An Implementation of an Extended File System for UNIX

Clement T. Cole
Perry B. Flinn
Alan B. Atlas

MASSCOMP†
Software Engineering

ABSTRACT

This paper describes the development of an extended file system for MASSCOMP's Real Time Unix (RTU) operating system. It discusses a mechanism by which users can perform file operations upon data physically residing in backing store on a remote computer. All operations are transparent to processes running on the local host. Migration of processes to the remote computer was not considered as a goal and no attempt was made at solving that problem. This system is client/server based and operates between two or more MASSCOMP computers on the same Ethernet. These machines run an enhanced version of the RTU kernel with an enhanced version of the network software. A new reliable datagram protocol (RDP) that supports multiple connections through a single endpoint has been developed to simplify the kernel's interface to the communication mechanism and to improve the throughput of remote file operations. All remote file operations are based on transactions. A global protection domain is used to simplify the concept of file ownership; that is, a single set of *user-ids* and *group-ids* is used on all machines.

1. Introduction

Given a network of computers running Real-Time UNIX (RTU), the MASSCOMP variant of the UNIXRit74a Time Sharing System, how can users of each machine on the network easily share databases? These databases might be anything from the system sources for a large programming project, to the telephone directory for an entire company. The key points are that the database is to be shared by many different programs and users throughout the network, and potentially could be *updated* by many different programs at different times. The programs used to update the shared database should be no different than those used to update a non-shared database. The goal of the MASSCOMP EFS project was to produce a new version of the operating system that would allow network-wide file operations without requiring modifications to existing user programs. Thus a user could easily and transparently share data between multiple machines.

---

† MASSCOMP and RTU are Trademarks of Massachusetts Computer Corporation.
UNIX is a Trademark of AT&T Bell Laboratories.

1.1 What is EFS?

EFS is the *Extended File System* facility for RTU that allows users to access data residing on remote backing store. The remote backing store is connected to a normal MASSCOMP machine operating with the EFS environment. Except for a small network time delay, any valid access to the data retained on the remote backing store is *indistinguishable* from an access to data retained within the backing store on the local host. This is referred to as *network transparency*. Note, however, that a new set of error codes was introduced to cope with the new types of errors that arise with the EFS environment.

1.2 Constraints

The UNIX system call interface standard proposed and accepted by the /usr/groupBuc84a and the draft standard of the IEEE P1003 POSE working groupIEEd)a have left a seemingly indelible mark on UNIX systems manufactured by different vendors. The standard interface dictates calling conventions and semantics for all major system calls, including file I/O operations. Therefore any attempt to produce an *extended file system* (EFS) for UNIX must lie within the boundaries set by this interface. Furthermore, the authors consider it non-optimal to force users to recompile or relink a working program when the operating system for the same computer hardware is changed from being a simple version of UNIX, to one that supports an EFS. The MASSCOMP EFS works within the constraints described above. All file operations (*open*(2), *close*(2), *read*(2), *write*(2), *ioctl*(2), etc.) continue to work as they did previously. Thus any program that works under a pre-EFS version of RTU continues to function correctly with an EFS version of RTU.

1.3 Why build an EFS?

In a timeshared system where all users perform their work on the same machine, data can easily be shared among users because it is all stored locally. Indeed, sharing is the norm. Most large systems go to great expense to allow users to share everything from files to in-core program images. In many cases, users may not even know they are working with objects shared by other users.

In the case of smaller workstations, however, data is stored on each of many machines. Since sharing is difficult, its occurrences become rare. In fact, data becomes replicated much more often than it is shared. Disks are copied either physically or via a network, but new versions of data from those disks are rarely sent to all sites that contain copies of it.

Consider the difficultly of keeping several workstations running the same version of system software, or worse yet, keeping many copies of some database up to date. An EFS allows workstations connected by a local area network to share data as though it were stored locally as it is in a large timeshared enironment.

For a more concrete example, consider an application such as the design of VLSI circuitry for a central processing unit. Graphical editors, such as KIC-2Bil83a or HAWKKel84a are used by IC designers to layout the circuit geometrically. The circuits are then simulated with tools such as SPICECoh76a, Qua83a and are finally converted to an IC mask. Along the development path, different engineers work on different pieces of the problem with different tools. Each engineer works on his or her piece of the circuit, yet each may periodically need access to other parts, or to the entire circuit as a whole. In order to help the entire design team work together, the master image of the circuit and the "cell libraries" are usually stored on one central machine. Each designer need only concern himself with the files associated with his portion of the project.

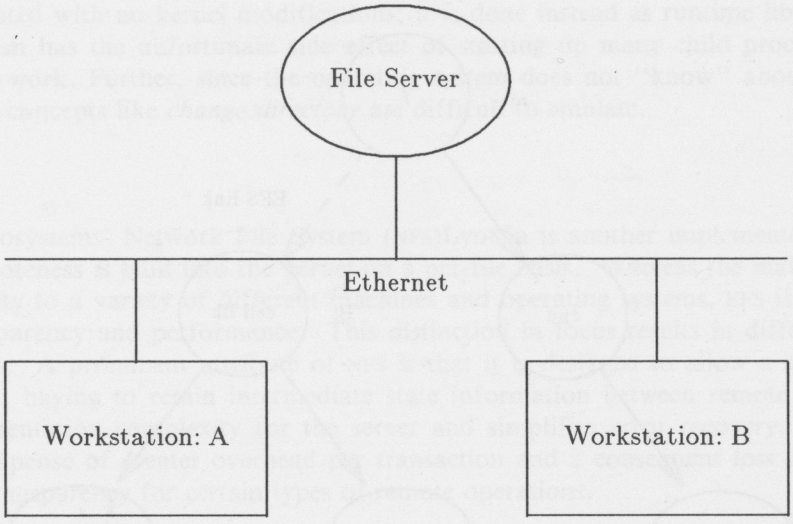

Figure: 1. Two design stations and a shared file server.

In figure 1, we see two different design stations sharing a common file server.[1] By letting each workstation remotely graft some part of the server's file system as part of its own directory hierarchy, any process running on a workstation, (such as the graphics editor) can access data on the remote file server *as though the data were stored locally.* The designer can then share common pieces, such as the cell libraries, without having to copy and store them locally. When the designer needs to use a cell, the editor many simply "read it in" from its master location on the file server.

Figure 2 shows two file systems "virtually linked" together so that each process on work station **A** views the remote directory hierarchy as part of its own. Cell libraries are contained on the remote file system, and the local system has a local copy of some part of the total circuit being designed.

1. Note, that you do not have to specify a "file server" as such. Any machine on the network that is participating in EFS can act as either a client or a server.

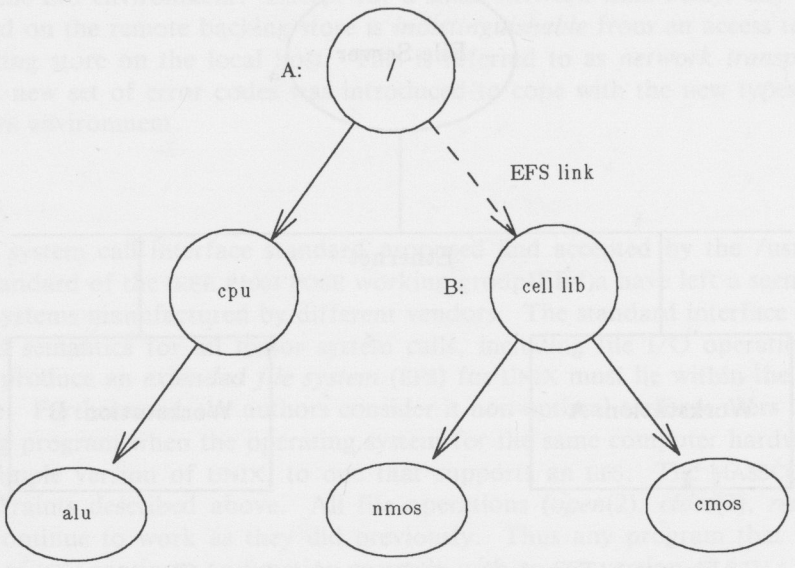

Figure: 2. Machine A is sharing part of Machine B's File System.

2. Previous Work

2.1 Version 8 File System

The goals of the MASSCOMP EFS project are similar to goals of the Version 8 File System described in the *Proceedings of the Summer 1984 USENIX conference*Wei84a and developed at Bell Laboratories. The Version 8 File System is proprietary to Bell Laboratories, and thus could not be used as a starting point for our efforts. The Version 8 File System, like EFS, is built around the *"inode"* concept.

2.2 Remote Virtual Disk

EFS differs from other network file system implementations in that the kernel has been modified to route the standard UNIX system calls to another host when necessary. It differs from the *remote virtual disk* (RVD) approach, as developed by Mike Greenwald and Larry AllenGre83a for the MIT Laboratory for Computer Science's multiple VAX/750 UNIX systems running as CPU servers. The MIT approach is similar to the LucasFilmDuf82a method. The major advantage of RVD is that under it, as under EFS, no user code must be modified. Unlike EFS however, RVD does not allow a given object-file or file system to be shared simultaneously by several machines. Rather it allows a single physical disk to be divided up into smaller chunks and parceled out to remote machines.

2.3 The Newcastle Connection

The work done by the University of Newcastle-upon-Tyne in EnglandBro82a was another approach that we chose not to follow. The Newcastle system is simple and has the advantage of being implemented with no kernel modifications; it is done instead as runtime library calls. The Newcastle system has the unfortunate side effect of starting up many child processes to do the network server work. Further, since the operating system does not "know" about the "remoteness" of a file, concepts like *change directory* are difficult to emulate.

2.4 NFS

Sun Microsystems' Network File System (NFS)Lyo85a is another implementation where the concept of remoteness is built into the kernel on a per-file basis. Whereas the main thrust behind NFS is portability to a variety of different machines and operating systems, EFS is aimed more at complete transparency and performance. This distinction in focus results in differences between the two designs. A prominent attribute of NFS is that it is designed to allow a server system to operate without having to retain intermediate state information between remote requests. This reduces implementation complexity for the server and simplifies error recovery. However, this comes at the expense of greater overhead per transaction and a consequent loss in performance, and a loss of transparency for certain types of remote operations.

3. The Design Space

The "design space" for our development effort imposed the following constraints.

1). All kernel code must be written to operate in a multiprocessor environment.

2.) EFS development must not impede any other operating system development.

3.) The design must operate using a "Client - Server" model.

4.) All operations revolve around *inodes*.

5.) Calls must be provided to enable and disable remote access.

6.) No single client process should be able block an EFS server from serving other client processes.

The next paragraphs describe these constraints and their impact on the design.

3.1 Multiprocessor Safety

Unlike other UNIX variants, notably AT&T's System V and the University of California's 4.2 BSD, RTU runs on both dual-processor and full multi-processor (MP) computer architectures. This constraint means that any new code added to the RTU kernel must not hinder the multiprocessor nature of the system.

3.2 Impact on other Development

At the time of the EFS development, three large kernel projects were underway, each of which entailed rewriting major sections of the MASSCOMP RTU kernel. The group working on EFS has become adept at folding changes into other versions of the kernel from other development groups. Overall, we have been reasonably successful, but that story is the subject for a different forum.

3.3 Clients and Servers

The operating model chosen is based on client-server style interactions. When a user process executes a system call locally, the file referenced by the call may be remote. In such cases, the local machine executes a "client" version of the operation that sends a message to the remote "server" machine. The server decodes the request, fulfills it if possible, and then returns the result to the client. The client process is blocked until the server completes the transaction.

3.4 *Inodes* and *Rinodes*

Inside the UNIX kernel, the focal point for all file operations is the *inode*.[2] This is a per-file structure that contains all of the information that UNIX keeps about the file, including its size, type, owner, protection, access and modification times, and location on the disk. While a file is being operated on, UNIX keeps a copy of its *inode* in-core. In general, the address of this in-core *inode* is the key used by UNIX system calls to gain access to the file.

In the EFS environment, an in-core *inode* address is insufficient to identify a remote file, since the regular *inode* can only describe files stored on the local system. We thus introduced the concept of a *remote inode*, or *rinode*, to describe remote files. An *rinode* is a special variant of an *inode*. Rather than containing the information that would be found in a normal *inode*, it contains a *machine-id*, which uniquely identifies a single host on the network, and the address of an in-core *inode* within the memory of that host. Where a remote file is involved, the *rinode* takes the place of the *inode* for system call operations.

3.5 Start Up and Take Down

To establish the mapping of a server's directory to the clients, a new system call was introduced: *rmount*(2). This call is analogous to the standard UNIX call, *mount*(2). Similarly, a new call was introduced to take down the connection namely: *rumount*(2). Again, there is an analogy with a standard UNIX call, *umount*(2).

3.6 Non-Blocking Server

Later in this paper, we describe the problems that can occur when a server process blocks. Simply, any client process must not cause the server to block in such a way that other client processes can not make requests to the server, or processes local to the server can not run.

4. The Implementation

The client machine can be thought of as the local host on which a user program executes. The server is the machine that is accessed by the local host to perform certain file I/O operations. All remote file operations are transactions from the client to the server. When the server receives an I/O request it takes all the information in the transaction and then operates upon that transaction. In certains cases, such as *write*(2), not all of the information is available to the server when the I/O request is sent. In those cases, the server sends a transaction back to the host requesting the needed information. All transactions return sucess or failure, and the user program is notified accordingly.

---

2. For a more complete discussion of the UNIX I/O system, seeRit78a

4.1 The *inode* and *namei*

In any operating system, there exists a method of mapping from a user's concept of the file specification to a pointer into the actual file system tables. Under the UNIX operating system, the user visible file specification is called a *pathname*, and the pointer into the file system is an *inode*. In the UNIX kernel, the routine called *namei* is used to convert from a *pathname* to an *inode*. It returns either a pointer to an *inode* or an error code. System calls that require a *pathname* as an argument such as *open*(2), *stat*(2), and *link*(2), internally call *namei* to perform a translation from the *pathname* to an *inode*.

To obtain a *transparent* file system, EFS operates on a new type of object, an *rinode*, which is a pointer into the file system on a remote machine. Thus *namei* must be able to return both *inodes* and *rinodes*. We have modified *namei* so that it walks a *pathname*, it checks each *inode* that it encounters for remoteness. This is indicated by a new bit in the *i_flag* field: IREMOTE.

To obtain this type of functionality, the path walk portion of *namei* includes new code, similar to:

```
if (inodePtr->i_flag & IREMOTE) {
    rinode = rnamei(ptrToRestOfPath);
    localInode = storeInInode(rinode);
    return(localInode);
} else {
    ... original namei() code ...
}
```

This new routine *rnamei* must look into the *rinode* and find the network handle for the remote system and put together all of the information for the remote procedure call.

In the pre-EFS system, when a user process wants to send data, it performs a *write*(2) system call which is mapped into the Ethernet driver. In the EFS based kernel, the *rnamei* calls the driver directly on a known[3] socket and performs a "remote procedure call" to the server process acting as the client's agent on the remote host.

Please notice that the read/write pointer is maintained by the client system, and not by the server, which implies that the open file pointer must be passed to the server each time an I/O operation is performed. Further, this implementation is an *inode* implementation not a *file* implementation, and no modifications to the normal UNIX open file table were required.

4.2 The Server Implementation

For the file system type system calls discussed in the client portions (*read*(2), *write*(2), *stat*(2), *link*(2), and etc.), the server must call its local system routine to obtain the needed information for the client. As an example, when the client calls its local *namei* routine and discovers it must call *rnamei*, the server picks up the *rnamei* request and calls its own *namei* routine to obtain a *remote specific inode*. This information is then sent back to the client so that client's *rnamei* routine can return the needed information to the client process that started this procedure in the first place.

For instance, when an *open*(2) takes place, the client does a *namei*, and then sends back to the server a message saying in effect: "open this file." The server has an incremented reference count for the *inode*. When the file is closed, the count is decremented.[4] Once referenced by the client, the file can be accessed by its *rinode* directly.

---

3. When the EFS is first started up the local host determines how to communicate with the remote. After the system is started, the communication information is stored locally so it can be used later for *namei/read/write* requests.

4. Incrementing the reference count does lead to error recovery problems which will be discussed later.

4.3 The Process Pool

The server contains a pool of available kernel processes that can act as agents for any EFS client request. This set of processes is known as the "process pool." This pool is a global resource and all EFS connections use the process pool to have their work performed. Furthermore, any *agent* process from the pool can act in the behalf of any client.

At EFS start up time, a master process on the server creates a socket at a well-known port[5] allowing client requests to be recieved. EFS This process is called the *lifeguard*. In general, the *lifeguard* is dormant. Its job is to dispatch incoming requests to idle agent processes, and manage error recovery.

Once a socket has been set up the system is ready to receive and handle client requests. When one is received, the *lifeguard* assigns the request to an idle *agent* process or puts it on queue until an agent is available to service the request. When a request comes in off the network:

1.) The lifeguard finds the first available agent;

2.) awakens the agent and passes the request to it.

3.) The agent copies information from the request to its *u*_area;

4.) invokes the "server" version of the requested system call;

5.) puts itself back into the available process pool as an inactive process;

6.) awaits a new request.

Notice that any process in the process pool can act as an agent for any request from any client.

In some cases the client will request more data than can be transmitted at once by the local area network. For this reason, a *more to come* flag was implemented in the returned packet. This notifies the client that more data should be expected. This feature is implemented at the lowest level of the request/reply packet handling routines.

4.3.1 Start Up and Take Down

Starting up EFS activity and taking it down is not quite as simple as with the standard UNIX *mount*(2) system call. As previously mentioned, a new call was introduced, *rmount*(2). This call is used to map a directory on the client machine into a directory on the server machine. Note that this is different from the *mount*(2) call which takes a "device name" on the local host and maps that into a directory on the local host.

When *rmount*(2) is called, the client contacts the master server process and requests that agents be made available for it. If all of the protection checks are passed, the master server sets up an active connection and allows transfers to continue. An agent process is given the request and return to the client a *network handle* with which all further communication takes place. From that point on, all EFS requests use that same *network handle* along with any file-specific data when requesting data of the server. Note that this handle must be stored with each *rinode* because it is possible for a local machine to be a client of more than one server, each on a different machine. That is to say, *the local machine may remotely mount many different directory hierarchies, and each remote file system request is performed by an agent processes running on its behalf, picked from the process pool on each server at each request.*

---

5. For details of communications see the sections on it. The concepts that are discussed are explained in detail in other literature.

It follows that the inverse operation must also be performed. A new call, *rumount*(2), was introduced to break out of a connection. As in *umount*(2), checks must be made to test if there are any active *inodes* from that client into that file tree. Since this call is not made often, a simple search is used to detect any server *inodes* that meet this criterion. If everything is clean, then the *rumount*(2) succeeds, otherwise an error is returned. As discussed later, there is a case where a *rumount*(2) is forced by error conditions. In that case, any file found active is made inactive.

5. Communications Support

Throughout the design of EFS, several assumptions were made about the attributes and capabilities of the underlying communications protocol. MASSCOMP supports an ethernet local-area-network using the DoD Internet protocol familyPos82a The programming interface to these protocols is the 4.2 BSD socket mechanismLef82a, Lef82b A primary assumption was that all communications between machines had to be carried out within the context of this mechanism.

Interactions between client and server systems are transaction-based. Most transactions consist of a single message requesting information or the performance of an operation, followed by a single reply supplying the requested information or the completion status of the operation. Others, most notably *read*(2) and *write*(2), may involve the transfer of large amounts of data in a single transaction. Thus, the communications protocol must provide a mechanism for grouping related messages. Further, since some operations may take longer than others to complete, either because of scheduling anomalies on the server or because the agent process must wait for some event, several transactions may be pending concurrently between a given pair of hosts.

Because transactions are so dynamic, the overhead of creating and destroying them must be low. To appreciate the importance of this requirement, consider that simply opening a remote file involves three separate transactions between client and server. This dynamism effectively rules out the possibility of using a protocol such as TCPPos81a which would require a separate socket and connection for each transaction.

In the process of carrying out remote operations, there are various intermediate states on both the client and the server in which network failures can have serious ill effects. Assume for example that an often used directory such as */tmp* becomes locked while a remote client is deleting a file from it. If in the midst of the operation the client system crashes, the server could slowly cease to function as processes queue up waiting to create or delete temporary files. It is clearly of vital importance that the communications mechanism provide reliable transfer of messages, and timely indication of communication failures.

To address these needs, a new *Reliable Datagram Protocol* (RDP) was designed.

5.1 RDP Functional Description

RDP provides a datagram-based communication service that guarantees in-order, reliable delivery of messages. That is, a successful return from a send operation implies that the message has been delivered to the peer RDP module on the destination host. An optional side effect of a send operation is the creation of a "connection" between the source and destination processes through which further related messages may be sent or received. The connection may be "duplex", meaning that both source and destination processes may transmit on the connection, or "simplex", meaning that only one or the other may transmit. Such connections are independent of any others that may also be in progress within the context of the same socket, and in fact, an arbitrary number may exist concurrently. A single process may manage several connections, or each of several processes sharing a common socket may manage a single connection.

When doing a receive operation, a process indicates whether it wants a message related to an existing connection, or one that is either not associated with a connection or is the first in a new one. Upon return from the receive, an indication is given about whether further messages may be received on the same connection, and whether or not reply messages may be sent.

Once a connection is established, each direction of transfer is under control of the sender. That is, the sender provides an *end-of-data* indication with the last message it sends, closing down one side of the connection. It may continue to receive from the other side, however, until the remote sender transmits its final message.

RDP provides a timeout mechanism to automatically abort a connection when the peer system crashes or some other communication failure arises. Idle connections are maintained by periodic "keep-alive" messages that are invisible to the communicating processes. Reliable delivery is ensured by positive acknowledgement of each message as it is received, and by periodic retransmission of unacknowledged messages.

5.2 RDP Implementation

RDP is implemented on top of the Internet Protocol (IP)Pos81b This choice was made primarily because it simplified the implementation by allowing us to take advantage of the considerable existing support framework for handling "Internet" addresses.

Access to RDP is provided through the 4.2 BSD socket mechanism using the "Internet" addressing domain and the SOCK_RDM socket type. RDP sockets are given local address bindings in the same manner as a UDP socket. The binding consists of a 16 bit port number and a 32 bit IP address.

Once the socket has been created and bound, a new address structure is used for send and receive operations. It is an extension of the standard "Internet" address structure:

```
struct sockaddr_rdp {
        short           sin_family;
        u_short         sin_port;
        struct in_addr sin_addr;
        u_long          sin_txid;
        u_long          sin_flags;
};
```

The first three fields correspond exactly to their *sockaddr_in* counterparts. The two additional fields, which overlay previously unused parts of a *sockaddr_in*, carry the information needed to manage RDP connections. The *sin_txid* field holds a 32 bit connection identifier that is unique among all other connections related to the same socket on the local machine. The *sin_flags* field holds two OR-able flags, SF_MORE and SF_REPLY, whose functions are described below.

5.2.1 Sending to an RDP Socket

When a send operation is performed on an RDP socket, the *sin_family*, *sin_port* and *sin_addr* fields of the destination address must be set up as they would be for a UDP socket. If the *sin_txid* field is non-zero, it must identify an existing writable connection.

A new connection is created when *sin_txid* is zero; upon return from the send, the zero is replaced by the new connection id. The bits in *sin_flags* determine whether a new connection will be simplex or duplex: SF_MORE alone creates a simplex connection from sender to receiver (i.e., in the direction of the original message); SF_REPLY alone creates a simplex connection in the opposite direction; SF_MORE and SF_REPLY together create a duplex connection. If neither bit is set, no connection is created, *sin_txid* remains zero, and the message is sent as a "standalone" datagram.

After a connection is created, the SF_MORE bit indicates that further messages are to be sent. Once a message is sent with this bit cleared, further attempts to send on the connection are rejected with an EPIPE error code (and possibly a SIGPIPE signal).

5.2.2 Receiving from an RDP Socket

In contrast to other protocols, the address structure passed to a receive operation on an RDP socket must be initialized before hand. In particular, the contents of the *sin_txid* field determine what effect the receive will have. If the field is non-zero, it must contain the identifier for an existing readable connection. Furthermore, the *sin_addr* and the *sin_port* fields must specify the remote address binding of the peer socket at the other end of the connection. In this case, the receive is satisfied only by a message sent on the specified connection; other pending messages directed to the same socket but on different connections remain queued.

If *sin_txid* contains zero, then only a message which is either the first in a new connection or is "standalone" (i.e., not associated with any connection) may be received. When the first message in a new connection is received, the returned *sin_txid* field contains the unique identifier for the connection. When a standalone message is received, the *sin_txid* field remains zero.

When any message is received on a connection (including the first message), *sin_flags* indicates connection status as follows:

SF_MORE indicates that further messages may be received on the connection. Once a message is received with this bit cleared, further attempts to receive on the connection return with a zero byte count.

SF_REPLY indicates whether messages may be sent on the connection. For a duplex connection, it is typically set unless the receiving process has previously sent a message on the connection without SF_MORE.

5.3 Development using UDP

Because the design and implementation of RDP proceeded in parallel with EFS implementation, much of the initial testing and debugging of EFS was performed using the User Datagram Protocol (UDP)Pos80a as a communication base. This was made possible by a conscious effort to isolate as much as possible the details of the network interface from the new EFS kernel code. This proved to be a successful plan. Very early on in the project, basic remote operations were being tested with UDP. In fact, even as the more complex operations were completed, UDP continued to serve as an adequate testing vehicle. Where UDP failed to satisfy EFS's communication needs was when the time came to test multiple concurrent operations. The difficulty here is that when a single socket is used for all EFS-related network traffic, UDP provides no mechanism for associating a reply message with its corresponding request. Thus, when two client processes on the same machine are waiting for replies from their EFS agent processes, nothing guarantees that the replies will be received by the right client.

6. Protection Issues

One of the trickier issues that must be discussed is protection across the EFS. In UNIX, a *user-name* (*clemc* for example) is mapped to a machine specific number, called a *user id* or *uid*. It is this number, not the name, that UNIX uses to determine a user's access rights. On one particular machine, XORN for example, the name *clemc* may be mapped to *uid* 92, while on another machine, say VAMPIRE, the *uid* may 25 instead.[6] By convention, user programs read the password file to obtain the *uid* from a *user-name* or vice-versa. Some network applications, such as the remote copy program, *rcp*(1), pass the *user-name* to the remote side, which in turn converts it to the corresponding local *uid*.

6. We have completely ignored the issue of users who have a different *user-name* on each machine. For instance, John Smith uses *johns* as his *user name* on the machine named *EARTH*, and uses *smith* as his *user name* on the machine named *MOON*.

In the case of EFS, when the two machines are able to share a file system, it would seem that they need to *share* a actual file *in some way*. Either they must share the password file, or the EFS software must have the server process convert each request from the *uid* from one client to a *uid* of the server.

Therefore, we wrote a program, *idupdate*(8), that works like *fsck*(8), and walks the file system in a standalone mode, and changes the *uid* from one to another. This serves as a conversion tool for customers who have the problem of different machines that do not agree with each other on the *uid*'s for certain users, (i.e. *clemc* not being 92 on all of the machines, for instance.)

After all of the machines in the network are converted to use the same password file, the normal UNIX protection mechanism may be used. This, however, is not enough. Without a change, this implementation would allow *Root* privileges on one machine to be maintained on the server. In most environments, there exist some files on the certain machines that should not be made accessible to everyone - including users from a remote machine with *root* privileges. For this reason, we added *selective mount points*. At *rmount*(8) time, the system administrator of the server has a file of mount points that a remote system is allowed to mount upon. In our VLSI example, that might be a path such as: /usr/vlsi/library. Any attempt by the client to mount elsewhere would fail.

Furthermore, because of the back pointer to the client used in pathname walking, (see the dot-dot problem described later) a program would not be able to subvert this mount point by performing a *change directory* to the remote and then using dot-dot style pathnames to get at data it should not have.

7. Error Recovery

With two hosts and two network interfaces involved, the number of failure modes is much greater than the usual single machine case. In addition, side effects of certain EFS transactions can cause problems ranging from the invisible and harmless to complete paralysis of the server. Error recovery is possible and has been included as part of the EFS.

7.1 Failure Modes

All EFS transactions are between two machines at a time, and error recovery can be analyzed for the two machine case even when fifty or more machines are all running EFS on the same network. In all cases, errors are treated as specific to the client-server pair of the current transaction.

The possible failure modes are:

1.) Client failure

2.) Client net failure

3.) Server failure

4.) Server net failure.

5.) Any combination thereof.

Client or server failure means an operating system failure such as a crash or power down. Net failure is the failure of the network interface or communications link without the host's failing also. This may result from intermittent cable faults or where the network is implemented in a coprocessor with a address space seperate from the host, a failure in that processor.

7.2 Side Effects

The client-server model used for EFS is not stateless. Once an *rmount*(2) has been done, information is required on both server and client to maintain the EFS relationship. Since EFS intervenes at the *inode* level in the kernel, some bookkeeping must be done to maintain reference counts sensibly. A *rinode* in-core on a client requires the associated *inode* on the server to be in-core there. This is accomplished by maintaining *inode* reference counts on the server for each client transaction on that *inode*. The effect is that if a client is exclusively using an *inode* on a server, the reference count on the server will mirror that on the client at all times. It is tricky but possible to achieve this, unless a failure occurs. Depending on which failure mode has occurred, a server may find itself with *inode*s in-core that are not being referenced by any local process but which have non-zero counts or a client may find itself with *rinode*s in-core which cannot be used since the server cannot be accessed.

A much more serious problem is the locked/unlocked problem. Inodes, being a shared resource, have a locking and unlocking mechanism that is used to prevent simultaneous access by two processes. Any process trying to access a locked *inode* blocks until it is unlocked. Certain EFS transactions leave *inode*s locked on the server, waiting for the next transaction from the client to complete the system call and free the *inode*. If a client failure occurs between these transactions, the server has locked *inode*s in-core which have been locked by a now-dead client.

7.3 Error Detection

There are two methods of error detection. Any client or server which receives an error return from a send or receive operation will activate a failure test immediately. If a failure of the local network interface is detected or if the operation times out (signifying the death of the other machine or its net interface), appropriate recovery procedures are instituted (see below). If not, the immediate system call or transaction still fails and returns an error code. It is simply too difficult (or impossible) to retry transactions at this point because of uncertainty as to exactly what happened before the failure.

The method of error detection just described is sufficient for the client since there are no side effects there that can cause damage in the absence of EFS activity. The case on the server is much different. Consider what happens when ten clients start ten transactions which leave ten *inode*s locked on the server, then each promptly fails or the network fails. The server must detect, in the absence of EFS traffic, that something is wrong and that certain *inode*s are locked on behalf of clients which could be unreachable right now. In order to detect this situation, the time of the last transaction from a given client is kept by the EFS *lifeguard* process. Periodically, a function is called which checks these "last transaction times" and sends a packet to any clients which haven't been heard from for a certain length of time. If no answer is received, the server recovers from that client (see below) and continues to check the other clients with which it has established a remote mount relationship until it has verified that all clients are alive.

7.4 Client Recovery

The simpler of the two cases by far, recovery on the client merely consists of identifying the offending server and unmounting all directories associated with that server. If the failure of the client's net interface is detected, then all *rmounts* are taken down. *Rinode*s which are being accessed by other processes on the client will each cause some net failure when accessed and will be cleared at that time.

7.5 Server Recovery

The ability of the server to recover is based on keeping information about *inode*s which are in-core on behalf of clients and which of those are locked. When a server process detects an error in trying to reach a particular client, it simply looks up all of the *inode*s which are in-core on behalf of that client and clears them out. If it is known *a priori* which *inode*s are locked, it is possible to circumvent the blocking problem mentioned above. Also, any *rmounts* which the dead client has are also removed from the server's *rmount* table.

Finally, each time a client tries to *rmount* to a particular server for (what the client considers) the first time (i.e., there are no *rmounts* on the client to that particular server) this belief is communicated to the server. If the server finds that it has *rmounts* from that client, it clears them out (and any *inode*s in-core from that client) before completing the current *rmount* request. This scheme corrects for the case where a client has failed and been rebooted before the server notices it.

8. Some Interesting Problems

In the course of development of EFS a number of problems were encountered that are generic to a file system that maps across multiple machines.

1.) The *core* file problem.

2.) The problem of server processes blocking on a client request.

3.) The problem of *pathnames* that cross "mount points" in a directory hierarchy.

4.) The *stat*(2) problem.

5.) The *ustat*(2) problem.

The next section will discuss each of these and how our implementation addressed each one.

8.1 The Core File Problem

One of the more interesting things to try on a machine that runs an EFS is to determine where the system should place the *core* file when an error arises that causes UNIX to create an image of the running process. This is a problem because of the way UNIX handles the *core* file. When UNIX decides to create a *core* file, it opens the current directory and creates the file called *core* in it. As it turns out, in our implementation, the *core*-file problem falls away.

Because the current directory is an *rinode* instead of a normal *inode*, the system does not see the difference and the normal UNIX mechanism takes place: UNIX opens a file in the current directory called *core*. The open is done with the *inode* of the current directory which in this case happens to be an *rinode*, the same basic frame work applies.

8.2 The Blocking Problem

As mentioned earlier, an EFS server maintains a pool of processes to act as agents for remote clients. The astute reader might ask, "Why not use a single agent process for each active EFS connection?" Arnovitz articulated the question further by asking: "If a single process is used as an agent, what happens if that process needs to block?"Arn84a The answer for a single agent process is that if the agent process must block, then all other requests from the client host will have to block waiting until the first request completes.

By using a pool of processes, an agent process can block on any single request without locking out further requests, either from another process on the same client system or from another client system altogether. If a second request comes in before the first one completes, a second agent process is found from the process pool to perform the new request.

8.3 Mount Points

A tricky problem in providing a fully transparent EFS is the correct handling of file system "back pointers" (i.e., links to shallower levels of the directory tree). Under UNIX, every directory initially contains two entries: "." (pronounced dot) and ".." (pronounced dot dot). The file "." is a link to the directory itself. This is provided as a convenience for the user. The file ".." on the other hand, is a link to the parent directory of the current directory. Thus if the user types:

```
cd foo
[any set of commands that don't
change the current directory]
cd ..
```

The user first enters the directory "foo," performs the given set of commands and then when the command "cd .." is executed, the user is placed in the directory from whence he originated. A problem arises if the directory "foo" is actually a remote directory. After the user executes the *chdir(2)* system call that moves his current directory to the remote machine, all pathnames are relative to the remote directory. Yet the directory in which the user lands on the remote machine contains an entry called: "..". When the second *chdir(2)* call is executed to perform the "cd ..", the server really needs to gate the user back to local host; *not walk back through the path on that machine pointed too by "..".*

This example demonstrates the need for the concept of a "mount point." When a file system is mounted remotely, the remote system must mark the in-core *inode* for that directory as a "mount point" for a remote file system. When a *namei* path walk is run and a "mount point" is crossed, the remote must decide if the processes that is walking this path is an EFS agent process or a normal user process. If it is a normal user process running on the remote host, the "mount point" is ignored and ".." is followed. However, if it is an agent process that is walking the path and the "mount point" originated from the client on whose behalf the agent is acting, *namei* must return a message back to the client system stating that the remainder of the path must be interpreted locally.

8.4 The *stat*(2) problem

In the UNIX Timesharing System, a system call is provided to get information about any file in the system. The call, *stat*(2), returns many pieces of information about the file. Two of those pieces are the *I-number* and the *device number* on which the file is stored. This information can be used by user programs such as the UNIX file copy utility, *cp*(1). One of the better human engineered features of *cp*(1) is that is does a *stat*(2) call on the source and the destination. It the compares the two *device numbers* and *I-numbers* for equality. If they are the same, *cp*(1) recognizes that the user has requested it to copy a file onto itself which would destroy the file, so it returns an error. Unfortunately, under EFS a file is no longer uniquely identified by just its *device number* and *I-number*. The comparison must now include a machine identifier. In EFS, the unique *network handle*, a 32 bit number, is returned as a new field in the *stat*(2) structure that contains the machine identifier.

Thus the *cp*(1) program had to be changed to use the new information. It now tests for equality of all three pieces that identify each file: *device number*, *I-number*, and the *machine indentifier*. To continue to meet the constraint that old code does not have to be recompiled, we retained the old *stat*(2) system call with its original system call number, but renamed it to *oldstat*(2). We then introduced a new *stat*(2) system call with a hidden third argument.[7] This

7. The third argument is maintained by the C library linkage. The user never sees it.

argument is the number of bytes that the *stat*(2) system call is expecting. As a result, all code that is compiled and linked will use the new call, as the old call is no longer in the distributed C library.

In this manner, we continued to have binary compatibility between the old user binaries and the new operating system product, but have warned users that a certain class of programs, that were written without EFS in mind may exhibit incorrect behavior when used across two different machines. In the case of *cp*(1), unless we had fixed it, it would have returned an error to a user stating that two files were the same file when in fact they were not. This behavior would not damage anything, but would cause a great deal of aggravation to the user.

8.5 The *ustat*(2) problem.

Some versions of UNIX contain a *ustat*(2) system call. This call returns information about a given file system, such as the amount of free space on a device that is passed as a parameter. The *ed*(1) text editor, performs a *stat*(2) on a file to see onto which device the file will be stored, and then calls *ustat*(2) to see if there is enough space to store the file. Unfortunately, with EFS the *device number* will be from the host where the file is stored, and *ustat*(2) will be local. *Ed*(1) was not working correctly within the multi-machine environment. The solution was to introduced a new system call, *rustat*(2) that takes the *machine identifier* as a parameter and then change *ed*(1) to use it.

Again, this is a function of a program that was trying to do something that is valid on a single machine, but not valid in a multi-machine environment. It is far easier to fix those few programs that do not work correctly, than it is to try to come up with a hack that fakes them into working.

9. What EFS Cannot Do

9.1 Remote Devices

In its present form, EFS provides transparent remote access only to disk files; it does not support remote devices. The reasons for this are twofold. First, when performing I/O operations on certain devices, *ttys* in particular, there is the possibility for indefinite delays to occur. For example, a read from a user's terminal device may block for hours while he steps out for a bite of lunch. In contrast, accesses to disk files, though they may cause the requesting process to block, always complete within a short period of time. In the EFS environment where a remote access is actually carried out by an agent process on the system where the resource physically resides, it is crucial that the operation be completed quickly. This is to ensure that the agent processes (a relatively scarce resource) do not all become blocked indefinitely, which in turn ensures that a remote client will always receive service in a timely manner.

The second difficulty involved with remote device access is the handling of the *ioctl* system call. The heart of this problem is that an *ioctl* command may call for the transfer of an arbitrary amount of data between the requesting program's address space and the device driver that implements the command. Further, the format of the data and the direction of transfer are completely determined by the device driver; that information is unavailable to any other part of the kernel. In the case where the device driver is on a remote machine, the local system cannot know *a priori* how much data, if any, to fetch from user space to send to the server. An obvious solution is to have the server request the appropriate amount of data from the client as needed, though it does incur the overhead of extra network transactions. Other approaches to this and other problems with remote devices are the subject of future EFS development work.

9.2 Diskless Nodes

The concept of "diskless workstations" is one that has gained increasing popularity of late. EFS in its current state does not directly address this issue. It assumes that each participating workstation has at least some local backing store containing its root file system, operating system image, paging area and various other files and programs necessary to boot up. Obstacles that stand between EFS and a completely diskless environment include remote paging, obtaining the initial boot image, and construction of an in-core root file system. An additional problem that arises where the network interface is an intelligent front-end processor is the initial down-loading that must precede any other network communications.

Though the authors believe that many users of diskless workstations rapidly find themselves in the market for add-in mass storage, this is another area where future development effort will be concentrated.

10. Summary

The principal motivation behind the MASSCOMP EFS project was the desire to offer our users fully transparent access to files on remote computers. The benefits yielded by this capability are more cost-effective use of mass storage capacity through reduced duplication of common files, and simplified management of shared databases. The goal of transparency has been achieved by building the concept of "remoteness" into the existing UNIX file system I/O architecture. In the same way that the existing UNIX *mount* system call attaches a local disk partition into the file system hierarchy, the new *rmount* system call attaches a subtree of a remote system's directory structure into the local name space. Because this functionality is built into the kernel, and not implemented as a user-level library, existing programs may access remote files without recompilation, and indeed without even being aware of their remoteness.

The implementation model is based on client-server interactions. Each file-related system call (as well as several internal system routines) has a client counterpart that takes responsibility for contacting an agent process on the remote system, passing the necessary information to it, and receiving the response. Each system supporting EFS also maintains a pool of kernel processes that perform the operations requested by remote clients. The principal kernel structure that describes a file, the *inode* is augmented with additional information to form an *rinode*. It is the *rinode* that informs a system call of a file's remoteness and provides the key that allows the client to identify the correct file to the server where it physically resides.

Inasmuch as the UNIX file protection scheme is based not on login names but on numeric user-ids, it was felt to be essential that all machines participating in an EFS share a common login-name to user-id correspondence. This is not only to avoid the complication of having to dynamically translate from one mapping to another, but also to promote the overall transparency of the system. Part of the EFS project included developing tools to assist individual system administrators in bringing their local file systems into correspondence with a network-wide mapping.

When EFS was being designed, consideration was given to what kind of communication services were necessary to support it. Because a fair amount of complexity in kernel modifications was anticipated, there was a strong desire to keep the communications interface simple. An examination of the then available protocols (TCP and UDP) revealed that neither was particularly suited to the task. TCP provides reliable connections but requires too much setup overhead to create a new connection for each client-server interaction. UDP is more suited to the client-server transaction model but does not provide for reliable delivery and has no capability to de-multiplex messages directed to different processes without requiring a socket per process. In view of these shortcomings, a reliable datagram protocol (RDP) was designed and implemented, providing guaranteed message delivery (or notification of failure) and the ability to dynamically create multiple low-overhead connections through a single socket.

One of the more challenging aspects of the EFS project was finding solutions to all of the new error situations that can arise in a distributed environment. Mechanisms were developed to detect and recover from errors resulting both from failure of a remote machine and from the disruption of network communications.

Areas not currently addressed by EFS are support for remote device files, and the operation of systems without local mass storage. Both of these areas are of interest to the authors and are the subjects of ongoing development efforts.

11. Credits

The authors would like the thank the members of the MASSCOMP staff who reviewed not only the ideas as they were developed, but the code. Finally, we would like to recognize and thank John Sundman of the MASSCOMP Documentation group who helped review this paper and certainly improved its structure.

Arn84a. Arnovitz, D., *Ideas about Distributed File Systems - Private Communication,* MASSCOMP (July 1984).

Bil83a. Billingsley, Giles and Keller, Ken, "Program Reference for KIC2," *Electronics Research Laboratory Memorandum* (December 1983).

Bro82a. Brownbridge, D.R., Marshall, L.F., and Randell, B., "The Newcastle Connection," *Software Practices and Experiences*, pp.1148-1162, Computing Laboratory - University of Newcastle upon Tyne (July 21, 1982).

Buc84a. Buck(ed), D. A., */usr/group UNIX Standard,* /usr/group (March 1984).

Coh76a. Cohen, Ellis, "Program Reference for SPICE2," *Electronics Research Laboratory Memorandum* **ERL-M592** (June 14, 1976).

Duf82a. Duff, Thomas, *An Extended File System for Unix Version 7,* LucasFilm (1982).

Gre83a. Greenwald, Michael and Allen, Larry W., *Ideas for an Remote Virtual Disk - Private Communication,* CSL - MIT (October 1983).

IEEd)a. IEEE, P1003 Working Group, *IEEE P1003 POSE - Portable Operating System Environment - Draft,* IEEE (January 1986 (expected)).

Kel84a. Keller, Ken, "Program Reference for HAWK," *Electronics Research Laboratory Memorandum* (June 1984).

Lef82a. Leffler, S. J., Fabry, R. S., and et., al., *4.2 BSD System Manual,* Computer Systems Research Group, EECS-UCB (June 1982).

Lef82b. Leffler, S. J., Fabry, R. S., and et., al., *4.2 BSD Interprocess Communication Primer,* Computer Systems Research Group, EECS-UCB (June 1982).

Lyo85a. Lyon, Bob, Sager, Gary, and et., al., *Overview of the Sun Network File System,* Sun Microsystems, Inc., Mountainview, CA (January 1985).

Pos80a. Postel(ed.), Jon, "User Datagram Protocol - UDP," *RFC 768*, USC/Information Sciences Institute (August 1980).

Pos81a. Postel(ed.), Jon, "Transmission Control Protocol - TCP," *RFC 793*, USC/Information Sciences Institute (September 1981).

Pos81b. Postel(ed.), Jon, "Internet Protocol - IP," *RFC 791*, USC/Information Sciences Institute (September 1981).

Pos82a. Postel(ed.), Jon, *Internet Protocol Transition Workbook,* SRI International, Menlo Park, CA (March 1982).

Qua83a. Quarles, Thomas, "SPICE3: User's Manual," *Master Report, EECS - UCB* (1983).

Rit74a. Ritchie, D.M. and Thompson, K., "The Unix Time-Sharing System," *Communications of the ACM* **17(7)**, pp.365-375, The Bell Telephone Laboratories (1974).

Rit78a. Ritchie, Dennis M., "The Unix I/O System," in *Unix Programmers Manual*, The Bell Telephone Laboratories (November 12, 1978).

Wei84a. Weinberger, P.J., "The Version 8 Network File System," *1984 Unix Summer Conference*, AT&T Bell Laboratories (June 1984).