided that no performance measurements should be taken to compare the first method of access (EFS) with the second (RFS). Any measurements obtained now would not correctly reflect the performance of the final system, and therefore no effort was made to benchmark the RFS system. Even though the EFS server executes in user space also, we felt it better to compare the two systems as they were intended to go into production to determine the difference between a kernel based server and a user process server.

Work is currently proceeding to eliminate the last bugs with in certain system call modules, further optimize the protocol, and move the server from a user process to a kernel process. When these things will have been accomplished, the system will be tuned for maximum performance. At this point, we have shown that remote file access can be added to the Berkeley UNIX 4.2BSD kernel. In fact, access to remote files may be provided through different methods of file naming, two of which have been examined here. Remote object access appears to be a useful feature for an operating system, especially given the current interest in distributed applications. Adding it to a system is not difficult, but there are several design decisions to be made which will affect the performance of the system and the ease of its use. This report has tried to examine some of those decisions and present different alternatives.

6. Acknowledgements

This manuscript was read, at various stages, by a number of people who gave me valuable comments. In this regard I would like to thank Professor Domenico Ferrari and Professor Luis Felipe Cabrera for proofreading, comments and suggestions. In addition, I would like to thank Mike Karels for information about the internals of the Berkeley UNIX 4.2BSD network software and for overseeing the design and implementation of the file system. Also, thanks to Jim Bloom for working on the software and providing valuable input on the design decisions.

References

1. B. Walker, "The LOCUS Distributed Operating System," *Proceedings of the 9th SOSP*, November 1983.
2. R. Needham and M. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Communications of the ACM*, December 1978.
3. J. Postel, "Transmission Control Protocol," RFC 793 USC Information Sciences Institute, September 1981.
4. Xerox Corp., "Level two: Sequenced packet protocol," *Xerox XNS Protocol Handbook*, May 1981.
5. J. Postel, "Internet Protocol," RFC 791 USC Information Sciences Institute, September 1981.
6. D. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the 9th SOSP*, November 1983.
7. H. Sturgis, J. Mitchell and J. Israel, "Issues in the Design and Use of a Distributed File System," *ACM Operating Systems Review*, July 1980.
8. D.R. Brownbridge, L.F. Marshall and B. Randell, "The Newcastle Connection or Unixes of the world unite!," *Software Practice and Experience*, 1982.
9. G. Luderer, H. Che, H. Haggerty, P. Kirslis and W. Marshall, "A Distributed Unix System Based on a Virtual Circuit Switch," ACM, December 1981.

10. W. Tichy and Z. Ruan, "Towards a Distributed File System," *IEEE Journal on Selected Areas of Communications*, 1984.

11. A. Birrell and B. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, October 1983.

12. M. Padlipsky, "TCP-ON-A-LAN," RFC 872, September 1982.

13. E. Hunter, "A Performance Study of the Ethernet Under 4.2BSD Unix," *Proceedings of the 15th CMG*, December 1984.

14. M. McKusick S. Leffler, W. Joy, R. Fabry, "A Fast File System for Unix," Berkeley Technical Report CSD 83/145, July 1983.

Table of Contents

Introduction 1
Remote File Access Issues 2
Previous Work in Distributed File Systems 4
Remote File Access Versus Distributed Files 6
Super Root Versus Remote Mount 7
Approach Taken in the project 8
LucasFilm Extended File System 9
System Design 9
Implementation details 10
How it works 11
The EFS Server 12
Remote File System 13
System Design 13
Implementation details 17
How it works 19
Operations on Remote Inodes 20
The RFS Server 21
The Hard Problems 23
Future research 24
Conclusion 26