A Distributed File System For UNIX

Matthew S. Hecht, John R. Levine, Justin C. Walker*
Interactive Systems Corporation
8689 Grovemont Circle
Gaithersburg, MD 20877
and
*441 Stuart Street
Boston, MA 02116

A DISTRIBUTED FILE SYSTEM FOR UNIX

(Extended Abstract)

Matthew S. Hecht
John R. Levine*
Justin C. Walker

INTERACTIVE Systems Corporation
8689 Grovemont Circle
Gaithersburg, Maryland 20877
(202) 789-1155
allegra!ima!matthew

and

*441 Stuart Street
Boston, Massachusetts 02116

November 1983
revised February 1984

Summary

We describe a design for achieving user-level transparent access to remote files in a local area network of homogeneous UNIX# systems. The design features (1) a *remote mount* system call that allows the mounting of a remote directory onto a local directory; (2) a specialized *remote function call* mechanism that allows one UNIX kernel to call functions in another; and (3) a connectionless scheme for communication between hosts. This note describes work in progress.

1. PROBLEM STATEMENT

The problem we solve here is how to make access to local and remote UNIX files (in a local area network) indistinguishable to users; that is, the syntax and semantics of local and remote file access are identical. For example, the user of a command like

```
cp x y
```

where x and y are arbitrary pathnames, can be oblivious to the location of files x and y (one or both may be local or remote) since the underlying UNIX system calls do the right things in either case.

Transparent access to remote files in a local area network of UNIX systems provides new opportunities for file sharing. Independent UNIX systems can share files, diskless workstations can obtain file service from another UNIX system, and workstations with low capacity disks can share databases. Transparent access also allows files to be relocated without breaking programs.

Our solution features a remote mount system call, and roughly places the interface to remote files at the i-node function level of the kernel. This work contains a novel mix of implementation ideas, yielding a simple, clean, and practical solution. Instead of assigning a remote file-server process to a local user process, we use a pool of kernel file-server processes that feed from a

UNIX is a trademark of Bell Laboratories

common request queue. In addition, we use a connectionless (datagram-based), specialized remote function call mechanism that draws ideas from work by Nelson [6].

Extant work on distributed file systems for UNIX is extensive and growing; we comment on a few related papers here. Chesson [1] attributes early work on remote mount to Lucas and Walker [5] at National Bureau of Standards. Glasser and Ungar [2] at Bell Labs studied a read-only, connectionless datagram scheme for remote mount. Our work is similar to that of Luderer and others [4] at Bell Labs on S-UNIX/F-UNIX and to a design of Plexus Computers called Network Operating System (NOS) by Picard described by Groff [3]. However, these papers indicate a different process architecture with communication based on virtual circuits. Also, the S-UNIX/F-UNIX work makes the cut at the system-call interface, not at the i-node interface. The COCANET project of Rowe and Birman [8] at UC Berkeley is more ambitious than our work; we do not handle remote processes. The LOCUS project of Popek and others [7, 9] at UCLA is also more ambitious; we do not consider replicated files nor transactions nor remote processes. To our knowledge, the only above-mentioned projects in operation now are NOS and LOCUS.

2. NETWORK NAME SPACE MODELS

Although various models exist for extending the UNIX filesystem name space to a network, we have chosen the remote mount model because it has properties that we desire, such as location transparency and ability to specify a remote root file system. The models include:

- Host- and ~~Path~~ Route-Qualified Absolute Pathnames
- Global Root (Super-Root)
- Per-Host Subdirectories
- Symbolic Links
- Remote Mount
- Combinations of the above

We explain these models with familiar directed graph terms. To get started, it is convenient to pretend that a single UNIX file system is a rooted, connected, directed tree (typically pictured as a triangle), with arcs directed from the root. Recall that an arc is a pair of points (tail, head) drawn as an arrow from tail to head (associate "head" with "arrowhead"). A single UNIX file system is not really a tree because of parent ("..") entries and links.

The various network name space models explain how to draw a network filesystem tree, and sometimes suggest implementations. *Uucp* has host- and ~~path~~ route-qualified absolute pathnames. Picture separate trees, one for each host system. A file name may consist of a host name followed by the character '!' followed by an absolute pathname, in which case the pathname is sent to that host for resolution. Or, a path of '!'-terminated host names specifying a route may precede an absolute pathname.

The super-root (global root) model makes each existing root file system a subdirectory of a new super-root. A super-root qualified pathname can begin with a special character, say '@', followed by a host name, in turn followed by an absolute pathname.

COCANET [8] supports per-host subdirectories, where the root of each host, in addition to its local subdirectories, has a subdirectory for each host, including itself.

Symbolic links [11] generalize normal, intra-filesystem UNIX links. A special i-node type, called a symbolic link file, contains a pathname. When the name resolution function *namei*() encounters this file while resolving a name, the contents of the symbolic link file are prefixed to the rest of the name (if any). If the symbolic link file contains an absolute pathname, the resulting name is interpreted relative to the root directory. Symbolic links allow inter-filesystem links on the same host. In addition, symbolic links to directories are permitted by a superuser. Consequently, we can have network links if we support a symbolic link file that contains a host- or

route
~~path~~-qualified absolute pathname.

The remote mount model generalizes the (normal) mount model. The *mount* command allows us to attach a tree, the root of another file system, onto a directory of our root file system. The *rmount* command allows us to attach a remote tree, a directory in a remote file system, onto a directory of our root file system. We can mount the tree with read-write access or read-only access.

In addition, combinations of the above models are possible. For example, we can use either symbolic links or remote mounts or a combination of the two to provide per-host subdirectories and to customize the resulting name space.

In this note we omit a discussion of the problems, advantages and disadvantages of each model. However, we do point out that interhost linking (by symbolic links or remote mounts) is potentially unsafe because it can introduce unsafe loops in the network tree that can result in undesirable behavior by tree walkers like *find*, *du*, and *cpdir* (cp -R). Also, commands like *mvdir* may not preserve loop-freedom. There are regimes of constraints for interhost linking that are both safe and flexible in practice.

3. SKETCH OF DESIGN

Our design consists of modifications to the UNIX System V kernel.

3.1 System Calls

We have added several new system calls, including

```
rmount(hostname, rdir, ldir, ronly), and
rumount(dir).
```

Rmount mounts remote directory *rdir* of system *hostname* on local directory *ldir*, where *ronly* specifies read-only or read-write access, and *rumount* unmounts a remote directory from the given local directory. The *rmount table*, analogous to but separate from the mount table, helps the kernel function *namei*() cross rmount points. A successful rmount installs a TO entry in the local rmount table and a FROM entry in the appropriate remote rmount table.

Figure 1 shows an example of a remote mount where directory /g of host green is mounted onto directory /b of host blue. Now, /b/w/z is a valid pathname on host blue.

3.2 Client (Request) Side

We distinguish i-nodes that represent locally stored files from i-nodes that represent remotely stored files. The latter we call *surrogate i-nodes*. If a pathname crosses an rmount point, the local system sends a *namei*() request with the remainder of the pathname to the remote system. The remote system resolves the pathname, calls *iget*() there, and sends, as the *namei*() response, a *handle* that identifies the remote i-node. The requesting system uses the (host, handle) pair to define a surrogate i-node, which it installs in the local *i-node table*. Only the local *file table* is used during access to a remote file, not the remote file table.

Operations on surrogate i-nodes generally translate into remote function calls. Currently, these operations include *access(), chmod(), chown(), closef(), ioctl(), iput(), itrunc(), iupdat(), lockf(), link(), maknode(), namei(), openi(), owner(), plock(), prele(), readi(), rmount(), rumount(), seek(), stat1(), unlink(), utime(), writei()*, and a few others. In addition, we are experimenting with more comprehensive operations to lessen network traffic. A new stub-manager module, which we call the *agent*, consolidates the remote function call mechanism for the client, and hides request/response message formats from caller modules.

3.3 Server (Response) Side

Incoming remote requests for local file service are enqueued on a common request queue. A pool of kernel processes handle remote requests to local i-nodes. The code for a server looks like:

```
for (;;) {
        wait for a request;
        dispatch function;
        send response;
}
```

3.4 Transport Protocol: Packet Exchange Protocol

As the transport protocol, we currently use the Packet Exchange Protocol (*PEP*) [10], since it provides most of the request/response model assumed by our design. This protocol uses a *socket* abstraction, a network address where processes can send packets and rendezvous, and a *packet* abstraction, a message container.

Figure 2 shows the software layers in the current prototype, and Figure 3 shows innards of an implementation of the PEP layer. In Figure 3, a circle denotes a process, a rectangle denotes a module, an arrow denotes a function call, and a "+" denotes missing details. Because PEP is not well-known, we include a brief description of it here.

PEP has a three-function interface. *WaitRequest()*, called by a server, waits at a known socket for an incoming request packet. *SendRequest()*, called by the agent, transmits a request packet to a known server socket, and blocks until it returns either a response packet or error. If necessary, the packet is retransmitted a specified number of times, waiting a specified interval between retries. *SendResponse()*, also called by a server, transmits a response packet to a socket and does not block. The caller of *SendResponse*(), a server here, is responsible for detecting duplicate requests and retransmitting responses.

Network layer code enqueues arriving PEP packets onto the PEP receive-queue, and wakes up the PEP receiver process. This process dequeues packets, matches them to *SendRequest*() and *WaitRequest*() entries in its match table, and wakes up the appropriate sleeping server or client process.

4. PROBLEMS ADDRESSED

In this section we give an abbreviated description of various design problems that arise and indicate solutions for some of these.

4.1 Rmount Implementation

Problem areas that arise in implementing the rmount model include multiple hops, "..", rmount loops, and security. We comment only on the first two.

In a multiple-hop design, messages are transshipped through intermediate hosts. A surrogate i-node on host H1 may point to a surrogate i-node on host H2, which in turn may point to the real i-node on host H3. However, it is possible to eliminate multiple hops and produce a single-hop design where messages are sent directly to the end host. In a multiple-hop design, *namei*() can cross an rmount point in a server by passing the remaining pathname to the next host. In a single-hop design, when crossing an rmount point a server *namei*() can return the identity of the next host and the pathname progress to the original client host, which sends the remaining pathname to the next host. Thus, we can stack the host path walked by system calls *chdir* and *chroot*, and pop retraced subpaths.

To ascend (with "..") an rmount point, we remember the previous jump-off point of this *namei*() and use a FROM entry in the server rmount table. Thus, we can allow more than one local directory per host to rmount the same remote directory. With a multiple-hop scheme, we can reflect the remaining pathname back to the requestor host at the jump-off point, which is sent in the original request. With a single-hop scheme, we can stack the rmount jump-off points and uniquely retrace "..".

4.2 PEP Model

PEP, while useful here, does not exactly match our application of remote and nonidempotent function calls. PEP is a request/response model for idempotent requests. While *idempotency* in this context normally means that reevaluating a duplicate request produces the same value and state as evaluating the original request, we can relax it to mean that request reevaluation is harmless, as in the case of a time server request. The function *namei*(), for example, is nonidempotent: if it locks an i-node the first time called, then the server process will block the second time called as UNIX does not remember the identity of the locking process. Consequently, the application layer RF solves some protocol problems that perhaps belong to a transport layer, like duplicate request detection.

4.3 Process Architecture

Three problems associated with our process architecture are side-stepping the notion of a session, cleaning up server state upon actual or presumed death of a client host that holds server host resources, and preventing server deadlock. We omit discussion of these here.

4.4 Miscellaneous

To avoid unnecessary packet copying, one can add a little slop space before a buffer to hold protocol headers, or use a scatter/gather data structure that network devices support.

5. CONCLUSION

The current design has limitations that. while not insurmountable, need mention. First, we consider a network of homogeneous hosts so that an executable from a remote host runs on the local host. With heterogeneous hosts, this is not the case. Second, we consider a partitioned, but not replicated, file system. Third, we do not consider remote processes, executing a command on a remote host, functionality we plan later. These intentional limitations let us chew the first bite.

Based on our experience to date with a prototype of this design, our conclusion is that a connectionless rmount approach for a distributed UNIX file system is practicable. More performance experience and code tuning is needed before we can remove the "b" and "e" from that last word.

Acknowledgment

We gratefully acknowledge helpful discussions and comments on the ideas in this paper by Lee Cooprider, Dan Franklin, Gary Gordon, Brian Lucas, David Marx, and Joe Sokol.

References

[1] Chesson, G., Borden, B., and Gurwitz, R., "UNIX Systems on Local Area Networks," tutorial, *1984 UniForum Conf.* (January 1984).

[2] Glasser, A., and Ungar, D., *Fifth Berkeley Workshop on Distributed Data Management and Computer Networks* (February 1981).

[3] Groff, J.R., "Modified UNIX System Tames Network Architecture," *Electronics*, pp. 159-163 (September 22, 1983).

[4] Luderer, G.W.R., *et al.*, "A Distributed UNIX System based on a Virtual Circuit Switch," *Proc. 8th Symposium on Operating Systems Principles*, Pacific Grove, Calif., pp. 160-168

(December 1981).

[5] Lucas, B.W., and Walker, J.C., unpublished work, National Bureau of Standards (1976).

[6] Nelson, B.J., "Remote Procedure Call," report number CSL-81-9, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, Calif. (May 1981).

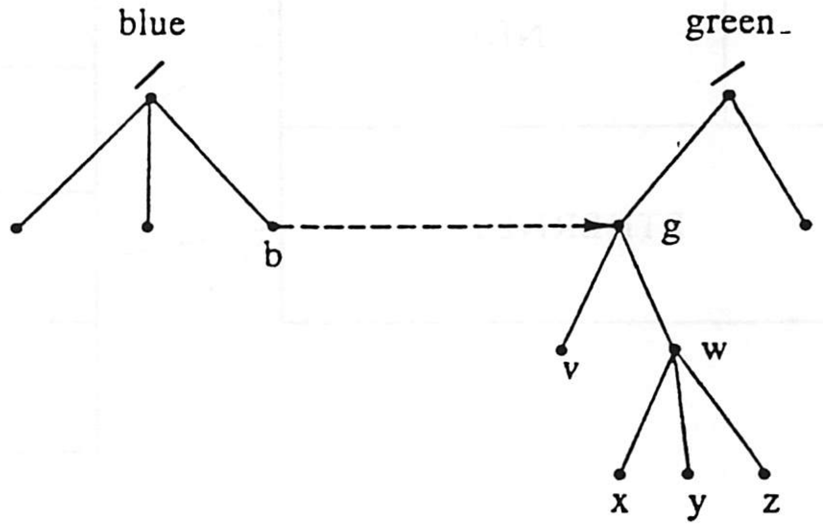
[7] Popek, G., *et al.*, *Proc. 8th Symposium on Operating Systems Principles,* Pacific Grove, Calif., pp. 169-177 (December 1981).

[8] Rowe, L.A., and Birman, K.P., "A Local Network Based on the UNIX Operating System," *IEEE Trans. on Software Engr.*, Vol. SE-8, No. 2, pp. 137-146 (March 1982).

[9] Walker, B., *et al.*, "The LOCUS Distributed Operating System," *Proc. 9th Symposium on Operating Systems Principles,* Bretton Woods, New Hampshire, pp. 49-70 (October 1983).

[10] "Internet Transport Protocols," Xerox System Integration Standard XSIS-028112, Chapter 8, pp. 49-51, Xerox Corporation, Stamford, Connecticut (December 1981).

[11] *UNIX Programmer's Manual,* 4.2 Berkeley Software Distribution, Univ. of Calif., Berkeley, Calif. (August 1983).



at blue: `rmount [-r] green /g /b`

Figure 1. Example of a Remote Mount.

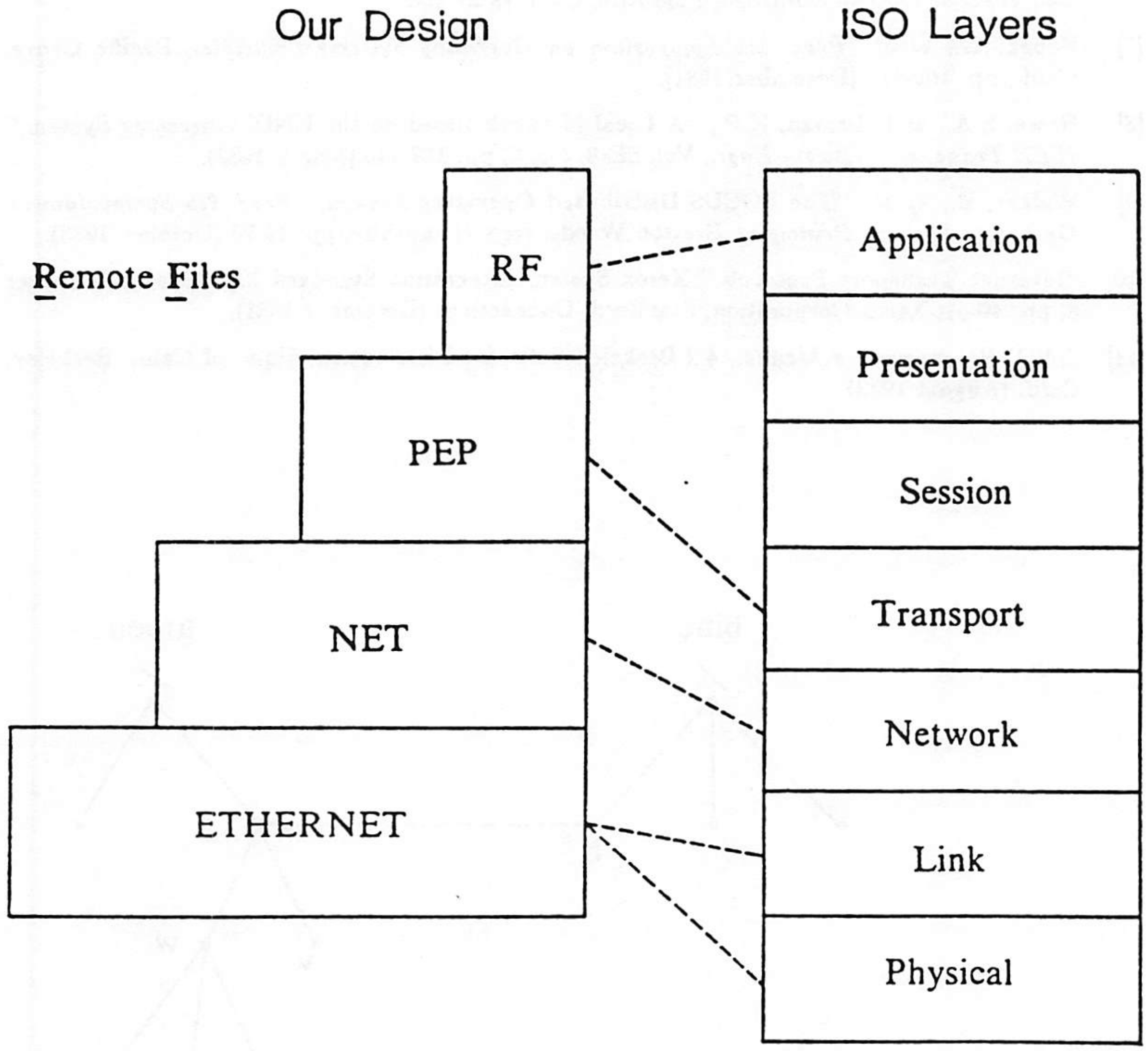


Figure 2. Software Layers in Prototype.

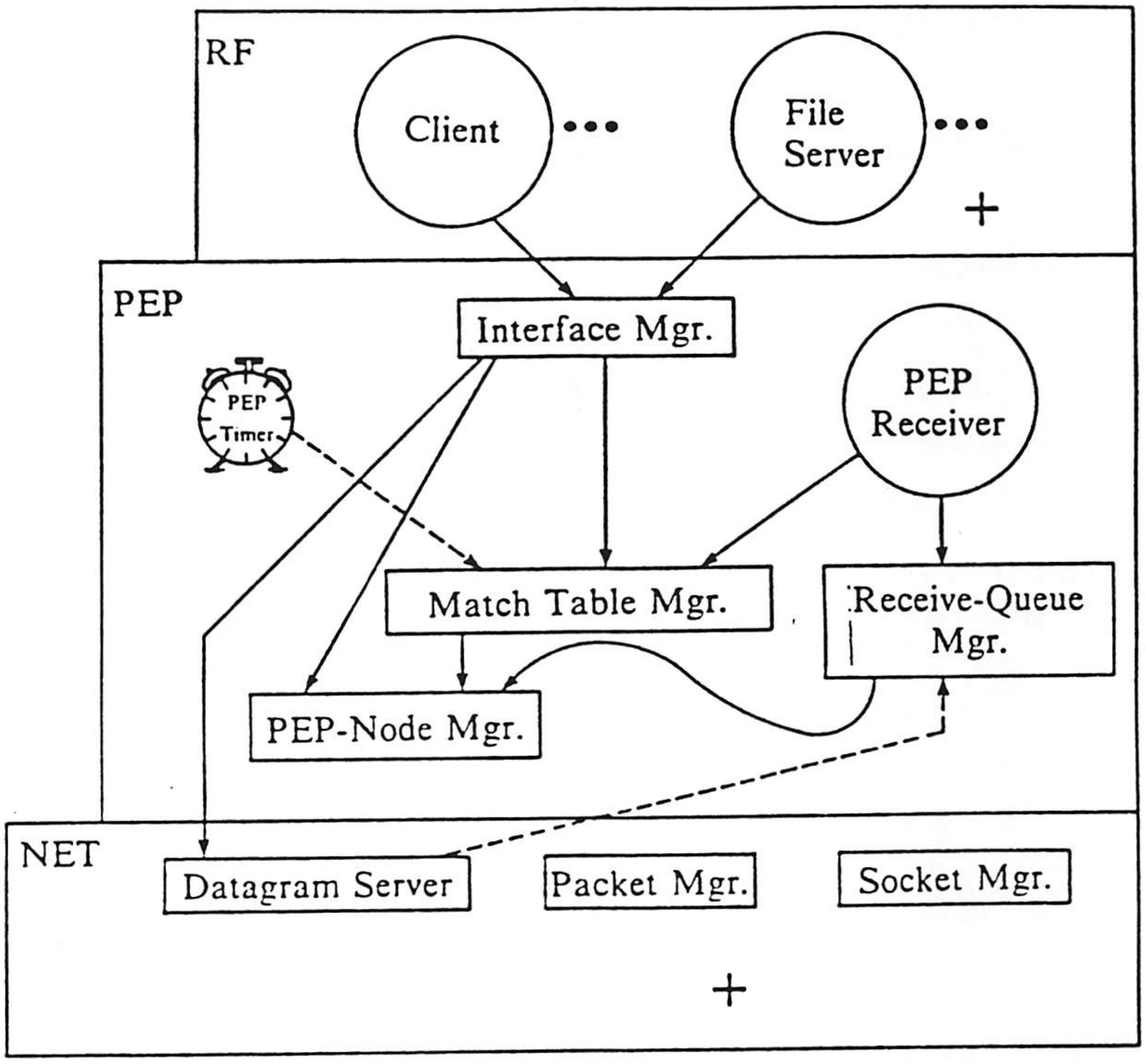


Figure 3. Packet Exchange Protocol (PEP) Innards.