2-28-1986

# Distributed file systems for Unix

Edward Ford

Rochester Institute of Technology
School of Computer Science and Technology

# Distributed File Systems for Unix

by
Edward Ford

A thesis submitted to the
Faculty of the School of Computer Science and Technology
in partial fulfillment of the the requirements for the degree of
Master of Science in Computer Science

Approved by:

Michael J. Lutz 2/28/86
Professor Michael J. Lutz

Lawrence A. Coon 2/28/86
Professor Lawrence A. Coon

Margaret M. Reek 2/28/86
Professor Margaret M. Reek

Title of Thesis: Distributed FILE SYSTEMS FOR UNIX

I _____ hereby (grant/deny) permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

or

I Edward Ford prefer to be contacted each time a request for reproduction is made. I can be reached at the following address:

_____

_____

Date: 3/3/86

# Distributed File Systems for UNIX(*)

Edward Ford

ABSTRACT

With the advent of distributed systems, mechanisms that support efficient resource sharing are necessary to exploit a distributed architecture. One of the key resources UNIX provides is a hierarchical file system. Early efforts supported distributed UNIX systems by copying files and sending mail between individual machines. The desire to provide transparent mechanisms on which distributed systems access resources has propelled the development of distributed file systems.

This thesis presents a brief history of the development of distributed systems based on UNIX, and surveys recent implementations of distributed file systems based on UNIX. The IBIS distributed file system is an example of the latter. The original capabilities of IBIS are discussed and modifications that enhance these capabilities described.

Computing Review Subject Codes

File Systems Management / Distributed File Systems (CR D.4.3)
Distributed Systems / Network Operating Systems (CR C.2.4)
Security and Protection / Authentication (CR D.4.6)

* - UNIX is a trademark of AT&T Bell Laboratories.

Key Words and Phrases

operating system (OS), local area network (LAN), distributed file system (DFS), remote file access (RFA), remote file server (RFS), super-root, remote-mount, location transparency, access transparency, server, client, remote procedure call (RPC).

Acknowledgements

I would like to give special thanks to my advisor, Professor Michael Lutz, for his support and guidance on this project.

TABLE OF CONTENTS

## 1. Introduction

The emergence of several distributed file systems based on UNIX is evidence of the desire and need to share hardware and software resources among users in a network. Even early telecommunications networks (i.e. the SABRE airline reservations system [Perry 1961]) reflected the need to share a central computing facility among geographically separated users.

There are numerous benefits to a distributed system, including resource sharing, system availability and reliability, and improved throughput and response time [Thurber 1979]. Depending on the design of a distributed system, not all advantages of a distributed system are realized or desired. For example, a distributed control system for an airplane may be designed to achieve high reliability (fault tolerance) and rapid response over short time spans. On the other hand, the primary goal of a distributed software development environment may be resource sharing with performance and reliability as secondary goals.

The advent of low cost local area network (LAN) implementations (i.e. Ethernet) and standardized communications protocols (i.e. TCP/IP) has propelled the development of networks [Tannenbaum 1981]. The evolution of UNIX networks has paralleled the development of networking technology and data communication protocols.

One of the earliest networking systems for UNIX was the Unix-to-Unix copy program (UUCP) system. UUCP connects individual Unix systems by modems and the telephone system. Most versions of UUCP support file transfer and remote command execution [Nowitz 1980]. The latter is used heavily for the exchange of mail and news among a world wide network of Unix sites (USENET).

Merely interconnecting sets of computer systems and providing primitive software mechanisms to access remote resources does not enable efficient resource sharing. In particular, it inhibits the most common form of sharing found in time sharing systems: a unified file system. The primary goal of a distributed file system is to enable resource sharing via the file system. Several distributed file systems based on UNIX achieve this goal by expanding the file system hierarchy across the network. Efforts in this area include the Newcastle Connection, COCANET, S/F - UNIX, 4.2 / 4.3 Remote Mounts, SUN Network File System, LOCUS, and IBIS. This thesis explores distributed file systems that are based on UNIX, and describes local modifications to improve Purdue's IBIS distributed file system for use on RIT's facilities.

Section two of this thesis defines concepts and attributes of distributed file systems. Section three describes in detail several UNIX remote file access systems. Section four describes enhancements to IBIS. Section five discusses

discrepancies within IBIS, and section six suggests potential changes to IBIS. Finally, section seven contains concluding remarks.

## 2. Concepts and Definitions

File access transparency hides the details of network accesses for remote files so that system calls operating on files (e.g. open, read, and write) do so in a consistent manner regardless of the location of files. It also eliminates the need for special primitives to provide access to remote files. However, the host where a file resides (file location) must still be provided in any file access primitive [Tichy 1983]. The file location may be syntactically specified by including the host name as part of a file name. Syntactic approaches may introduce file naming conventions that collide with other system or application software*. Alternately, a UNIX special file pathname component can semantically map the file location based on major and minor device attributes.

File location transparency eliminates the need to specify (syntactically or semantically) the host where a file is located. Choosing the host where a file resides becomes the responsibility of the file system manager [Howard 1985]. The two primary strategies used by a file system manager in determining the location of a file are file replication and file migration.

File replication propagates copies of files (replicates) to improve the efficiency of read access and to

* - Such as UUCP file naming and the csh history mechanism.

increase system reliability. In particular, read access to files (especially directories) near the root of the file system benefit from file replication [Tichy 1983]. File migration, on the other hand, attempts to speed access time by migrating a copy of the file to the node issuing the reads and writes. However, updates to a frequently changing replicated file are very expensive [Barbara 1984]. To address this problem, one copy of the file can be designated the primary copy, and the remaining replicates invalidated in exchange for remote access of the primary copy [Tichy 1983].

UNIX remote file access packages can be partitioned into three categories based on the model used to provide the distributed file system [Hunter 1985], namely super-root systems, remote mount systems, and global file name space systems. Systems such as the Newcastle Connection, COCANET and IBIS are classified in the super-root category because they allow potential access to an entire remote file system. Normally a special file is used to extend the root directory so that it spans all nodes in the network [Hunter 1985]: hence, the term super-root. Systems based on a super-root approach support file access transparency.

Systems such as the Bell distributed UNIX system (S/F-UNIX), the SUN Network file system, and the 4.2 / 4.3 Remote mount facilities fall into the remote-mount category. These systems use a variation of a mount system call to "virtually

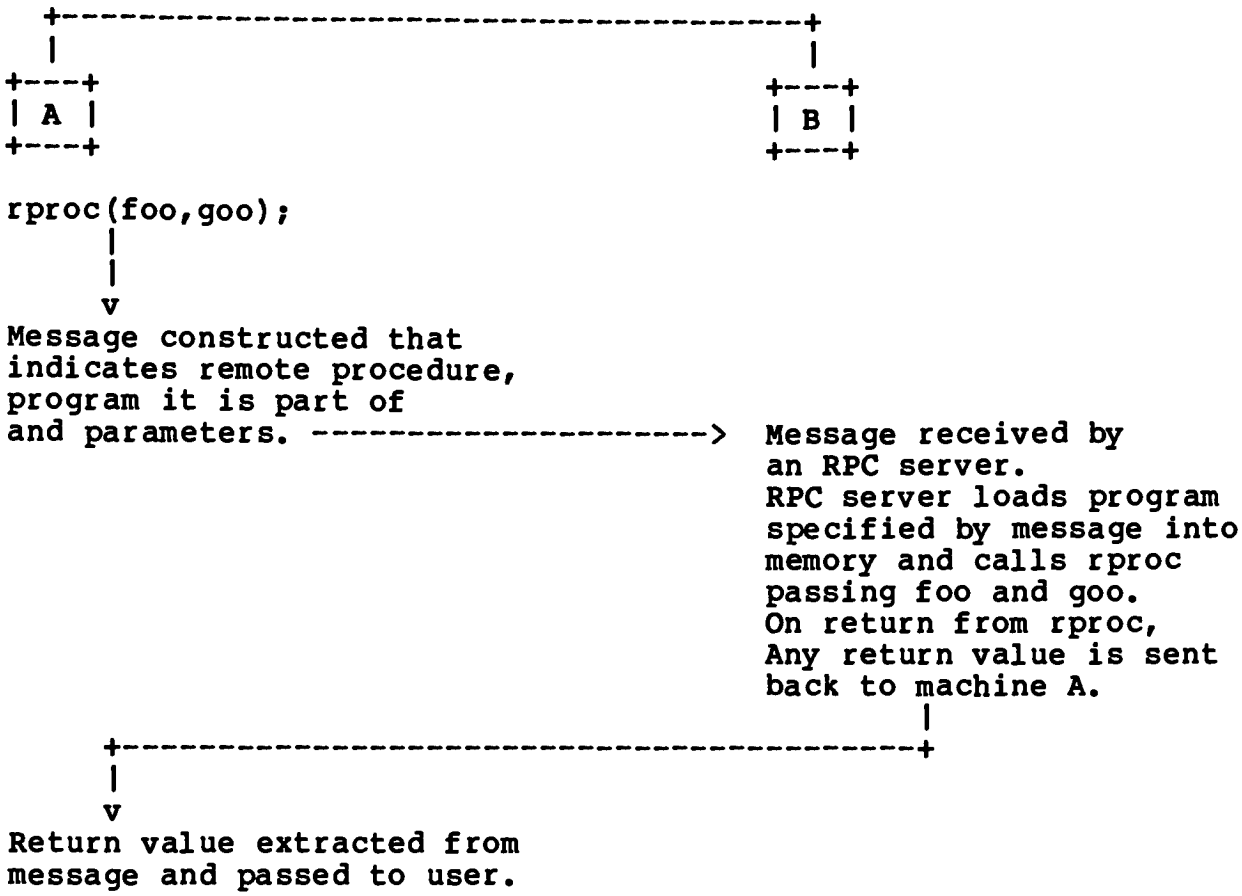
mount" a portion of a remote file system. Remote file access systems that use the remote-mount approach support file access transparency and file location transparency*. A remote-mount allows more control than the super-root approach so that only specified subtrees of the remote file system are accessible [Hunter 1985].

The penultimate stage in the development of distributed file systems is represented by systems that support a global file name space. In this case, the same file name is used to reference a file from any machine in the network. Thus, the system provides the illusion of a single machine even though there are many in the network. Systems based on a global file name space support file access transparency and file location transparency. A system that falls into this category is the LOCUS system.

The software components that implement a distributed file system reside on multiple machines a network. Often these components communicate with one another by a remote procedure call (RPC) protocol. Conceptually, an RPC represents a "network JSR", which distributes a computation (and its parameters) [Peterson 1983]. The following figure illustrates an RPC from machine A to the procedure "rproc" on machine B:

---

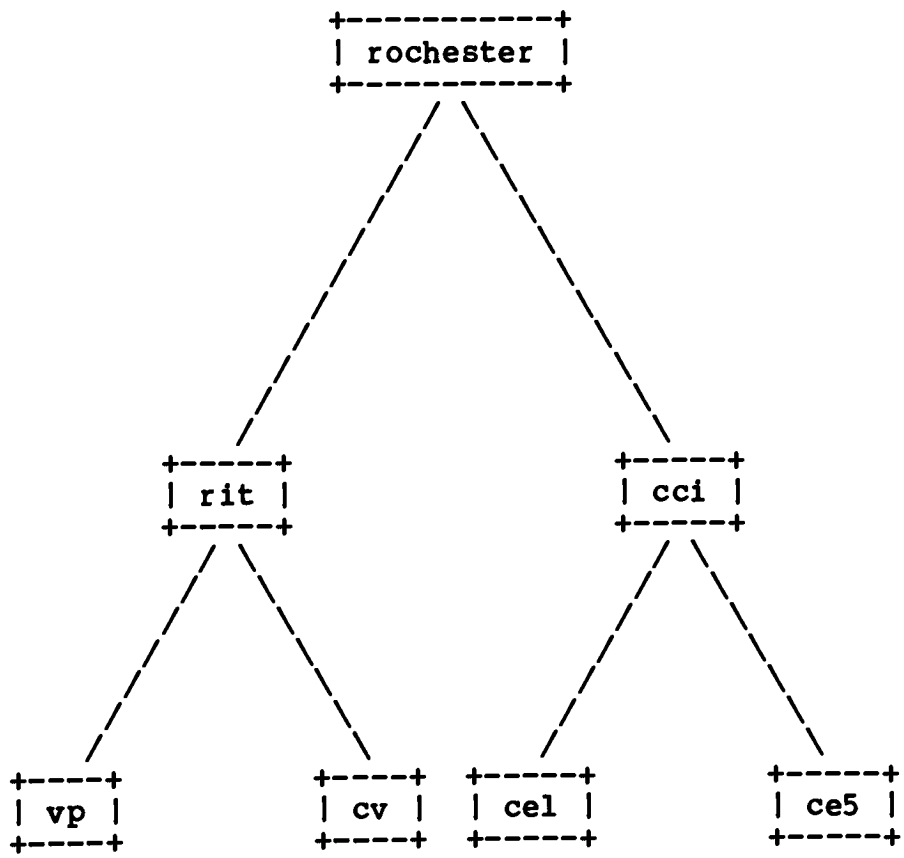
* - Only for regular users. The system administrator doesn't have file location transparency since the location of a remote file system must be specified in order to mount it.


A
B
rproc(foo,goo);
Message constructed that indicates remote procedure, program it is part of and parameters.
Message received by an RPC server.
RPC server loads program specified by message into memory and calls rproc passing foo and goo.
On return from rproc, Any return value is sent back to machine A.
Return value extracted from message and passed to user.

# 3. UNIX Transparent Remote File Access Systems

## 3.1. Newcastle Connection

The Newcastle Connection was developed at the University of Newcastle upon Tyne, England. The Newcastle Connection (hereafter NC) uses the super-root approach to map a naming structure onto the file system hierarchy [Brownbridge 1982]. The following is an example NC hierarchy:

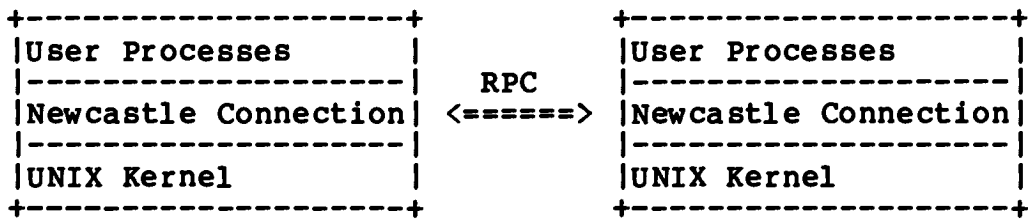



Leaf nodes represent actual systems, and interior nodes represent the naming hierarchy that "glues" the network together. The file foo on cel is referenced from vp with

the path:

/../../cci/cel/foo

The NC consists of user level modifications [Brownbridge 1982]. Actual inter-processor communication occurs at lower levels within the UNIX V7 kernel, so that even though the logical layering is as illustrated below, low level protocols are supported within the kernel. In terms of hardware, the NC runs on a set of PDP 11/70's and 11/23's that communicate via a Cambridge Ring [Brownbridge 1982].

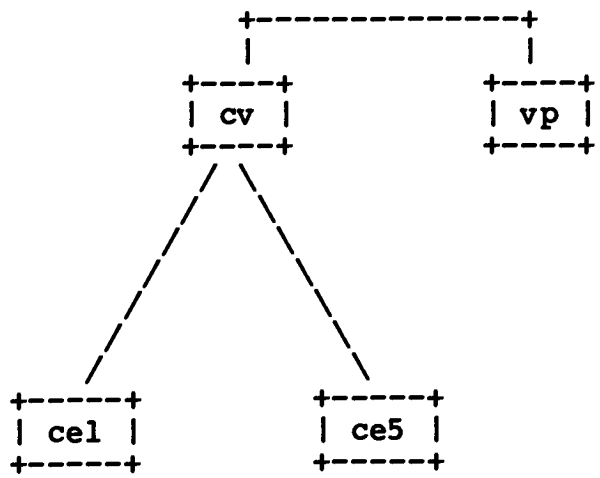



The NC intercepts all system calls that deal with pathnames and file descriptors. Pathnames that reference objects within the local system naming structure are passed to the local kernel for service. If a pathname component corresponding to a remote system is encountered, then an RPC is issued to the remote system. A remote pathname component is marked with the address of the corresponding remote system, and this address is presented to the RPC as routing information. The actual remote file access is performed by a file server process. One such server process is allocated by a remote server "spawner" for each client process on the local system. In this way, the file server is an extension

of the user's local environment, and any changes in the local environment are mirrored by the file server. For example, if a user process forks, then all connected file servers also fork [Brownbridge 1982].

Communication with a file server is the responsibility of the RPC protocol. The actual request may span several systems in the network, and the RPC guarantees reliable communication ensuring that calls are not lost. RPC requests are packaged with the same information as an equivalent local call plus user-id information [Brownbridge 1982].

The NC naming structure is independent of the physical network topology. For example, given the NC naming hierarchy (shown above) for rochester, the physical network topology could be as follows:




In this case, all traffic to cel and ce5 would flow through cv, therefore, cv fulfills a gateway function.

However, the naming structure would still reference file foo on cel from yp with the path:

/../../cci/cel/foo

In this case, cv is acting as an invisible transport agent. To accomplish this, all decisions pertaining to routing are performed within the NC, and are completely transparent to the user [Panzieri 1985].

Thus, the NC isolates most user programs from changes in the network topology by assuming that the naming structure is stable and unique. This is possible because the underlying routing is not bound to the file system naming hierarchy, but to special files that map remote systems. This mechanism means the communication driver and device layers need only have knowledge of adjacent nodes; routing concerns are ignored at this level. However, in practice, as a request is passed around the network, the naming information (described in a pathname) has to be changed by removing the local "prefix" that corresponds to the local system issuing the request. Hence, the remote "suffix" is passed to the remote system. Therefore, within the RPC, some logical coupling (independent of physical routing) does exist between a file's location and its name. This distinction between logical naming and physical routing is an advantage. In this way, topology changes in a network have minimal impact on applications. Only the NC supports this advantage.

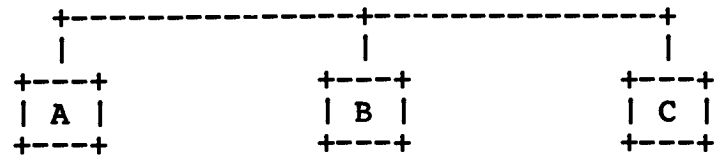
Under the NC, access control mechanisms are "extended" so that local user requests are distinct from remote user requests. This degree of control is accomplished by expanding the user name space. Normally, user identifiers under UNIX only pertain to local users. For example, the identifiers x and y are allocated to local users x and y. However, the NC treats user requests from machine A on behalf of user x distinct from local user x requests by assigning a separate id (in this case A/x) for user x on machine A. This scheme supports flexible, autonomous administration of systems in an NC network.

## 3.2. COCANET

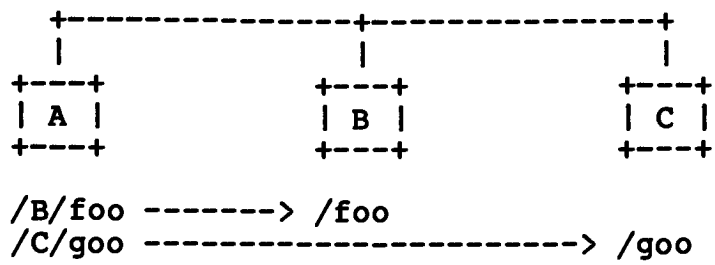
Another system that uses a super-root approach is the COCANET UNIX system. COCANET UNIX was developed at the University of California at Berkeley in 1981 [Rowe 1981] and is a distributed operating system supporting distributed process groups, remote execution, and remote file access. COCANET consists of modified system calls (in the kernel), user level servers, and a network manager (i.e. a kernel-resident process).

COCANET uses almost the same concept as the NC to map the network onto the UNIX file system. The difference is that in COCANET the file system only grows "below" the root. In contrast, NC file systems can also grow "above" the root (i.e. "/.."). In addition, unlike the NC, COCANET does not make a distinction between a file name and its physical location in a network.

Given the network:




Users on system A reference files foo on B and goo on C as follows:

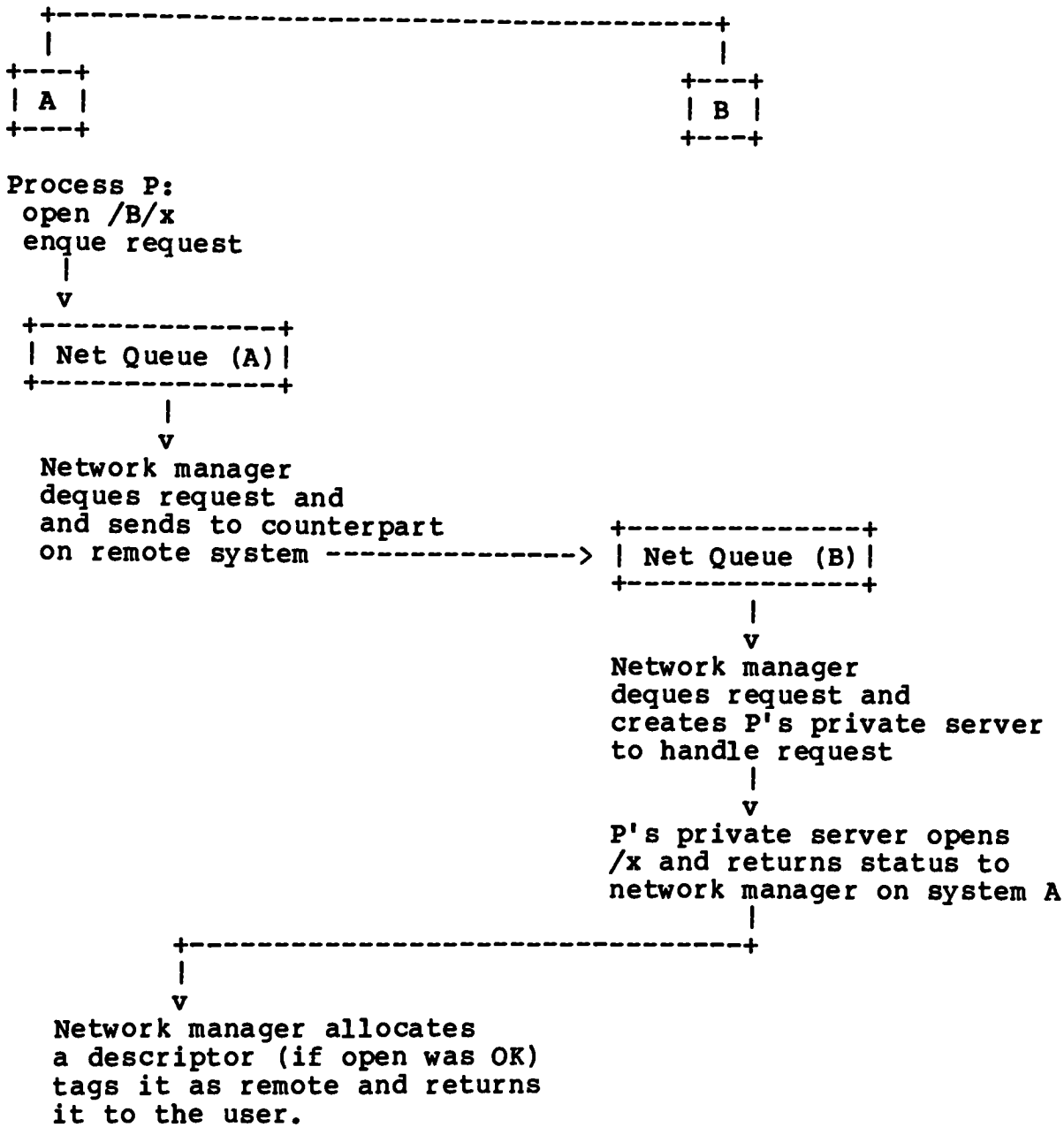



Similarly, file /moo on system A is referenced from system B as: /A/moo. Access is symmetric, so A can access /moo by two names: /moo and /A/moo.

Access to remote files in COCANET is actually handled by server processes on remote machines [Rowe 1981]. Private servers handle operations for a single user process (client). Shared servers handle operations for multiple user processes (clients). The remainder of this section describes the use of these servers and the implementation of remote file access and execution.

COCANET recognizes references to remote pathnames via network special files *. The UNIX open system call has been modified in COCANET so that accesses to pathnames containing a network special file will fail locally, such requests are queued by the kernel for the network manager (a kernel-resident process). The network manager presents an interface similar to a device driver [Rowe 1981]. The following figure traces the opening of /B/x by a process on system A.

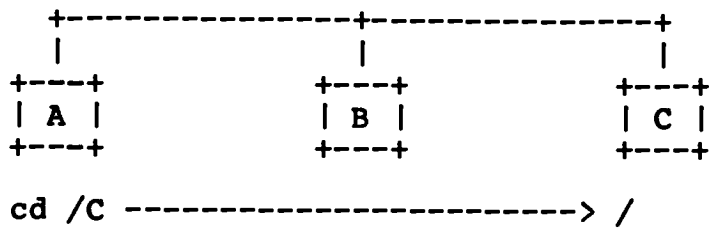
* - Actually a UNIX character special file.




Subsequent operations on the descriptor are passed to the private server where the operations are performed and results returned.

COCANET also supports remote working directories. In this case, extra logic is needed to detect remote file

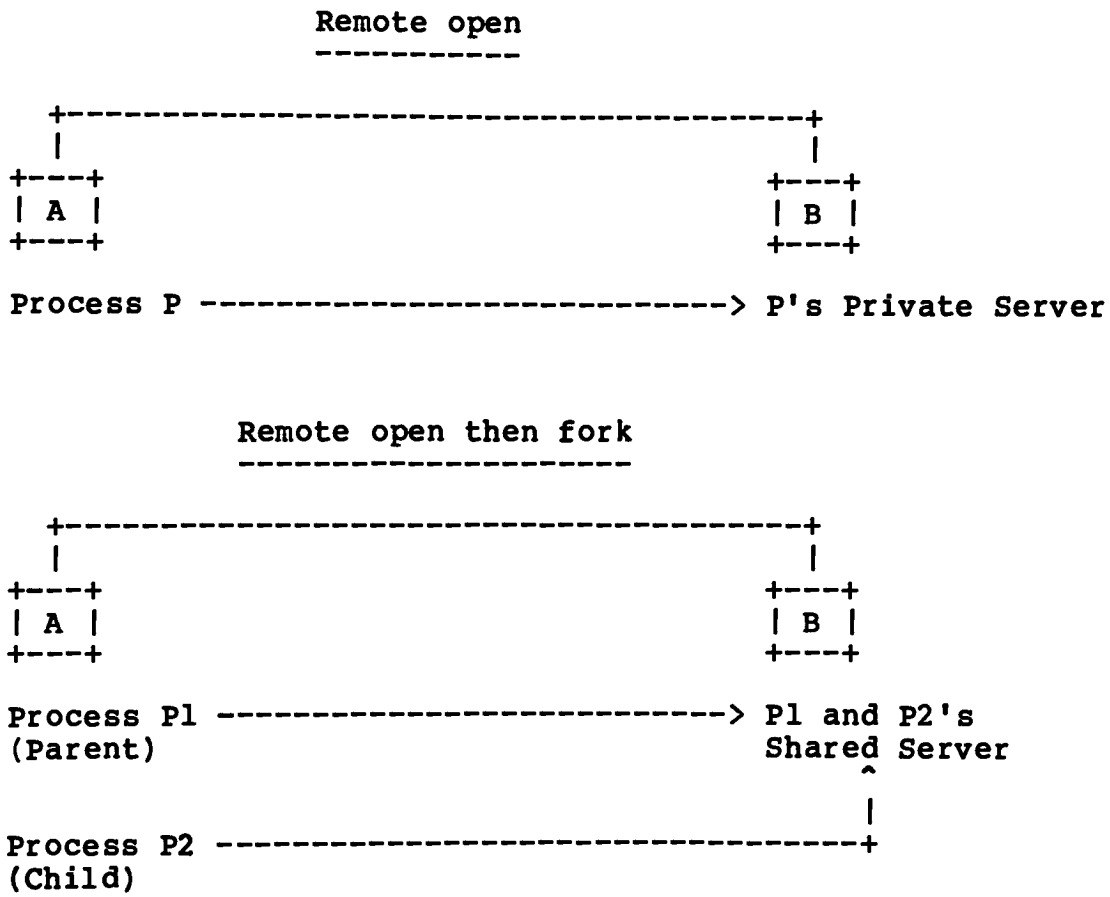
references, since pathnames that do not start with '/' are implicitly remote (given a remote working directory). To change to a remote working directory, a pathname containing a network special file must be used. The following figure shows a chdir from A to C:



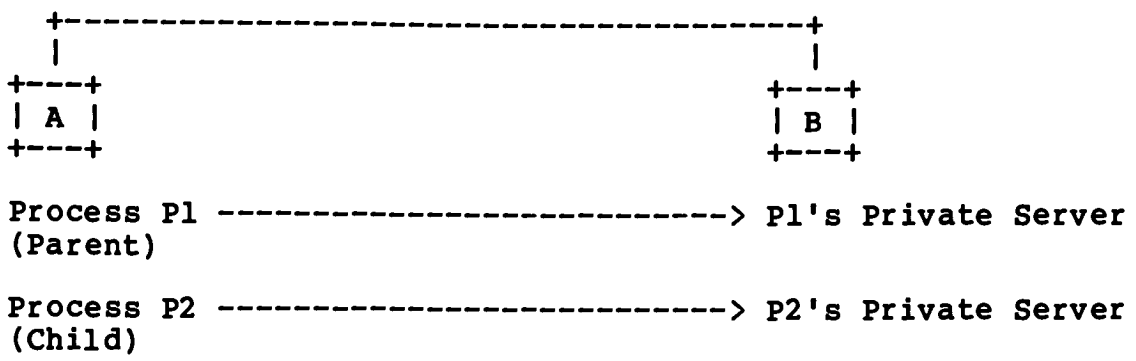

The chdir system call detects and handles this case in a fashion similar to the open system call. The private server on the remote system changes to the requested directory and returns the system call status. Assuming the chdir was successful, the process environment (in the user control block) is marked to indicate a remote working directory and the location of the associated server. Subsequent opens on files in the current (remote) directory are automatically dispatched to the server [Rowe 1981].

Forks from a user process with open remote files are handled in manner similar to the NC. If a user process has an open remote file and it forks, the private server becomes a shared server, handling requests for both parent and child. Should the child change to a remote directory, the shared server forks two private servers to maintain working directories: one for the parent and one for the child. However, if a user process with a remote working directory

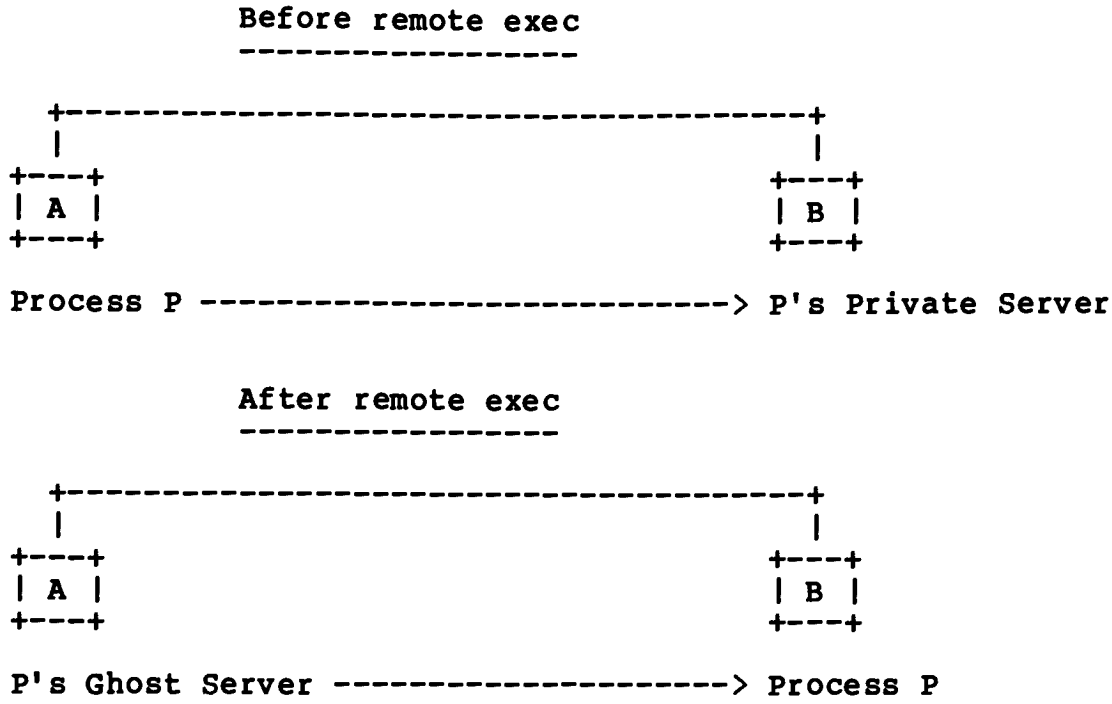
forks, then the private server forks another private server for the child [Rowe 1981]. This is necessary because the private server is bound to the parent's execution environment (i.e. working directory). Forking a new private server for the child allows the parent and child to have distinct execution environments (working directories) in the event that the child does a chdir.



Remote open, chdir then fork




COCANET also supports remote execution. The exec system call modifies the instruction and data segments of a process but otherwise does not alter the execution environment (i.e. open file descriptors, working directory, user-id, process group, etc.). COCANET views a process and its servers as a distributed entity. From this perspective, a remote exec shifts the point of control for a process from one system to another. For example, remote execution is indicated by specifying the path of a remote file to exec. First, the server inherits the execution environment for the process issuing the exec. Second, the server does the actual exec. Third, the process which issued the exec is designated a ghost server to manage the local environment for the exec'ed remote process [Rowe 1981].




Support of remote execution is not an immediate concern of a distributed file system. However, in the context of a distributed UNIX system, remote execution under COCANET maintains the semantics of the shell [Ritchie 1974] in a distributed environment.

User authentication under COCANET is similar to that employed by the NC. In COCANET, a user-id is authenticated when a server is first allocated. A request to system B from user X on system A is treated by system B as local user-id A/X. This allows individual administration of systems as in the NC. For security reasons, the super user is denied remote access capabilities. This ensures that security violations on one machine cannot cannot be "bridged" across the network. However, COCANET does not guard against

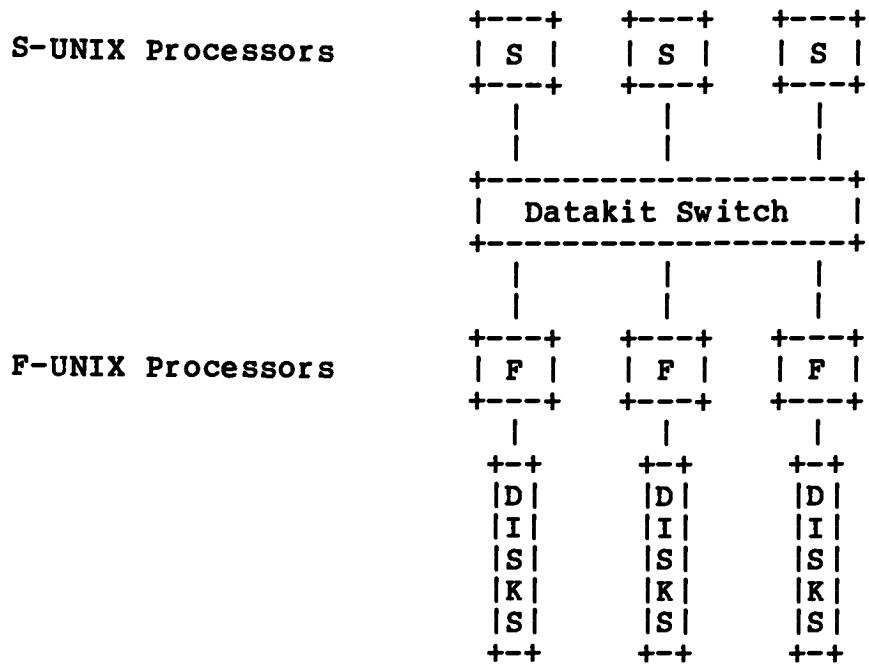
attempts to intrude on the network (i.e. bogus servers / requests, line tapping, etc.) [Rowe 1981].

## 3.3. S/F-UNIX

A system that uses the remote mount approach is the distributed UNIX system based on a virtual circuit switch, or S/F-UNIX, developed at Bell Labs in 1981 [Luderer 1981].

The S/F-UNIX architecture consists of processors (PDP 11/45's) that are partitioned (based on functionality) into two categories: F-UNIX Processors and S-UNIX Processors. F-UNIX processors provide file server services, whereas S-UNIX processors are "stripped" of file management functions. S-UNIX processors are computation servers that send requests to F-UNIX processors to access stored files. At the limit, S-UNIX processors could evolve to single-user, diskless workstations.

The F-UNIX and S-UNIX processors are connected to each other via the Datakit virtual circuit switch [Luderer 1981].




The Datakit switch provides virtual circuit functionality via packet switching technology [Fraser 1979]. In most distributed systems, a message or datagram service is preferable to virtual circuits because:

- Fixed bandwidth allocation (virtual circuit) is not appropriate for the "bursty" nature of data communications [Luderer 1981].

- Maintenance of circuit states introduces overhead in the network protocol [Luderer 1981].

However, for reliability reasons, virtual circuits are desirable for certain distributed applications (i.e. distributed file system) [Luderer 1981]. Therefore, at some protocol level virtual circuits can be provided even if the

underlying mechanisms are not virtual circuit based. The Datakit virtual circuit switch provides virtual circuits at a low protocol level that enables the attractive features of packet switching (dynamic bandwidth allocation) and a virtual circuit (reliability) [Fraser 1979].

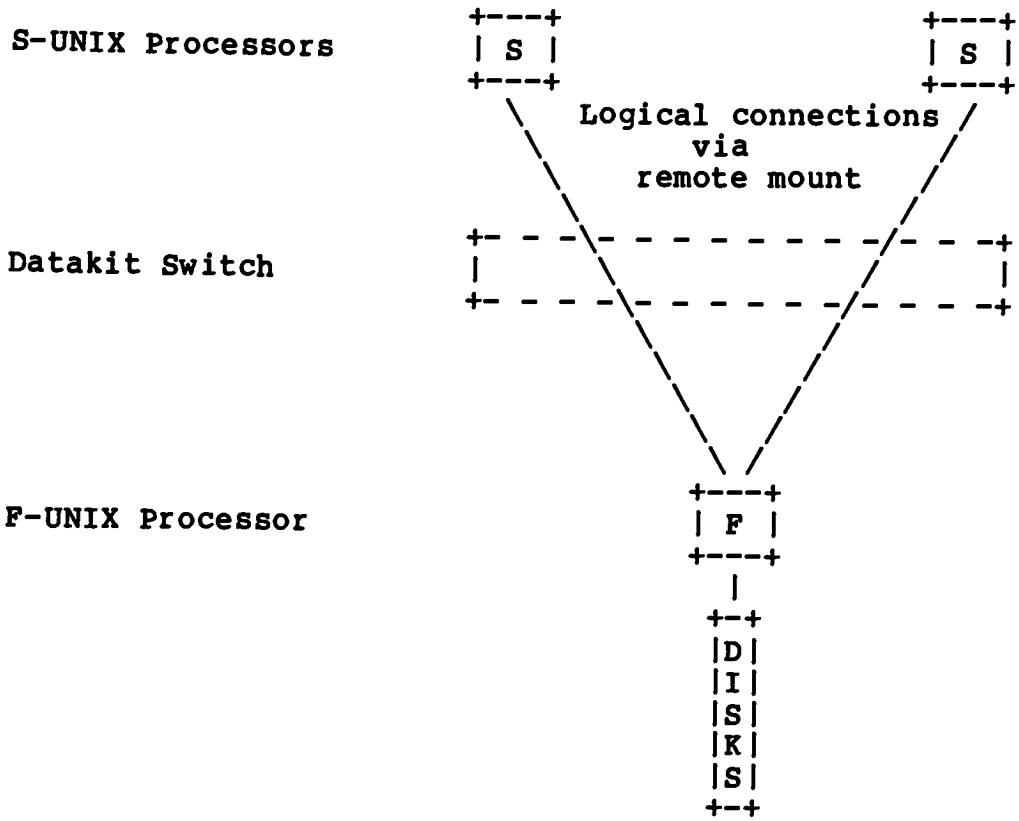
S-UNIX processors in an S/F-UNIX system can be diskless. However, in practice, local disks and file systems are maintained on S-UNIX processors for two reasons. First, for efficiency, it makes sense to store some files locally (i.e. temporary files). Second, without a local file system, the F-UNIX processors would have to represent the entire file system hierarchy for the network. By maintaining a private local file name space, the F-UNIX file systems can be remotely mounted onto the S-UNIX hierarchy [Luderer 1981]. In this way, the user's view of the network hierarchy can be personalized for each S-UNIX processor. Some restrictions apply with respect to file system access between S-UNIX and F-UNIX processors:

- An S-UNIX processor can only remotely mount an F-UNIX file system [Luderer 1981].

- An F-UNIX processor can only access its local file systems [Luderer 1981].

User processes only execute on S-UNIX processors. A user process accesses files on remotely mounted F-UNIX file systems in the same way as local file systems. Local and

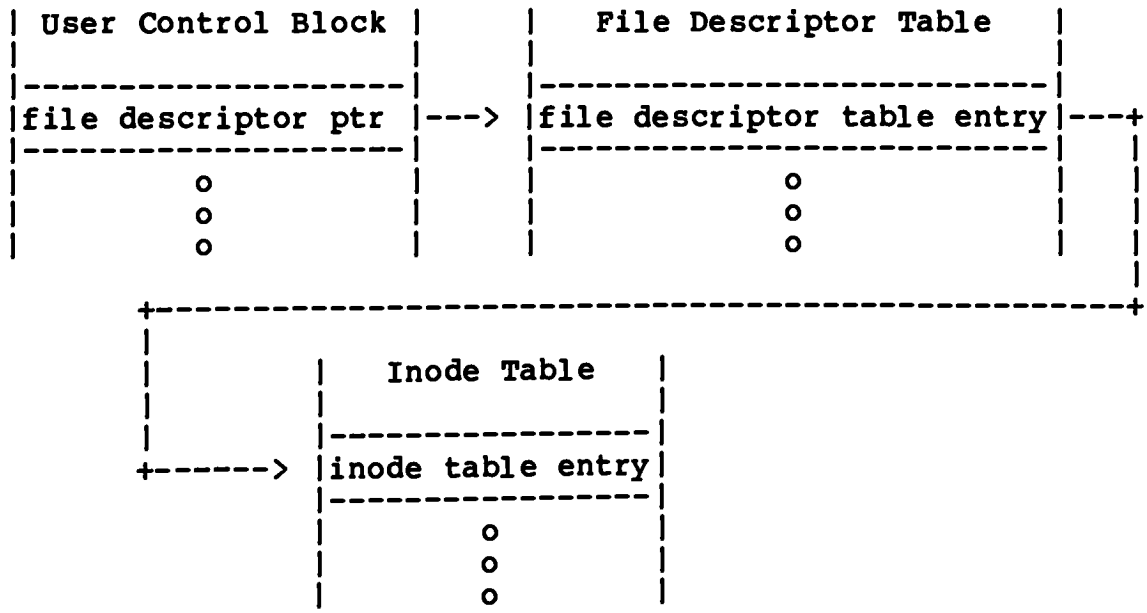
remote mounts may be simultaneously active on an S-UNIX processor, so, S/F-UNIX achieves file access transparency. Any F-UNIX file system can be shared (via remote mounts) by multiple S-UNIX processors; for example:




User authentication is identical to that provided by UNIX. An F-UNIX file system provides a globally accessible "passwd" file; as a result, user-ids are unique to the entire system. All modifications to support the S/F-UNIX distributed system are in the S/F-UNIX kernels. This makes the network as secure as the UNIX operating system on which the S-UNIX and F-UNIX kernels are based [Luderer 1981].

Within the kernels, a hierarchy of resources exists that corresponds to the file system layers within UNIX [Thompson 1978]. The resources at each of these layers are the file descriptor table, mount table, in-core inode table (cache), disk block buffer cache, and device request queue.

In UNIX, a file descriptor is the fundamental "handle" a user provides to system calls that access opened files. The file descriptor is an integer that indexes an array of file descriptor pointers in the UCB (user control block). These pointers select a file descriptor table entry, which in turn selects an inode. An inode is the central mechanism used in the file system manager [Thompson 1978]; see the figure below.




The S-UNIX kernel handles remote file access at the system call and inode level [Luderer 1981]. Since the inode

is the central kernel mechanism for file manipulation, no special provisions are necessary to handle system calls that deal with file descriptors instead of pathnames.

The kernel routine (namei) returns an inode in response to a pathname request. This routine was modified so that the S-UNIX kernel can detect remote access whenever a pathname crosses a remote mount point. A request is generated to the appropriate F-UNIX file server to acquire a "remote inode". A remote inode identifies the F-UNIX file server where the corresponding file (inode) resides, and a unique number (the remote inode number) assigned by the F-UNIX file server [Luderer 1981].

File specific information bound to an inode (access permissions, file length, disk addresses, etc.) is maintained by the file server on the F-UNIX processor [Luderer 1981]. In this way, the in-core inode tables (cache) of the S-UNIX kernels are consistent because they only refer indirectly to the actual (F-UNIX) inodes. The S-UNIX kernel exploits this treatment of remote inodes at the system call level to avoid the need to distribute pathname locks * across both S-UNIX and F-UNIX processors.

Not all S-UNIX file state information is distributed to the F-UNIX file servers. For example, F-UNIX file servers

* - In UNIX, whenever the namei() routine performs a pathname search, each directory (inode) read in is locked to prevent it from changing while the process is blocked on the disk read.

do not maintain a count of descriptors allocated to a file (number of opens on a file) [Luderer 1981]. Instead, F-UNIX file servers maintain a list of S-UNIX processors and associated files (inodes) that have been allocated at least one file descriptor. Hence, if an S-UNIX processor crashes, the F-UNIX file servers can close the corresponding files left opened by the crashed S-UNIX processor. In the event of an F-UNIX processor crash, connections from S-UNIX processors to the crashed F-UNIX processor are broken and associated remote mounts are removed [Luderer 1981].

To enable concurrent updates of remote files, the S-UNIX kernel does not cache disk blocks for remote files. Remote file disk blocks are cached in the F-UNIX kernel on the processor where they reside [Luderer 1981]. For increased performance, the disk block cache in the F-UNIX kernel is increased to include most of the available memory on the F-UNIX processor [Luderer 1981]. Since no user processes execute on the F-UNIX processor, the large memory allocation for the disk block cache has no diverse impact. However, if file replication between F-UNIX processors was added to the system, disk cache synchronization (i.e. cache invalidation) could be expensive.

## 3.4. 4.2 / 4.3 Berkeley Remote Mount

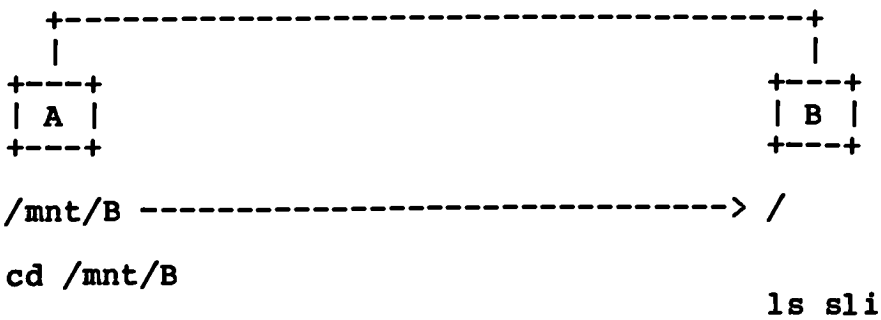
The 4.2 / 4.3 Berkeley UNIX remote mount facilities are currently under development at the University of California at Berkeley [Hunter 1985]. The Berkeley remote mount uses the same approach to access remote files as the S/F-UNIX system. Like S/F-UNIX, the Berkeley remote file access mechanisms are also implemented at the system call and inode level [Hunter 1985]. In terms of architecture, processors in the 4.2 / 4.3 UNIX systems are connected by an Ethernet or possibly serial lines. Modifications are at the kernel level to support remote mounts and a remote file server that currently executes at user level (for debugging ease) [Hunter 1985].

Local execution of remote programs is supported, but special mechanisms were needed to accomplish this [Hunter 1985]. Under S/F-UNIX (based on UNIX V7, a swapping system) remote execution was supported by modifying exec to issue a remote read after memory allocation for the program. However, 4.2 / 4.3 UNIX is a demand paging system based on a global page replacement policy. As a result, memory is not statically allocated at exec time for an entire user program.

Under 4.2 / 4.3 BSD UNIX, the virtual memory loader initializes page tables with the raw disk block numbers of the corresponding virtual address pages within the executable image file. On page faults, the page fault handler

"pages-in" the raw disk block for the virtual address that generated the fault and then finishes execution of the interrupted instruction. The problem is that no coupling exists between the raw disk blocks and the corresponding remote inode. Only logical blocks within a file system can be remotely accessed. A long term solution would add the capability to the virtual memory loader and page fault handler to recognize logical disk blocks for a remote file system and generate requests to the file server. Currently, local execution of a remote program is supported by copying the remote program to local swap space and then "paging-it-in" from there [Hunter 1985].

Handling symbolic links also presents problems in a distributed environment [Hunter 1985]. For example, assume the current working directory is on a (locally mounted) remote file system. If a symbolic link starting with '/' is referenced, there is no way to tell whether the local or remote root is implied.




Where slink is a symbolic link to /etc.

Which /etc is used? The one on A or B?

To ensure that a symbolic link cannot be used to access files and directories outside of the remotely mounted file system, the local root is always used.

File locking presents reliability concerns in a distributed environment. Lock information is kept in the in-core inode table, so, the server system must perform the actual file locking. If a client crashes with files locked, the server must detect this and unlock the files. Conversely, if a server crashes and reboots itself, the client must re-establish the locks that were lost. Currently, remote file locking is not supported.

Because the remote file server (RFS) executes at user level, the namei() routine must be simulated. Pathname searching is performed using stat() to check for symbolic links; each stat() call requires a call to namei() within the kernel. Namei() is one of the most time-intensive operations in the kernel, so it is to be expected that long remote pathnames impact the RFS performance heavily. Future versions of the RFS will execute as a kernel process.

## 3.5. SUN Network File System

The SUN Network File System (hereafter, NFS) uses a remote mount approach to implement remote file system access. The NFS is implemented in the kernel and uses the client/server model. The NFS introduces two abstractions, VNODE and VFS. A VNODE is a variant of an inode. While SUN's NFS maintains a UNIX operating system interface, the VNODE separates file system operations from the semantics of their implementations [Lyon 1985]. Thus, it is possible to support other operating system interfaces on top of the VNODE abstraction.

The layer under the VNODE is called a VFS (for virtual file system); the interface to a VFS is similar to a device driver. Two varieties of VFS can coexist; local and remote. Requests to a local VFS are serviced locally. A request to a remote VFS is transformed into a network request that is remotely serviced by a server. Because the VFS interface corresponds to a device driver, server requests are "stateless" or transaction oriented [Swinehart 1979]. Hence, crash detection and recovery mechanisms are simpler. For example, if a server or network crashes, a client simply retries the operation until the server or network is repaired. On the other hand, if a client crashes, no special processing is necessary on the server.

Server requests observe an RPC protocol. Sandwiched between the RPC and VFS is the SUN XDR (External Data

Representation) layer. The SUN XDR is used to convert between local machine representation and XDR (a machine independent representation). A server can selectively control remote access to local file systems via the exportfs command. The exportfs command designates file systems that can be exported to the network and which client machines have accessibility to the file systems. Once remotely mounted, the usual UNIX user authentication is used by the NFS.

Not all user level file operations are supported by the SUN NFS. For instance, file and record locking are not supported because servers are "stateless". SUN NFS does not support intermediate servers (internetwork routing) because it impacts performance, complicates access control, and permits recursive cycles in service requests. However, a client can remotely mount a server file system that is remotely mounted on another server. In this case, the remote mount request is "passed-on" to the actual server where the file system resides.

To minimize the impact of server crashes, NFS clients do not release write-delayed blocks until the server verifies that the data is written [Lyon 1985]. Even with reliable operation, it is possible that the file system caches between the server and client could become inconsistent because the server is implemented as a driver (below the file system cache) and is "state-less" [Hunter 1985].

## 3.6. LOCUS

LOCUS is a distributed operating system that was developed at the University of California at Los Angeles [Popek 1984]. Features LOCUS supports are network transparency, high reliability, availability, and good performance. The LOCUS architecture is built on PDP 11's connected by an Ethernet.

Individual machines in a LOCUS network cooperate to create the illusion of a single machine. Support for this illusion is implemented entirely in the kernel. In this way, LOCUS achieves a high degree of network transparency. LOCUS applications never need to reference a specific site or node in the network. Each site in the LOCUS network is a complete system and is capable of functioning alone or as part of the network [Popek 1984].

LOCUS supports replicated file systems to increase performance and reliability. Mechanisms in LOCUS check and resolve the consistency of replicated files as sites enter and leave the network.

LOCUS achieves location transparency through a global file name space. Two types of global names exist in LOCUS: user global names and system internal global names. User global names or high level names are similar to pathnames in a single processor UNIX environment. All user level global names are unique within the LOCUS hierarchical file system.

System internal global names are based on the concept of a file group *** composed of data blocks and file descriptor blocks **. A file descriptor block is identified by a file descriptor number. In this way, a system internal global name or low level name is the ordered pair (<file group>, <file descriptor #>). Since a file group can be replicated, the <file group> specified in the low level name is a logical file group. One logical file group can map many physical file groups. Each physical file group corresponds to an individual replicate of a logical file group. Information bound to a physical file group is device dependent and specifies the location of the group on the mass storage subsystem. File groups form the system wide naming structure via a system wide mount table.

On creation of a file, a file descriptor block is allocated. To avoid races (that may arise from multiple processors allocating file descriptor blocks) the file descriptor space in a file group is logically partitioned. Consequently, each storage site * (SS) for a file group allocates descriptor blocks from its own partition. File deletion requires synchronization among all file SSs prior to reusing file descriptor blocks.

File sychronization in LOCUS follows a centralized

*** - A file group is similar to a UNIX file system.
** - A file descriptor block is similar to an inode.
* - Processors (and associated disk subsystems) that store a particular file group are designated as the storage sites (SSs) for the file group.

protocol. A central synchronization site (CSS) is designated for each file group. All file opens, (except for pathname searching) generate a request to the CSS. It is not necessary that the CSS be the site at which the file resides. In this way, the actual SSs where the file resides can be selected by the CSS at open time. For replicated files, the CSS can choose the SS that provides optimum service.

After selecting an appropriate SS, the CSS notifies the requesting or using site (US) of the file descriptor allocated. Subsequent accesses to the file cause the US to directly request the SS for the file services. If the US, CSS, and SS are the same, then file operation corresponds to a local file reference that avoids network overhead. On the other hand, if the US, CSS and, SS are all different, then this is a worst case scenario with a higher degree of message processing overhead.

Typically, when a pathname is specified it is necessary to open for read all directory (components) of the pathname. In a development environment supporting many users, a high degree of UNIX kernel time is spent searching pathnames [Hunter 1985]. File creation and renaming places a temporary write lock on the directory containing the file in question. As a result, all opens that contain the locked directory (as a pathname component) are temporarily blocked. To counter this performance penalty, LOCUS supports a spe-

cial nolock read access type for reading directories. A nolock read access bypasses any synchronization on the file being read. However, nolock read accesses do guarantee to return a consistent state of a possibly dynamically changing file. Thus, nolock read semantics do not present problems to system software that accesses directories [Popek 1984].

LOCUS ensures replicated file consistency in the event of internetwork topology changes. As sites enter and leave the network, LOCUS invokes an algorithm to select a unique coordinating site. The coordinating site determines if any CSS has been lost, and if so, CSSs are initialized and file group recovery begun. If the network is partitioned, LOCUS permits updates on replicated files within each partition. A version vector reconciles differences in replicated files when the network partitions are merged. The basis of the mechanism is a version vector associated with each replicate. Each vector entry corresponds to a SS for the replicated file. Whenever a copy of a replicated file is updated, the corresponding entry in the version vector is incremented. Thus, two replicates are automatically reconciled * by comparing the version vectors of each replicate. If replicate A has a version vector where each entry is greater than or equal to the corresponding entries for

* - Not all inconsistencies are handled automatically. For instance, creation of two different files with the same name in different network partitions is classified as a directory name conflict and is manually reconciled [Popek 1984].

replicate B, then replicate A should be propagated. If this relation (i.e. A >= B or B >= A) does not exist, then replicates A and B are in conflict and manual recovery is necessary.

## 3.7. IBIS

IBIS is a distributed file system that was developed at Purdue University [Tichy 1983]. It is implemented entirely at user level and runs on VAXs supporting 4.2 BSD UNIX. IBIS uses a super-root approach with a syntactic convention that redefines a UNIX pathname to be:

[<hostname>:]<pathname>

In addition, there are restrictions concerning remote working directories that do not allow relative path changes to occur properly across machines.

To a user, the IBIS package is a library (libra.a). Consequently, a user adds remote file access capability to a program by re-linking with the IBIS library instead of the standard c library (libc.a) [Tichy 1983]. The IBIS remote access library redefines the system call layer and includes the standard I/O (stdio) package. The standard I/O package is not modified but included to ensure that system call references are resolved using the remote access definitions and not the standard ones in libc.a. The remainder of libra.a includes the "client-side" of IBIS and renames the local system calls to <syscall>_l so that references within IBIS to local system calls are explicit both syntactically in source modules and referentially in object modules.

The "client-side" contains the main logic which determines the destination (local or remote) of a user (client)

file access. Remote file access requests are packaged by the "client-side" into messages that are sent via 4.2 TCP facilities to a server that processes the requests on the target machine. The messages exchanged between client and server observe an RPC-like protocol.

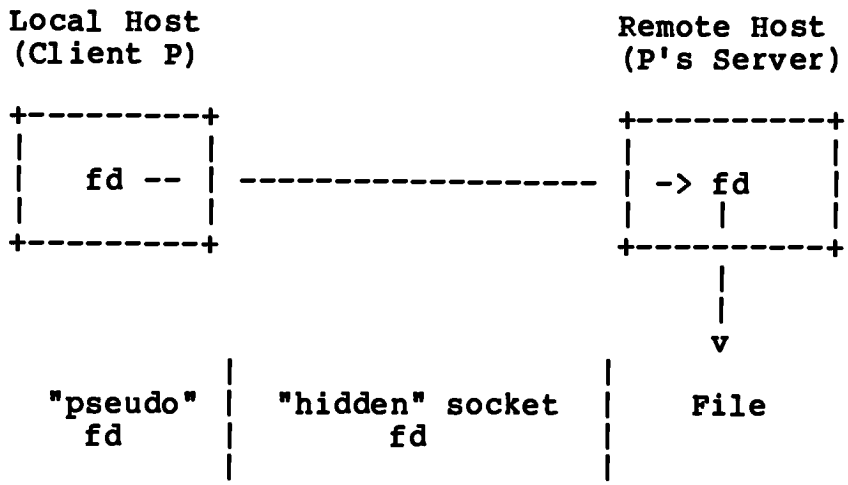
IBIS provides the sharing and inheritance of file descriptors across the process control primitives fork and execve. In fact, IBIS depends on and exploits this property of UNIX in order to provide it to the client.

A user gains access to a file by issuing an open file system call. Assuming the access is to a remote file, IBIS establishes a connection via the 4.2 TCP facilities to the server-creator on the remote machine. The server-creator then forks a "copy" of itself to handle requests from the client so that connections are established on a one to one basis (1 server process per remote host accessed by 1 client process).

Next, the client issues a local open to allocate a descriptor that will be "mapped" to the remote file. The client then sends a remote file open request passing the remote file pathname, local "mapping" descriptor and other open specific arguments to the server over the new connection.

The server receives the request and opens the file locally. If the local file descriptor (fd) returned doesn't

match the mapping fd then the local fd is locally "dup2'ed" over the mapping fd, closed, and then reset to the mapping fd. Finally, the server returns status of the open to the client. This type of fd allocation on opens ensures a mirror image of the client's fds on the server.




Upon receiving the status (of the open) from the server (assuming it is OK) the client fd is marked as a "pseudo" fd with the descriptor for the socket that connects the client to the server. The socket descriptor for the connection is also marked as "hidden". IBIS marks descriptors (via fcntl(2)) in an unused (by 4.2 UNIX) part of the flag field in the descriptor entry. Marking the connection descriptor as "hidden" is done both to provide a degree of protection (a self checking mechanism), and to quickly isolate connections (socket descriptors) during a fork operation.

The remote open is complete and the mapping fd is returned to the user who treats it as a normal fd in

subsequent operations. For example, a read issued is easily classified as local or remote by examining the associated flag field. If remote, the connection descriptor is extracted from the flag field. With this information, a request is sent to the server (via the connection descriptor) that includes the request type (read in this case), user's fd, and number of bytes to read.

Except for error handling, the above sequences describe the operations performed by IBIS to handle remote file requests.

In the event of an exec system call, all open descriptors are inherited (unless a user specifys the close-on-exec flag). Thus, descriptors used to map remote file accesses remain intact and the behavior of execve is preserved. In the case of a fork, more overhead is required to establish a new "thread of control" with respect to remote files. Specifically, the client sends each server process (one per remote host accessed by the client) a remote fork request, along with a new port number. On reception of the request, the server forks itself and allocates a new socket. The child server binds the socket to the new port number and connects to it.

The parent server sends the client the port number of the remote host connection. The client receives the port number of the server and saves it for verification purposes. Next, the client forks, and for every server connection

(descriptor marked as "hidden"), the child client accepts connections on the new port number that was sent to the server. Upon accepting a connection the child client verifys that the peer port number equals the port number that was received earlier from the remote host and closes the original "old" descriptor. Finally, both parent and child return (parent with child's pid and child with 0).

IBIS relies on the standard UNIX file access protection and control mechanisms; no separate authentication server is provided. Thus, files accessible to user X by "rlogging" into remote host A as user X are also accessible via IBIS. One problem that arises is the authentication of request messages from client to server. An extreme approach would require privileged (i.e. super user restricted) connections between all clients and servers. This would require all client and server programs to be privileged, negating the ability of ordinary applications to make use of IBIS. We can avoid this extreme approach by noting that the connections themselves need not be privileged, they just need to be validated by a trusted authority.

IBIS solves this problem by exchanging authentication information between a privileged connection starter and a server-creator on the remote system. These processes use privileged channels to set up and validate non-privileged connections between the clients and servers. This exchange of critical information establishing the dedicated client /

server connection, is called a "two way handshake via secure channel" [Tichy 1983]. Once the dedicated connection is established, it is only accessible to children of the client / server and the super user.

## 4. IBIS Enhancements

The goals of my thesis project were:

(1) Convert IBIS to a super-root approach that achieves file access transparency semantically without redefining the UNIX pathname.

(2) Enable IBIS to work on a heterogenous network of 4.2 UNIX systems (VAX, Power 6/32).

(3) Enable a fully generalized current working directory, so that users can transparently change directory from one machine to another both explicitly and relatively.

(4) Enable IBIS to work on a heterogenous network of UNIX variants, specifically 4.2 (VAX, Power 6/32) and system III (Masscomp).

In order to demonstrate the above objectives are met and to facilitate development/testing, a package was developed to exercise and test the remote and local access capabilities of IBIS. In addition, some commands and library routines that are tightly coupled to UNIX were modified. All modifications are described in appropriate sections of this thesis.

## 4.1. Functional Specification

IBIS is purposely designed to redefine the system call layer in a transparent manner. Hence, user inputs and outputs are as described in section 2 of the 4.2 bsd release of the UNIX programmers' manual.

## 4.2. UNIX 4.2 BSD Dependencies

In theory, user level DFS's like IBIS should be more portable and maintainable than kernel based systems. However, it turns out that the IBIS design is both 4.2 system call dependent and 4.2 kernel dependent. These dependencies are a major obstacle that must be overcome for IBIS to operate in a heterogenous network of UNIX variants. Specifically, the following table gives 4.2 system calls without direct equivalents in system III or system V.

fchmod, truncate, ftruncate,
fsync, dup2, mkdir, rmdir,
rename, lstat, readlink, symlink,
getpeername, wait3

Other system calls that have different behavior in 4.2 BSD and Masscomp's system III include:

open, creat, fcntl, flock,
lock, stat, fstat

In the case of fcntl, the problem is not in the explicit interface. Rather, IBIS makes assumptions about the sizes of some unused portions of kernel data structures. These (unwarranted) assumptions do not hold in the case of the Masscomp.

Finally, Masscomp's variant of system III supports an older version of Berkeley's socket extension. Some facilities IBIS depends upon are missing, and need to be simulated with special purpose routines.

These differences illustrate the problem of transparently supporting non 4.2 variants of UNIX via IBIS. Such support represents an extension to a heterogenous network: not only are the architectures of the machines different (VAX, Power 6/32 and 68000), but the services provided by the respective OS's (4.2 and system III) vary as well. IBIS is designed to provide a mapping mechanism that translates local file system requests into identical remote file system requests; given the system variations, such translations may be impossible.

## 4.3. Modifying IBIS

Initial modifications allowed IBIS to run as an ordinary (non-privileged) user. Defining the compile time flag "TSTUSR" disables the use of privileged functions (i.e. setuid). In addition, "TSTUSR" forces the use of a non-privileged TCP channel for the two-way handshake.

Originally, the local system call layer within IBIS consisted of VAX assembly language modules. To facilitate ports to other systems, these files were converted to use C preprocessor macros and the compile time flags "VAX", "P632", "MC2A" and "MC42" were introduced for the systems under consideration. More information on compile time flags and generating IBIS is in the appendices.

Depending on which flag is set, the appropriate local system call definitions are generated*. In addition, the client and server use these flags to compile system dependent source code. This allows easier maintenance of the package, because a single set of source files forms the base for all machines. However, there are some instances where the changes implied by the multiple compile time switchs are so pervasive that separate source may improve code readability.

As part of the IBIS enhancements, a super-root approach

* - Except for the Masscomp, because it does not use a system 4.2 interface and does not pass args via the stack but via different registers based on system call.

that designates remote file systems semantically, was adopted. The IsLocal() routine within IBIS detected remote file requests, originally by looking for a ':' in the first path component. IsLocal() was modified to detect any pathname component that corresponds to a "network special file" **. The server was modified to expect absolute path names. The functionality of "network special files" was tested and verified before adding support for generalized remote working directories.

To add generalized remote working directories The IsLocal() function was broken into several subfunctions: IsLocal(), oklocomp(), goloc(), nxtwdidx(), minbynam(), allwdidx(), setwdidx() and getwdidx(). IBIS allocated a descriptor for the connection to the server (remote machine) on which the current working directory resided. An additional descriptor was allocated to indicate the "working directory index level" (wdidx). A wdidx of -1 indicates a local working directory. A wdidx of 0 or more indicates a remote working directory and the level of the directory within the remote file system hierarchy. For example, 0 indicates the remote root (i.e. /), 1 indicates one level below the remote root (i.e. /bin) and so on.

Three routines manage the wdidx: allwdidx(), setwdidx()

** - A "network special file" is actually a character special file that uses a major device type not allocated by UNIX. Minor devices denote individual hosts in a network.

and getwdidx(). The actual machine the remote directory resides on is indicated by "the current working directory file descriptor" (cwdfd). The cwdfd is managed by the routines: initcwdfd(), setcwdfd() and getcwdfd(). If the cwdfd is -1 then the current working directory resides on the local machine, otherwise cwdfd is the value of the socket descriptor for the connection to the server (where) the working directory is located. Descriptors are used to store working directory state information that must be preserved across fork and execve. This is necessary because a user process cannot directly store state information in its control block.

The actual remote working directory path is not stored by the client. Instead, it is maintained by the server where the current working directory is located. To determine the remote directory of a server (cwdfd != -1 and wdidx >= 0), the client issues a special request to the server.

For IBIS to work correctly between different 4.2 UNIX machines (i.e. VAX and P6/32) changes were applied to all messages sent between client and server to compensate for byte ordering differences. The 4.2 network support facilities included library routines to convert between network and host byte ordering. Basically, modifications were made to message exchanges so that all non-character data is converted from host to network order on transmission and vice-versa on reception. The areas changed included the connec-

tion starter, the client / server RPC, and several system calls that dealt with non-character data. Only non-character data (i.e. integers) required conversion; character data is transmitted correctly over the network.

The last group of modifications compensate for functional differences in services provided by 4.2 and system III. The degree of compensation possible is proportional to the degree of system call transparency achievable.

Generic functions (read, write, open, creat, close, lseek) may not behave identically. Equivalents, by definition, must be functionally identical across all UNIX variants. For example, specification of contiguous file creation on Masscomp's system III requires specification of a maximum file size; no such option exits in 4.2. Therefore, a 4.2 server cannot exactly duplicate the effects of contiguous file creation. In general though, most generic functions can be mapped to a remote form that achieves the same result. This is because the functions performed by generic operations can be cleanly separated from the data being operated on (raw user data in the cases of read and write) [Israel 1979].

However, functions that tightly couple function and data (i.e. stat, lstat) cannot always be mapped between UNIX variants. In the context of IBIS, this presented a problem. The intent of IBIS is to provide a consistent environment that maps the file systems at each network node into a

single file system accessible via the standard mechanisms from anywhere in the network. To do this in the UNIX environment requires that system programs such as csh, ls, mkdir, pwd, editors, language processors etc. can operate with remote access capability. This is turn implys that non-generic functions (i.e. stat) must be supported in a UNIX variant network (in this case 4.2 and system III) *.

This could be accomplished by several alternatives:

(1) Extend the RPC message protocol of IBIS to incorporate the client/server UNIX type on requests/responses. Consequently, both client and server must be able to convert between all supported variant UNIX types and provide reasonable conventions for facilities not supported and error responses between variants.

(2) Create a virtual machine that would execute on all nodes. The virtual machine simultaneously emulates a set of UNIX variants (4.2, system III, system V).

(3) Support a small set of variants (4.2 and system III) by providing conversions where possible the two. Use the "super-set" for network transmission since client and server have no knowledge of each others' variant type.

Alternative (2) has attractive properties, but is

---

* - The satellite processor system maps all system calls to a central node to resolve conflicts in a heterogenous network [Lycklama 1978].

beyond the scope of IBIS. Alternative (1) imposes overhead in the form of conversions on each message exchange that may impact performance. For these reasons, the third alternative was chosen. For unsupported 4.2 system calls a system III server returns EUAVAIL (81). The following table describes the mapping from 4.2 to system III:

| Client 4.2 | Server system III |
|---|---|
| mkdir | simulate via /bin/mkdir |
| rmdir | simulate via /bin/rmdir |
| rename | simulate via /bin/mv |
| open | open, but contiguous files are opened as regular files |
| creat | creat, but contiguous files are created as regular files |
| readlink | not supported, no symbolic links |
| lstat | " " " " " |
| symlink | " " " " " |
| truncate | not supported by system III |
| ftruncate | " " " " " |
| fchmod | " " " " " |
| flock | not supported because lockf (sys III) semantics differ |
| stat, fstat | stat, fstat. convert system III stat buffer to 4.2 form and supply 0's for undefined portions. |
| dup2 | use close, fcntl to simulate |

Another incompatibility between 4.2 and system III is the structure of directory entries. The readdir library routine under 4.2 is the preferred interface for accessing directory entries. While readdir is not a system call it is redefined within IBIS and treated as a pseudo system call. This is necessary because 4.2 directory entries contain

non-character data. Hence, a readdir request from a 4.2 P6/32 client to a 4.2 VAX server returns the correct result by transparently converting non-character data to and from network byte order. To enable readdir to work between 4.2 and system III required further massaging. Network transmission is in the "super set" 4.2 directory entry representation. Consequently, a system III server executing a remote readdir converts the system III directory entries to an equivalent 4.2 form before conversion of non-character data to network order. Conversely, readdir for a system III client converts the 4.2 directory entry representation received to system III representation. Unfortunately, readdir is not a standard feature of system III. Therefore, to allow system III programs that read directory entries to work transparently with IBIS, user level modifications are needed (convert to use readdir) and a system III readdir needs to be added to libc.a, which IBIS redefines for remote access capability.

The differences and corresponding compensation outlined so far between 4.2 and system III affects user system call transparency. However, within IBIS itself there were several 4.2 dependencies (both system and kernel) that had to be overcome before IBIS could be ported to system III. These dependencies are:

4.2 system call(s)
---
IPC TCP facilities
wait3
fcntl F_SETFL and F_GETFL functions

4.2 kernel
---
The file descriptor flag field must be a 32 bit int of which the high 19 bits are not used by the file system manager in 4.2.

The Masscomp system III implementation does support TCP and a "4.2 like" interface based on the concept of a socket [MASSCOMP 1985]. However, there are differences which had impact on IBIS:

(1) The bind and listen system calls do not exist on the Masscomp TCP implementation. Socket addresses are bound at socket creation time via the socket call. No backlog can be specified for connection requests. Additionally, a socket "acceptor" is specified at socket creation time to distinguish it from a "connector".

(2) The connect and accept system calls do not specify the length of the peer's address. The accept call does not return the descriptor allocated for the accepted connection. Instead, the descriptor specifying the "listening" address is used for the exchange of data.

(3) Under TCP, recv and send are not supported, instead read and write must be used.

(4) Reception and transmission of out-of-band (OOB) data is done via ioctl calls and not through recv and send.

One would think that a user level implementation should not be dependent on a particular kernel; however IBIS is. Specifically, the method used to mark a descriptor as being allocated to a remote file employs the fcntl system call get and set descriptor flag sub-functions. The VAX and P6/32 4.2 implementations allocate an int (32 bits) to the descriptor flag within the kernel. IBIS restricts itself to the upper 19 bits of this field (unused by 4.2 file system manager). The Masscomp System III implementation provides and identical fcntl system call specification. However, only a byte is allocated to the descriptor flag field.

One could argue, "how can the system III fcntl specification be identical to 4.2?". IBIS uses the fcntl get/set capabilities in a non standard, not-advertised-as-supported manner [UNIX 42]. Technically, a user is supposed to use (and the OS is supposed to allow) only the advertised flag values to be set. The VAX and P6/32 implementations of 4.2 are very forgiving, and allow IBIS to "get away" with the use of unsupported descriptor flags. Unfortunately, the Masscomp system III fcntl system call was not as forgiving. This tight coupling of IBIS to the 4.2 kernel created a major obstacle and dilemma in porting IBIS to the Masscomp. Several alternatives for descriptor marking were developed:

(1) Remove the 4.2 kernel dependency (use of descriptor flag fields) by developing a user level mechanism that achieves the same result. This was not tenable since the entire IBIS design was based on using the descriptor flag field as the mechanism for mapping remote file system requests. The easiest modifications use a private flag table (in user memory) indexed by file descriptor. IBIS references to the descriptor flag field would be replaced by references to this user level representation. The exec and fork calls, as well as the /lib/crt0.o (runtime startup routine), have to be modified to pass this flag table via a pipe. In this way, sharing and inheritance of descriptors is emulated by user level code. Unfortunately, this requires two descriptors (for the pipe), and alters the exec semantics (since an exec is now actually a fork/exec).

(2) Develop less drastic alternatives that avoided the massive modifications and redesign above. The following are two such alternatives:

2a Use the descriptor offset field instead of the descriptor flag field to hold all mapping information. The lseek system call can simulate a get/set function by using an appropriate value for the "whence" argument to lseek. Bit 31 cannot be used because the offset must be positive. Bits 30

and 29 of the offset field could be allocated to the pseudo and hidden flags needed by IBIS. These bit positions (30 and 29) are used so that normal file accesses aren't misinterpreted as remote file accesses. For example if bits 0 and 1 were used, then any access to an odd or divisible by 2 offset would be interpreted as a remote access. Allocating use of bits 30 and 29 to IBIS flags prevents this from happening but also limits file sizes to a maximum of 2**28 + 1 bytes. If the pseudo flag bit is set then the remaining value of the offset field (bits 0-28) does not represent an offset value but is interpreted as the socket descriptor for the connection to the remote machine server.

2b On inspection of system include files for the Masscomp, bits 6 and 5 of the descriptor flag field appeared to be unused. A test program verified the availability of these flag fields and a degree of forgiveness on the Masscomp. Hence, using these two bits to designate the pseudo and hidden flags eliminates the file size constraint in 2a above. If the pseudo flag is specified then the descriptor offset field is interpreted as in 2a above.

Approach 2b was adopted. This approach gets around the
4.2 coupling between IBIS and descriptor flags without

rewriting the entire package. 2b was implemented by redefining the local version of fcntl (fcntl_l) to a special Masscomp version called MC_fcntl. This routine transparently handled the F_SETFL and F_GETFL incompatibilities via the procedure described in 2b above. The remaining fcntl functions are handled in the normal manner.

## 4.4. Internal Interfaces

The internal interfaces of IBIS are partitioned into three areas:

- The IBIS client.
- The IBIS RPC protocol.
- The IBIS server.

The IBIS client side redefines system calls. System calls that use local resources are "passed-on" to the local kernel. System calls that use remote resources are packaged (via an RPC protocol) into a request to a server on the host where the remote resources reside. The server services the request and packages the results into a response that is returned to the client. The client translates the response back into a local representation and returns it to the user.

The IBIS RPC protocol is not technically a module to module interface (although conceptually it is). The IBIS RPC protocol is based on message exchange between the client and server and is detailed under the communication among modules section.

The remote resources supported by IBIS are files. Files are designated through either a pathname or file descriptor. Therefore IBIS must determine from a pathname or descriptor (based on system call) if an access is local or remote. Given this constraint, all system calls

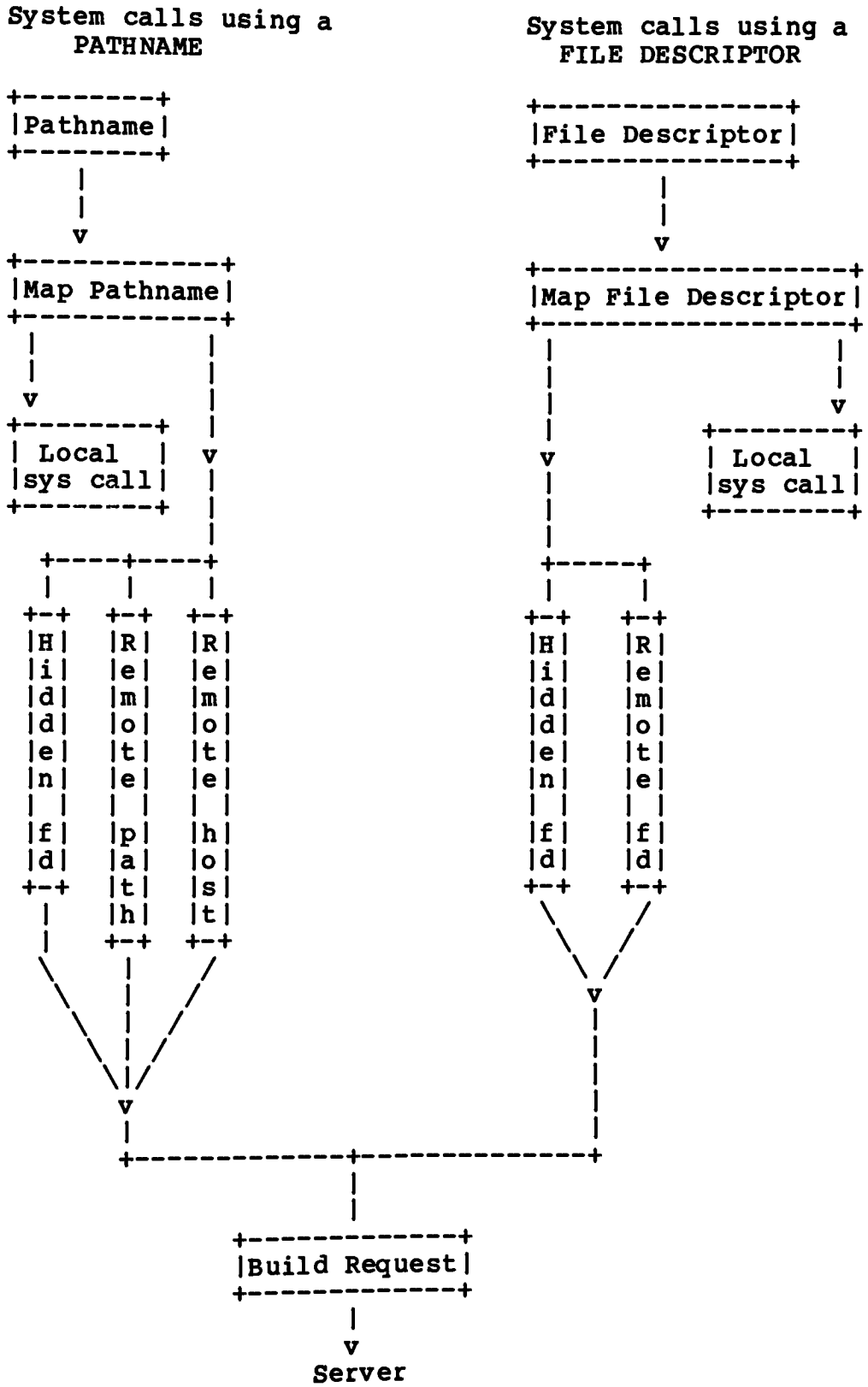
redefined by IBIS can be classified as determining local or remote access based on:

- Pathname and/or current directory.
- File descriptor.

If a path is anchored at the root directory, then a network special file must be a component of the pathname for remote access to occur. If the current working directory is remote, then the pathname must select a component within the remote file system hierarchy to remain a remote access.

Once a file is opened, it is known whether or not the file is local or remote. IBIS tags the file descriptor to indicate a remote file descriptor. If no tag exists, then the file descriptor is for a local file. To maintain the semantics of process control system calls (i.e. fork, execve) when remote files are opened, IBIS distributes the process control operation to include the server.

These classifications for file access can be shown diagramatically as follows:


System calls using a PATHNAME
System calls using a FILE DESCRIPTOR
Pathname
File Descriptor
Map Pathname
Map File Descriptor
Local sys call
Local sys call
Hidden
Remote
Remote
Hidden
Remote
Remote
path
host
Build Request
Server

## 4.5. Module Designs

Given a pathname, the logic for determining remote access is found in the function IsLocal():

Inputs.

(1) Pathname.

(2) Current working directory is implicitly established within this function.

Outputs.

(1) Remote Host Name

(2) Pathname on Remote Host.

(3) Descriptor of Remote current working directory or -1 if the current working directory is local.

Returns.

(1) 0 = Remote Pathname.

(2) 1 = Local Pathname.

Description.

(1) If the cwdfd and wdidx are not initialized, then initialize both of them. The cwdfd must be initialized first because if the cwdfd indicates a remote file then it will be used to generate a request to the server to acquire the remote working directory pathname. Next,

using this pathname, the wdidx is initialized.

(2) If the cwdfd indicates a remote working directory, then find the associated host name using the routine hostnamebyfd(). Hostnamebyfd() takes one input parameter, verifies the socket descriptor is "hidden," then issues a getpeername() system call to get the internet address of the peer. Using this internet address, hostnamebyfd() issues a gethostbyaddr() library call to get the host entry structure. A pointer to the host name is returned.

(3) If the pathname starts with '/' or the wdidx is -1 (local), then invoke oklocomp(). oklocomp() scans the pathname and breaks it into a local path, a remote path, and a minor device number for a host (if a network special file is encountered).

(4) If the wdidx is not -1 (local) then goloc() is invoked. goloc() determines whether the path accesses a remote file or local file. If a local file is accessed, then goloc() returns the local path, otherwise goloc() returns a null pointer and the path passed to goloc is the remote path.

Caveats.

(1) Special provisions are made within IsLocal() to handle the chdir() system call and the resulting recursive call from the gethostbyaddr() library routine in step

(2) above. If a user does a chdir() to a remote machine through a local path, then the working directory of the client must be changed to the local portion of the remote path. This is necessary so that if a relative (i.e. "../") path is used to "bridge" back to the local machine, the client ends up in the correct local directory. To support this, IsLocal(), oklocomp() and goloc() place the local bridge path into a private buffer. The chdir() system call accesses this bridge path buffer through a client routine and (locally) chdir()'s to it after issuing the chdir() service request to the server.

Because of the use of this private bridge path buffer, IsLocal()/IBIS is not re-entrant. This is not a problem under UNIX because only one system call can be outstanding at a time for a user. However, because certain library routines (in particular gethostbyaddr()) can generate an open() system call, an endless recursive call "loop" can be made to IsLocal() that exhausts user stack space. Another sideeffect is that the private bridge buffer is invalidated. Since it is already known that gethostbyaddr() opens /etc/hosts (a local file) and because it was not desirable to modify (or redefine) the network library routines, The invocation of hostnamebyfd() was surrounded by a semaphore to prevent an endless recursive call to IsLocal(), and corruption of the private bridge buffer.

Given a file descriptor, the logic to determine whether or not it corresponds to a remote file descriptor is in the routine getmap():

Inputs.

(1) User file descriptor.

Outputs.

(1) NONE

Returns.

(1) Hidden descriptor for connection to server that the corresponding file resides on.

(2) -1 if the file descriptor is not mapped to a remote host.

Description

(1) The file descriptor flag field contains the value of the hidden descriptor (for server connection) and a flag to indicate that the users' descriptor is a pseudo descriptor. Subsequent requests to the server occur over the socket designated by the hidden descriptor. Each request packages the user's fd as an argument to the server (since the server provides a mirror image of the users' fd's).

Given a user fd that maps a remote file, it is not

necessary to establish a connection to the server, since the fd itself is used to acquire the connection descriptor. However, system calls that deal with pathnames must map the host name returned from IsLocal() to a server connection. The routines getHostConn(), ConnOn(), GetHst(), Hidden(), and getpeername() (4.2 system call) accomplish this function. Recall that IsLocal() sets the output descriptor to the remote server, if the working directory is remote. In this is case, there is no need to determine the socket descriptor using the host name provided by IsLocal(). However, if the current working directory is local but the pathname corresponds to a remote file, then the host name returned by IsLocal() must be translated into a server connection (establishing the connection if necessary).

GetHostConn() converts a host name into an internet address, and invokes ConnOn() with this address as an argument. ConnOn() scans all descriptors, looking for hidden descriptors indicating a server connection. For each one found, GetHst() is invoked with the descriptor as an argument. GetHst() invokes getpeername() and returns the peer internet address. ConnOn() compares the address returned by GetHst() to the one requested, and returns the corresponding descriptor to getHostConn() on a match. If no match is found, ConnOn() returns -1 and getHostConn() invokes EstabConn() to establish the connection to the server, returning the resulting descriptor.

On the Masscomp system III TCP implementation, the getpeername() system call is not supported. This problem was circumvented by allocating a pseudo system call that sends a request to each connected server requesting its internet address.

## 4.6. IBIS Client to Server Communication

The interprocessor communication between client and server mirrors the data flow between the user and the kernel (i.e. system call). This correspondence is illustrated below.

In the file client.c, the function SendCommand() builds the request sent to the server. SendCommand() supports the following combinations of system call parameters to be sent to the server.

2 path names and up to 2 integers.

1 path name and up to 3 integers.

0 path names and up to 4 integers.

Assuming a connection has been established between client and server, the server (rfiled.c) is blocked in recv() waiting for a request from the client. In essence the server is waiting for RPC service requests.

The RPC protocol specifies the system call to be performed by the server and any associated arguments. The server will generate a response, based on its ability to satisfy the request.

Each request and response consist of a variable number of messages. Two types of messages exist:

- A header message (hdr msg). A hdr msg is a fixed size message containing a service request code, result code, and size code.

- A data message (data msg). A data msg is a variable size message, containing, for example, the data to be written to a remote file. Each data msg is preceded by a hdr message giving the type and size of the data msg.

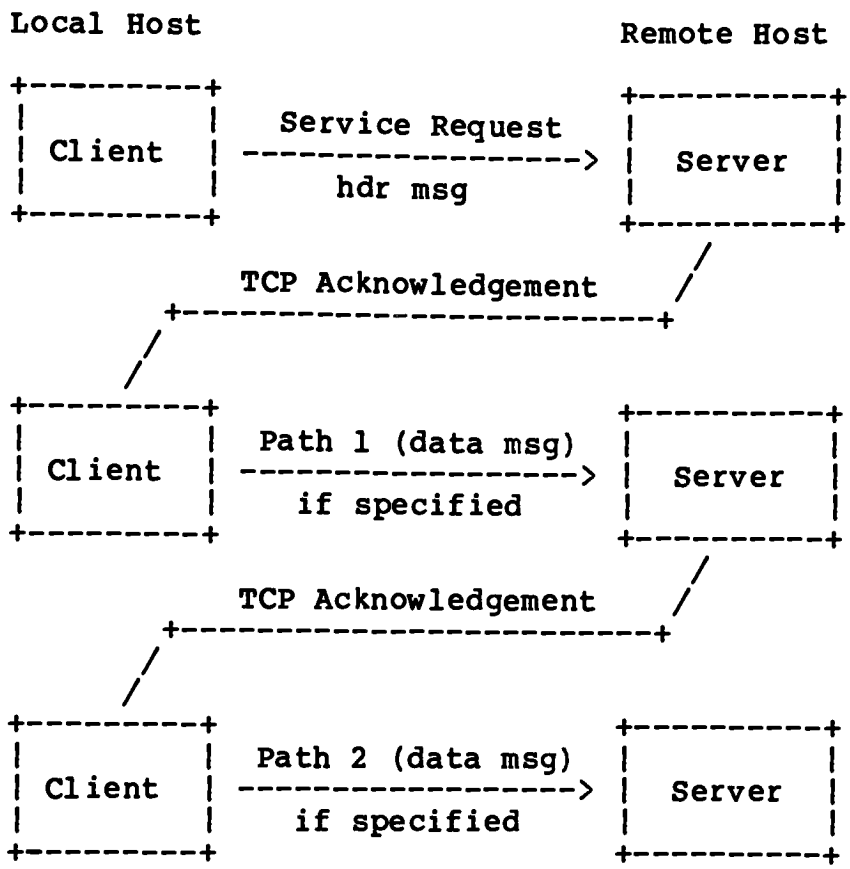
The request that determines the type of system call to be invoked is called a service request. The client uses SendCommand() to transmit a service request, which is picked up by the server in RecvCommand(). The program interfaces are as follows:

```
Client
------
SendCommand(fdHid, scode, npaths, p1, p2, a1, a2, a3, a4)

int     fdHid, scode, npaths, a1, a2, a3, a4;
char    *p1, *p2;

fdHid:  connection descriptor for the remote host
scode:  system call code (see ra.h)
npaths: # of paths (0,1 or 2)
p1:     1st path, iff npaths==1, else NULL
p2:     2nd path, iff npaths==2, else NULL
a1:     0 iff npaths>0, else arg
a2:     0 iff npaths>1, else arg
a3:     arg
a4:     arg

returns 0 if OK else -1.
```

```
Server
------
RecvCommand(cmdbuf, p1, p2)

struct  command *cmdbuf;
char    *p1, *p2;

cmdbuf: holds hdr msg received
p1:     holds 1st path, if one received
p2:     holds 2nd path, if one received

/* declaration for struct command is in ra.h */

returns 0 if OK else -1.
```

SendCommand()/RecvCommand() was modified to encode/decode non-character data (the hdr msg) into/from network order from/to native machine order.

Diagramatically, a service request proceeds as follows:




Note - the transmission of paths 1 and 2 is optional.

Depending on the system call, the service request may provide sufficient information to allow the server to execute the requested system call on the remote host. For example, if the requested system call is lseek(), the header message contains all the necessary arguments, in particular:

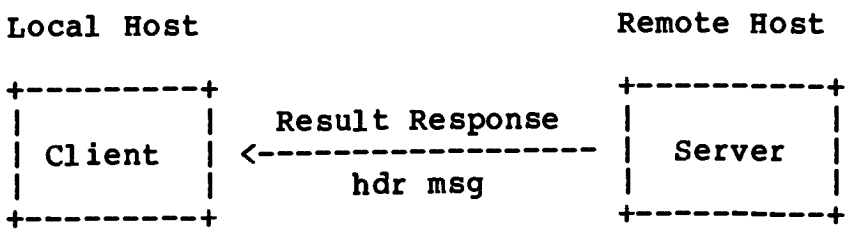
```
arg1 = fd
arg2 = offset
arg3 = "whence"
```

The lseek() can be executed and the result (new offset

location or -1 for error and associated errno value) returned to the client. The response is called a result response and consists of a message that contains the return code of the system call the errno value. The program interfaces are as follows:

```
Client
------
RecvResult(fd)

int     fd;

fd:     descriptor for connection to server

returns received return code.
asserts if any error on reception of return code.


Server
------
SendResult(fd, code, errno)

int     fd, code, errno;

fd:     descriptor for connection to client
code:   return code from system call
errno:  errno value on error

No return value, asserts that result is sent OK.
```

Diagramatically, a result response proceeds as follows:




Some system calls cannot be executed using the service request alone. For example, the write() system call requires extra information (the data to be written) before

execution can proceed. This transmission of "extra" information is called an extended service request, using the following interfaces:

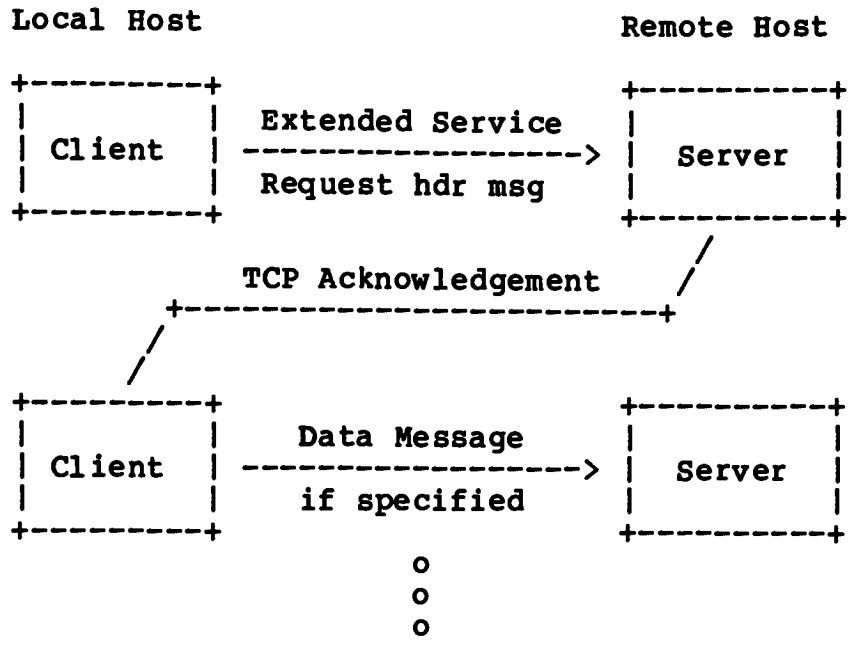
```
Client
------
SendData(fd, buf, len)

int     fd, len;
char    *buf;

fd:     descriptor for connection to server on
        remote host
len:    total number of bytes to be sent
        to server
buf:    data buffer
```

SendData() partitions the data buffer into messages each containing up to 512 bytes. Modifications were added to SendData() to convert non-character information in the hdr msg into network order.

```
Server
------
RecvWrite(fd, fdLocal, len)

int     fd, fdLocal, len;

fd:     descriptor for connection to client
fdLocal:fd for file to be written
len:    total number of bytes to be written
```

RecvWrite() can buffer up to a maximum of 512 bytes. Modifications were added to RecvWrite() to convert non-character information in the hdr msg from network to host order.

Diagramatically, an extended service request proceeds as follows:




The sequence repeats till all data is sent.

The last categories include system calls that return data to the user, for example read() and stat(). Read() is differentiated from stat(), fstat() and lstat() for two reasons:

(1) The amount of data returned for a read() is variable.

(2) The stat() type calls return non-character data that must be converted to network order before transmission.

The transmission of read data is called a read data response and the transmission of other data (i.e. stat()) is called a system data response. A read data response uses the following interfaces:

Client
---

```
RecvData(fd, buf, len)

int     fd, len;
char    *buf;
```

fd: descriptor for connection to server
len: total # bytes to be read
buf: read buffer

Returns the total number of bytes read. Otherwise, it returns -1 for an error and sets errno. Modifications were added to convert the hdr msg from network to host order.

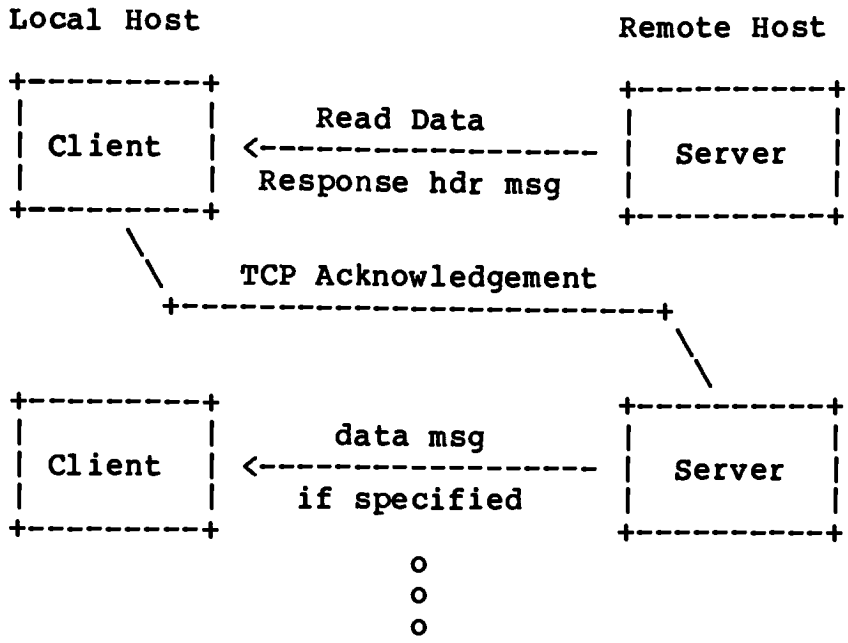
Server
---

```
ReadSend(fd, fdLocal, len)

int     fd, fdLocal, len;
```

fd: descriptor for connection to client
fdLocal: descriptor for file to read from
len: total # bytes to be read

Returns no value. Instead, ReadSend() asserts that the hdr and data msg's are sent correctly. Modifications were added to convert the hdr msg to network order.

Diagramatically, the read data response proceeds as follows:



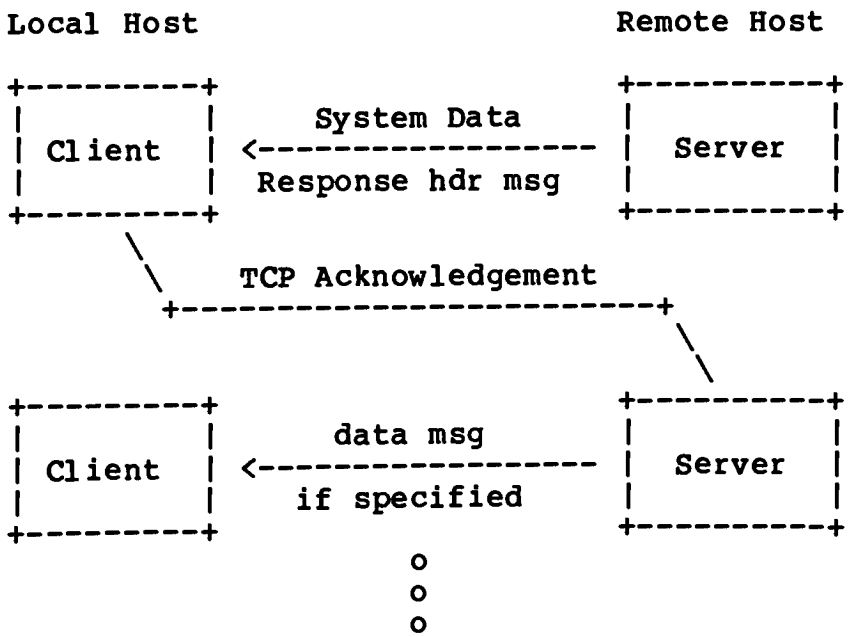

The sequence repeats until a hdr msg that indicates the last data msg or an error.

A system data response follows exactly the same logic as the read data response on the client side. On the server side, the system data response is handled similar to a read data response except that no read is performed. A separate routine called SendDatal() is used on the server side. The program interface to SendDatal() is:

```
Server
------
SendDatal(fd, buf, len)

int     fd, len;
char    *buf;

fd:     descriptor for connection to client
len:    # of bytes to send to the client
buf:    buffer containing information to be sent
        to client
```

SendDatal() assumes that the data to be sent is character data. Hence, network order encoding is not applied. Modifications were made to server system calls that send non-character data (i.e. stat(), fstat(), lstat()). These modifications encode non-character information to network order prior to invoking SendDatal().

Diagramatically, the system data response proceeds as follows:




The sequence repeats until a hdr msg that indicates the last data msg or an error.

## 5. IBIS Discrepancies and Shortcomings

(1) The user authentication used by IBIS does not allow individual system autonomy because of reliance on a global system wide mapping of user names to user-ids.

(2) Descriptor dependencies exist that require reserved allocation by IBIS of at least three descriptors to access the first remote system, and another descriptor for each additional remote system. This dependency is tightly coupled to the UNIX 4.2 kernel, and decreases the portability of IBIS. This is the prime reason that it was decided not develop a system III client. There was no way to detect this dependency until the port to system III was in the debugging phase.

(3) Pathname routing through multiple remote systems (servers) is not supported.

(4) Assuming a 4.2 client and server, if a user chdir's to a remote working directory with a pathname that contains a symbolic link, IBIS does not guarantee the correct operation of subsequent chdir's that use relative pathnames. Other remote file access systems also have problems with symbolic links (i.e. 4.2 / 4.3 remote mount).

(5) The utimes() system call is not supported. This affects programs that need to update the time fields in the inode, such as touch.

(6) Set-uid programs will not work correctly. This is a problem with other remote access file systems based on UNIX. (e.g. the Newcastle Connection, COCANET and Berkeley RFS mount).

(7) Remote temporary files constructed from the process id of the client, cannot be guaranteed to have unique names in the context of a network. This isn't a significant limitation, as such files should rarely be remote, because of the overhead of remote access.

(8) Incompatibilities between UNIX variants. This is a "large scale" problem that must be solved in order to preserve system call transparency. Generic functions can be mapped, but the non-generic functions that are needed by the system utilities of UNIX are hard to map between UNIX variants. In addition, IBIS is coupled to 4.2 TCP implementation support (i.e. getpeername()) that is not provided on other TCP implementations (i.e. Masscomp). Therefore, depending on diversity of capabilities needed, support of non-generic UNIX variant functions may in some cases be beyond the scope of IBIS.

(9) Performance impact of maintaining the state of many TCP connections. Perhaps a datagram protocol that can multiplex requests at a server should be used.

(10) Reliability issues. For example, crash recovery for remote file locking. Currently, this is only a concern between 4.2 systems since file locking it is not supported between variants.

(11) Reliability on the Masscomp. Because of the semantic differences of the accept() system call on the Masscomp, the server creator recreates another socket after forking a child for a client. When a client exits, TCP connections are not properly broken on the Masscomp. This leaves active servers in the system that are doing nothing but eating up process table space in UNIX.

## 6. Suggestions for Future Work

The current mechanisms IBIS uses to map remote files are too tightly coupled to 4.2 UNIX, and may not work on future versions of the kernel. Changes to these mechanisms require substantial source code modifications. I mention the pitfalls of the current approach in section 4.2, and provide some alternate design methodologies in section 4.3 of this paper.

The current 4.2 dependencies must also be removed if pathname routing through multiple (remote) systems is desired. Routing currently does not work, because if a pathname crosses a system twice, a file descriptor (i.e. allocated to the path on an open()) may have to refer to two different systems. Currently this is impossible, as a pseudo descriptor can only map one hidden descriptor. The user level version of a descriptor table introduced in section 4.3 could be extended two a two-dimensional table, where one axis is the descriptor index and the other is a "hop" index. Each time a request is routed through another server a hop value in the request is incremented before transmission. Thus, the hop value corresponds to the number of servers (nodes) the request has been routed through. Some maximum number of hops (route throughs) could be imposed so that the user level descriptor table can be statically allocated.

Adopting Newcastle or COCANET-like user authentication

would be an administrative improvement. The only change required is in the connection starter to map the user-name to local-host/user-name. In addition, a syntactic convention would be established to distinguish local from remote user-names.

Another extension would convert IBIS to use a true RPC protocol. Advantages are the cleaner programmer interfaces provided by the RPC, and a separation of the communication concerns from the application logic [Panzieri 1985]. The problem with this is that IBIS imposes restrictions on the operation of a server (i.e. to maintain fork semantics on remote files) that may not fit into the structure of an RPC server. The SUN NFS fits in cleanly with an RPC because no "memory" is required at the server. This constraint is lifted because the SUN NFS operates at the driver level within the kernel. Hence operations at this level are not concerned with preserving high level semantics but only those of a block device. Nevertheless, IBIS uses an "internal RPC" that could be further abstracted to allow easier modification of the package as a whole without having to actually modify IPC primitives, encoding/decoding, or messages exchanged. This may also allow cleaner support of UNIX variants, as one could call the appropriate remote routine based on the version of UNIX executing on the remote host. This would eliminate the need to massage messages as is currently done.

## 7. Conclusions

Development of a distributed / remote file system involves many issues:

Administrative issues: user authentication and security in a network environment. This is especially troublesome when servers are executing at the user level on an insecure system, communicating over insecure channels. Many feel security affects all aspects of an operating system and should be examined in that light [Hunter 1985].

Reliability issues: crashes must be detected in a distributed environment and handled correctly.

Performance issues: at worst it is hoped that the remote file system access capabilities are not slower than simple remote login. Meeting this performance goal depends on the type of load a remote file system package must contend with. Packages implemented at user level are subject to further performance impact by virtue of user mode execution (i.e. paging overhead, file system overhead already in the OS).

Developing a caching strategy for a distributed file system is a possible way to overcome performance problems. The trick is to obtain the proper balance between network overhead (IPC) and file system overhead (cache synchronization). Too centralized an approach may be easy to implement, but can quickly become a bottleneck. Too distributed

an approach can become equally as inefficient in an environment requiring frequent updates [Svobodova 1984]. More data need to be gathered to determine the nature of file system access in a distributed environment. As hardware costs decrease, it may be advantageous to adopt architectures that naturally enhance the sharing of resources [Luderer 1981], [Klienrock 1985].

Extensibility issues: can the system be reconfigured (i.e. systems added / removed) without interruption of service.

Adaptability issues: this ties back into the degree of access / location transparency provided to the user. There are tradeoffs between the amount of transparency desired and performance. For example, if no user updates are performed on the remote systems, then a query transaction approach could be adopted which does not require the exchange of state information between client and server.

In summary, the layered structure UNIX is designed on, seems well suited to distributed file systems. The numerous distributed file systems based on UNIX are evidence of the validity of this approach. In addition, the clean interfaces, common hierarchical name space (file, device, command), and use of a good systems programming language (C), are all attributes of UNIX that facilitate distributed file system development.

# BIBLIOGRAPHY

[Barbara 1984] Barbara, D. and Garcia-Molina, H., "How Expensive is Data Replication? An Example", Department of Electrical Engineering and Computer Science, Princeton University, Princeton N.J., 08544

[Brownbridge 1982] Brownbridge, D.R., Marshall, L.F. and Randell, B., "The Newcastle Connection or UNIXes of the World Unite!", Software -- Practice and Experience, 1982

[Fraser 1979] Fraser, A.G., "Datakit - A Modular Network for Synchronous and Asynchronous Traffic", Proc. ICC 1979, June 1979, Boston, Ma.

[Howard 1985] Howard, J.H., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., Spector, A.Z. and West, M.J., "The ITC Distributed File System: Principles and Design", Proc. of the Tenth ACM Symposium on Operating Systems Principles, December 1-4, 1985, Orcas Island, Washington

[Hunter 1985] Hunter, E., "Adding Remote File Access to Berkeley UNIX 4.2BSD Through Remote Mount", Report No. UCB/CSD 85/224, PROGRES Report No. 85.3, February 1985

[Israel 1979] Israel, J.E., Mitchell, J.G. and Sturgis, H., "Separating Data From Function in a Distributed File System", Operating Systems: Theory and Practice, North-Holland Publishing, 1979

[Klienrock 1985] Klienrock, L., "Distributed Systems", Communications of the ACM, November 1985, Vol 28, Number 11

[Luderer 1981] Luderer, G.W.R., Che, H., Haggerty, J.P., Kirslis, P.A. and Marshall, W.T., "A Distributed Unix System based on Virtual Circuit Switch", Proc. of the Eighth symposium on Operating Systems Principles, Dec 14-16, 1981, Asilomar, Ca.

[Lycklama 1978] Lycklama, H. and Christensen, C., "A Minicomputer Satellite Processor System", Bell System Technical Journal, Vol 57, No. 6, (July-August 1978).

[Lyon 1985] Lyon, B., Sager, G., et. al.,"Overview of the Sun Network File System", Sun Microsystems Inc, 1985

[MASSCOMP 1985] Masscomp Ethernet Managers' Guide, Order No. 075-00023-00, Revision B, February 1985

[Nowitz 1980] Nowitz, D.A. and Lesk, M.E., "Implementation of a Dial-up Network of UNIX Systems", COMPCON Fall 1980, 21st IEEE Computer Society Conference

[Panzieri 1985] Panzieri, F. and Randell, B., "Interfacing UNIX to Data Communications Networks", IEEE Tran. on Software Engineering, October 1985, Vol. SE-11, No. 10

[Perry 1961] Perry, M.N. and Plugge, W.R., American Airlines SABRE electronic reservation system. In AFIPS Conference Proceedings, Western Joint Computer Conference, Vol. 19, AFIPS Press, Arlington, Va., May 1961

[Peterson 1983] Peterson, J. and Silbershatz, A., Operating Systems Concepts, 1983, Addison Wesley Publishing Company

[Popek 1984] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G. and Thiel, G., "LOCUS A Network Transparent, High Reliability Distributed System", Proc. of the Eighth symposium on Operating Systems Principles, Dec 14-16, Asilomar, Ca.

[Ritchie 1974] Ritchie, D.M., Thompson, K.L., "The UNIX Time Sharing System", Communications of the ACM, July, 1974

[Rowe 1982] Rowe, L.A. and Birman, K.P., "A Local Network Based on the UNIX Operating System", IEEE Transactions on Software Engineering, March 1982, Vol 8, Number 2

[Schmidt 1980] Schmidt, E., "An introduction to the Berkeley Network", 4th Berkeley Software Distribution UNIX Documentation, Computer Science Division, University of California, Berkeley, Ca., 1980

[Swinehart 1979] Swinehart, D., McDaniel, G. and Boggs, D., WFS: "A Simple Shared File System for a Distributed Environment", Proceedings of the Seventh symposium on Operating Systems Principles, Dec 10-12, 1979, Asilomar, Ca.

[Tannenbaum 1981] Tannenbaum, A.S., Computer Networks, 1981, Prentice-Hall, Inc., Englewood Cliffs, N.J.

[Thompson 1978] Thompson, K., "UNIX Implementation", The Bell System Technical Journal, June 1978, Vol 57, Number 6, Part 2

[Thurber 1979] Thurber, K.J. and Masson, G.M., Distributed-Processor Communication Architecture, 1979, Lexington Books, Lexington, Massachusetts

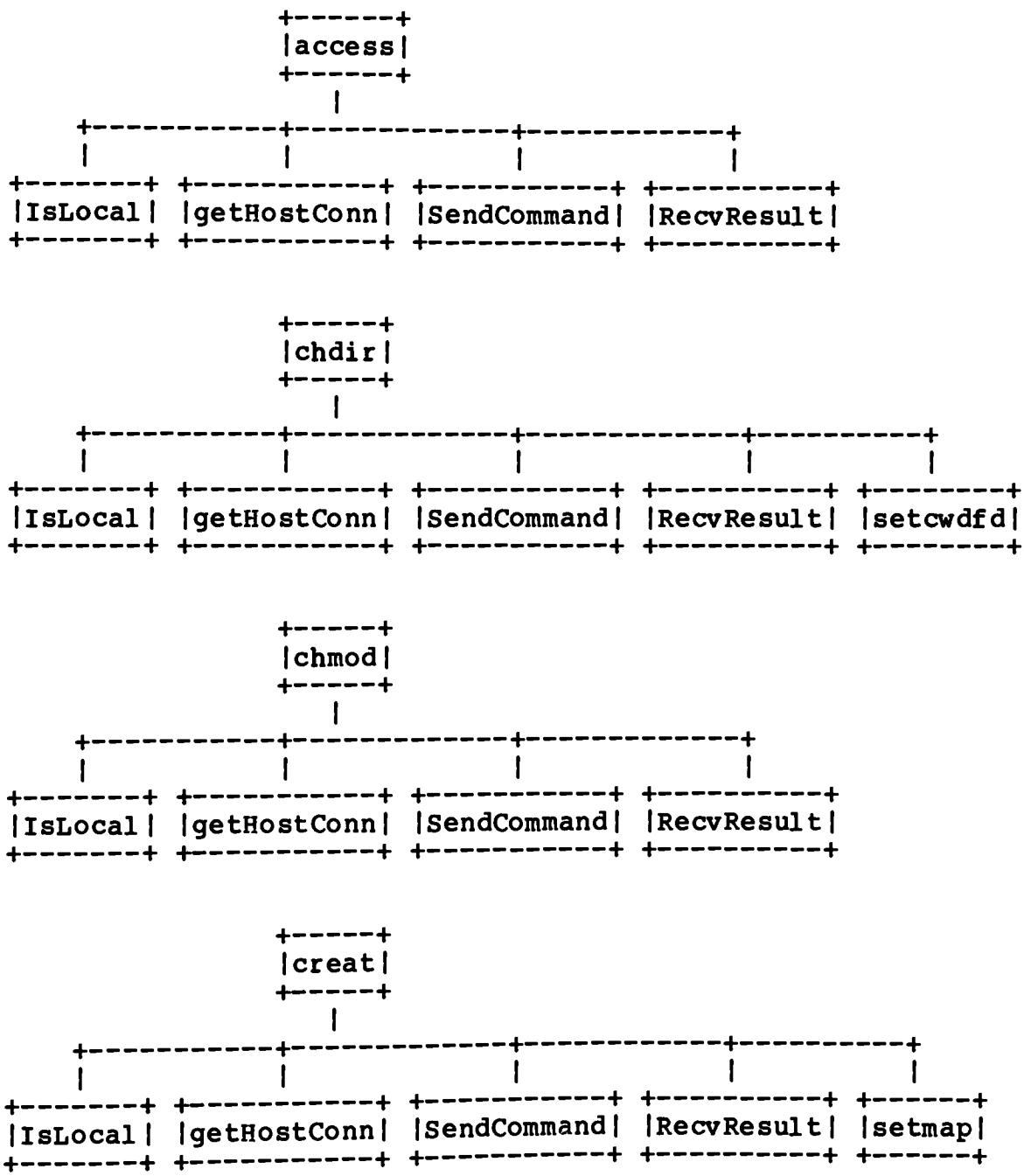
[Tichy 1983] Tichy, W.F. and Ruan, Z., "Towards a Distributed File System", Usenix Proceedings, 1985

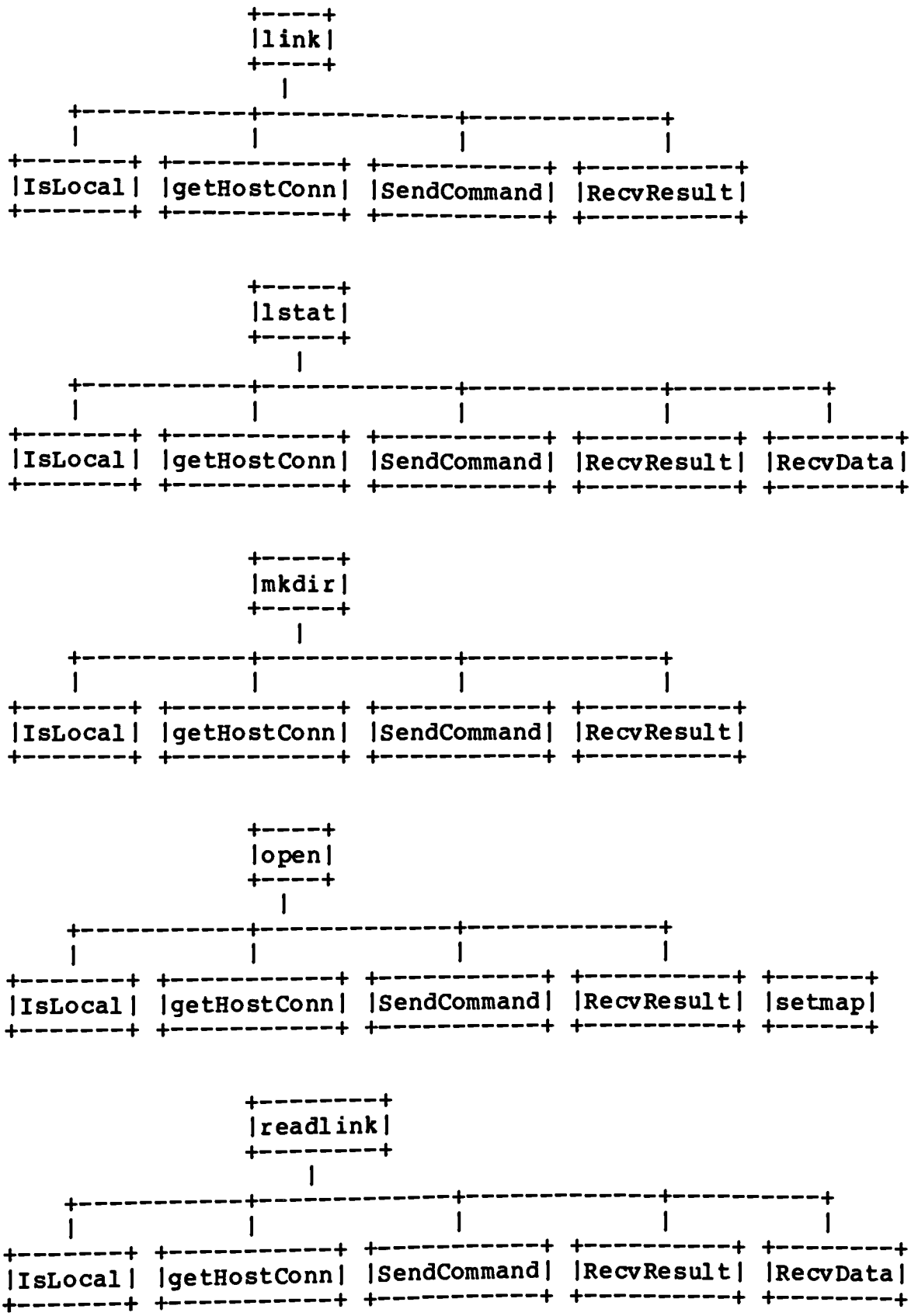
[Svobodova 1984] Svobodova, L., "File Servers for Network-Based Distributed Systems", ACM Computing Surveys, December 1984, Vol 16, Number 4

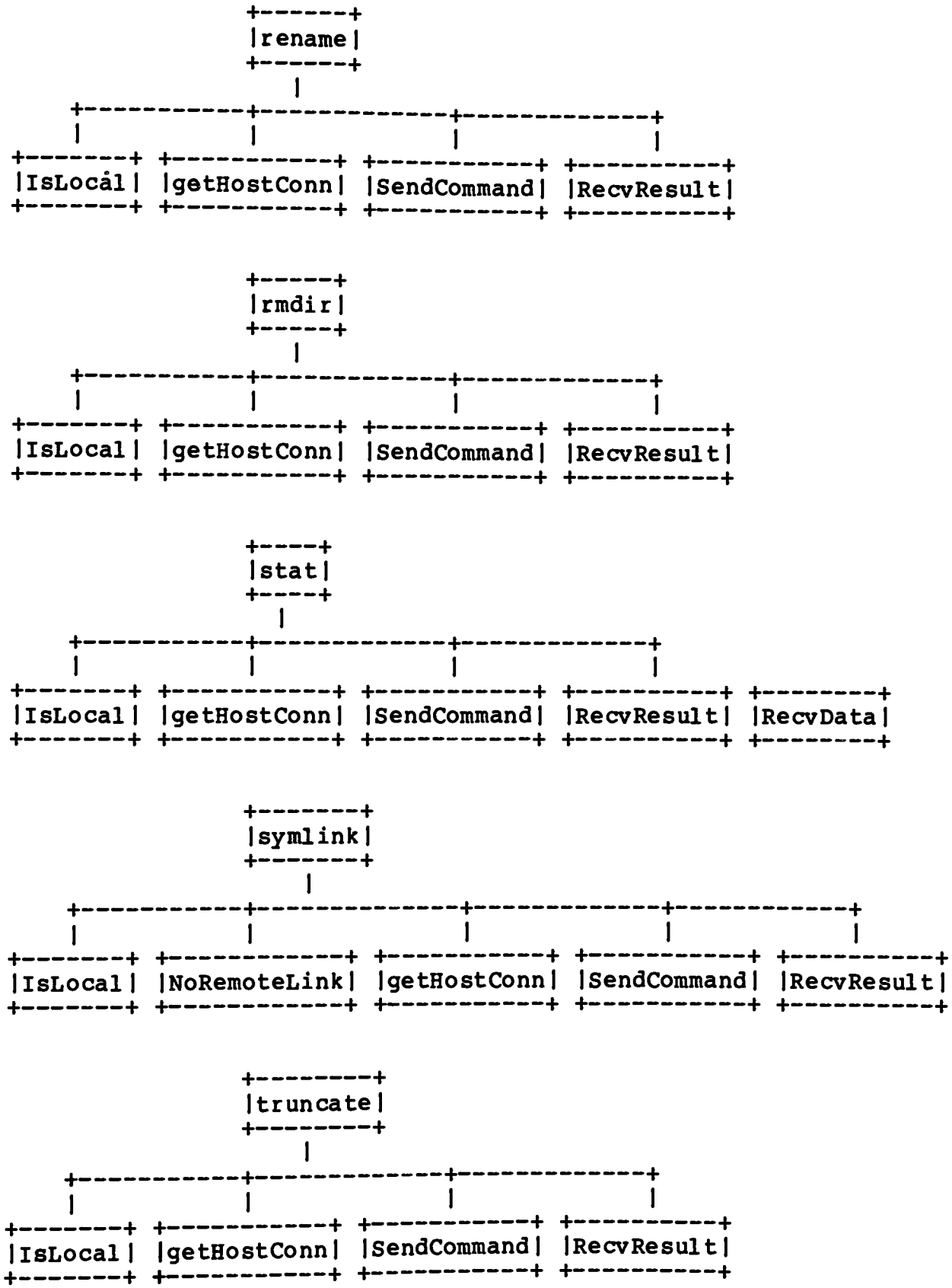
[UNIX 42] UNIX Programmer's Manual - 4.2 Berkeley Software Distribution

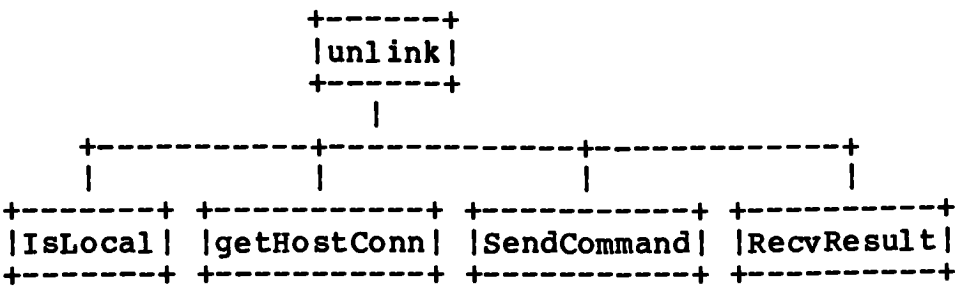
## Appendix A: IBIS Structure Charts

By system call, the IBIS client interfaces for system calls that use a pathname are as follows:

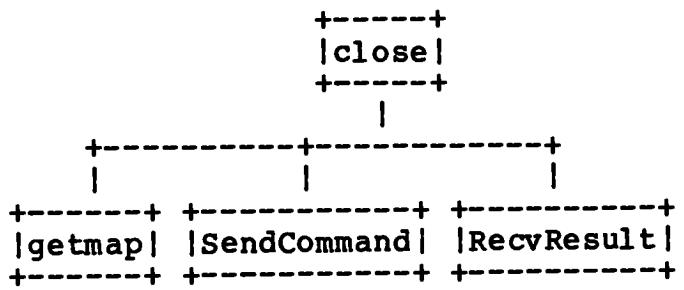


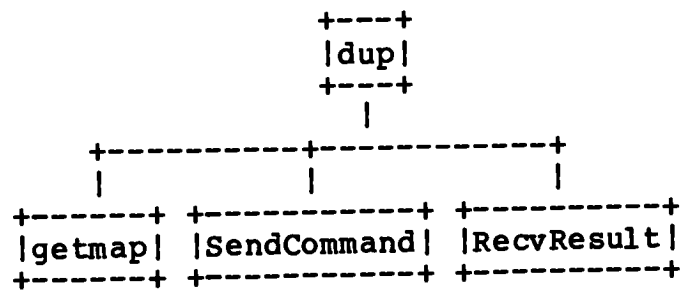

link
IsLocal
getHostConn
SendCommand
RecvResult
lstat
IsLocal
getHostConn
SendCommand
RecvResult
RecvData
mkdir
IsLocal
getHostConn
SendCommand
RecvResult
open
IsLocal
getHostConn
SendCommand
RecvResult
setmap
readlink
IsLocal
getHostConn
SendCommand
RecvResult
RecvData

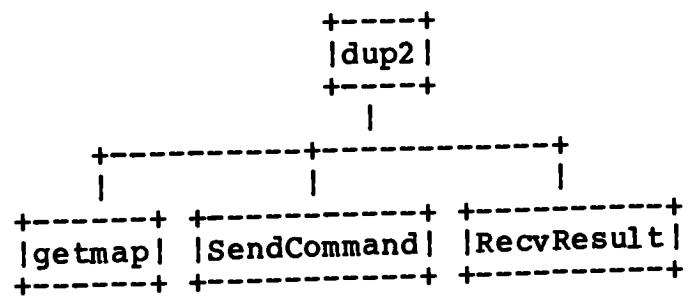

rename
IsLocal
getHostConn
SendCommand
RecvResult
rmdir
IsLocal
getHostConn
SendCommand
RecvResult
stat
IsLocal
getHostConn
SendCommand
RecvResult
RecvData
symlink
IsLocal
NoRemoteLink
getHostConn
SendCommand
RecvResult
truncate
IsLocal
getHostConn
SendCommand
RecvResult

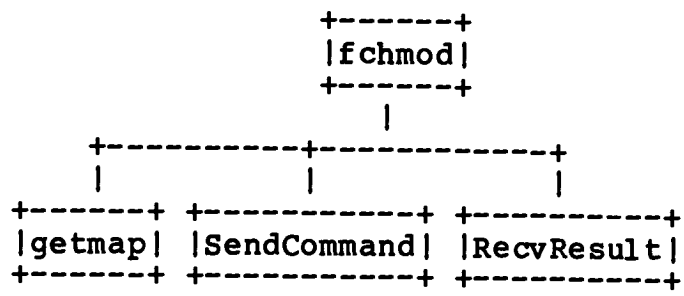


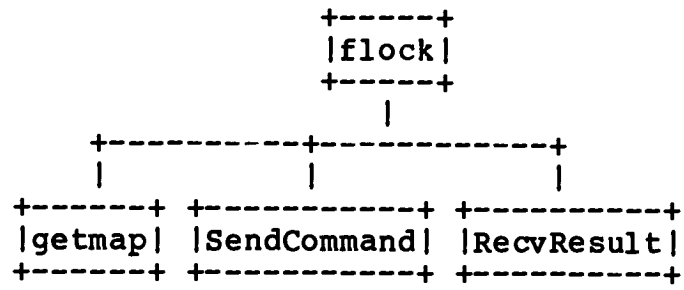

The IBIS client interfaces for system calls that use a file descriptor are as follows:

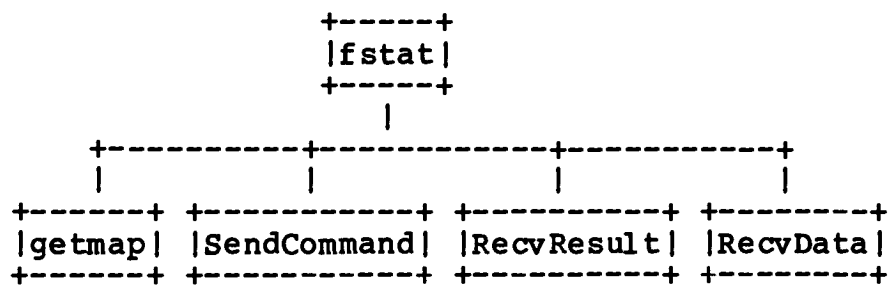


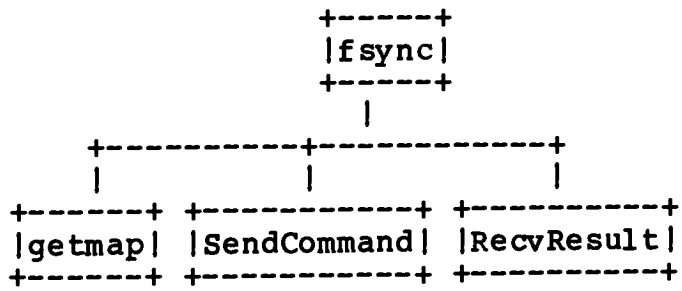



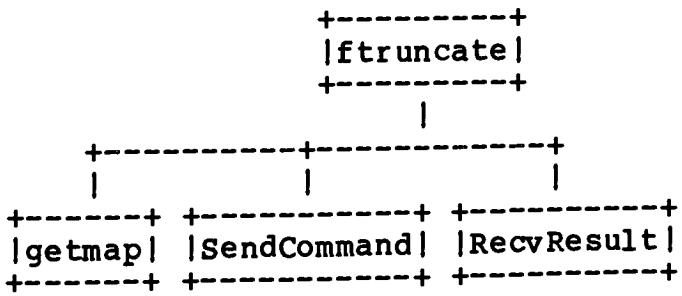



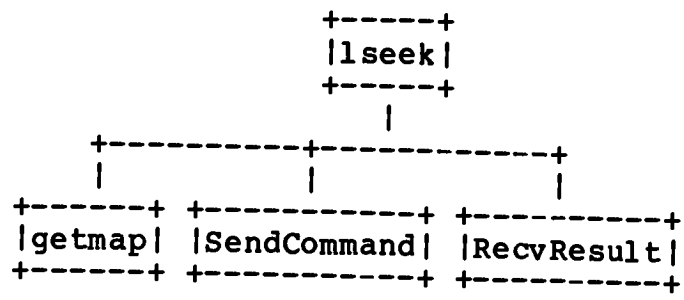

fchmod
getmap
SendCommand
RecvResult

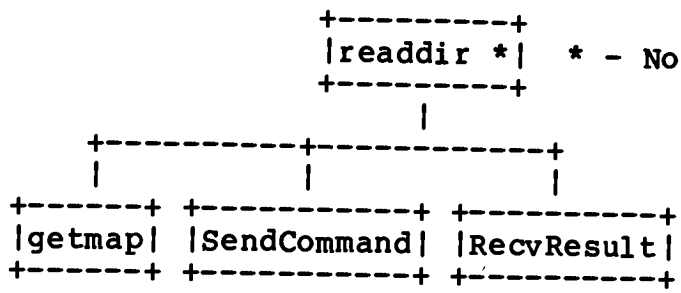

flock
getmap
SendCommand
RecvResult


fstat
getmap
SendCommand
RecvResult
RecvData

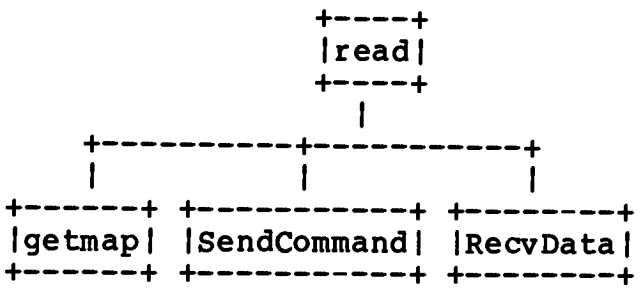

fsync
getmap
SendCommand
RecvResult

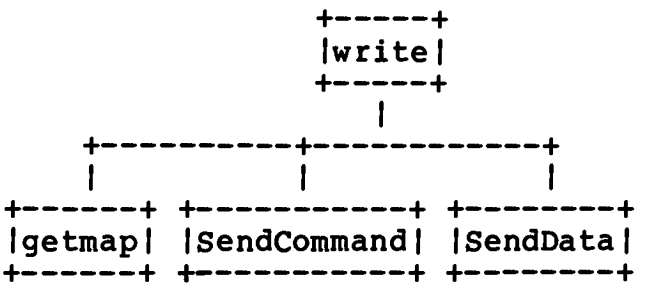

ftruncate
getmap
SendCommand
RecvResult



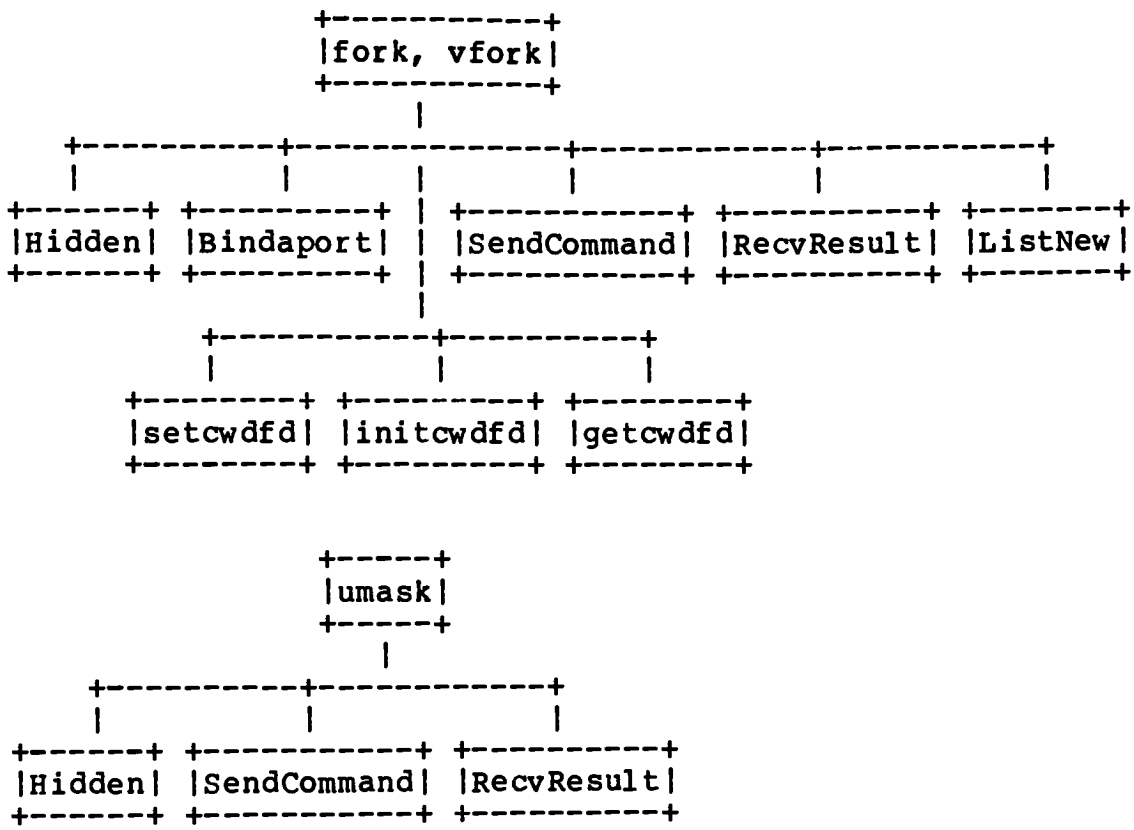




* - Note, a pseudo system call



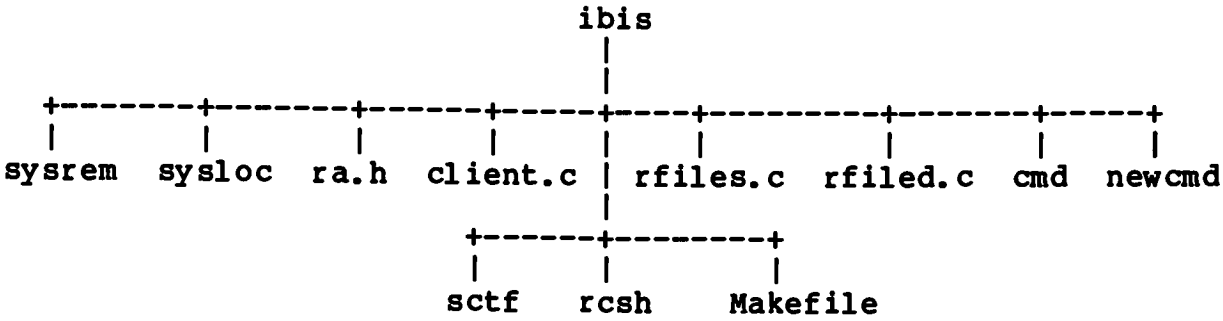




The IBIS interfaces for system calls requiring special processing are:


fork, vfork
Hidden
Bindaport
SendCommand
RecvResult
ListNew
setcwdfd
initcwdfd
getcwdfd
umask
Hidden
SendCommand
RecvResult

## Appendix B: IBIS Source Directory Organization

The directory structure for IBIS is as follows:




sysrem Contains the source code for remote access system calls that are redefined by ibis.

sysloc Contains the source code for local access system calls that are redefined by ibis.

ra.h Include file that defines various symbolic constants used within ibis source files.

client.c Contains the client interfaces for remote access system calls.

Makefile The ibis makefile for the remote access library (libra.a) the connection starter and server.

sctf A directory that contains the sources and Makefile for the system call test facility.

rfiles.c The ibis connection starter.

rfiled.c The ibis server.

cmd Contains the UNIX sources for the following commands:

newcmd Contains more UNIX sources:

rcsh The ibis remote csh source files.

The entire ibis packaged is generated as follows:

(1) Create the directory structure by tarring in the ibis tape.

(2) chdir to the ibis directory.

(3) Run the Makefile as follows:

```
make VERSION="-DTSTUSR" LOCMACH="-D<host>" \
     LIB="`pwd`/lib" \
     ETC="`pwd`/etc" \
     BIN="`pwd`/bin"
```

where <host> is either:

| | |
|---|---|
| VAX | for a VAX 11/7XX |
| P632 | for a Power 6/32 |
| MC2A | for a System III Masscomp |
| MC42 | for a 4.2 Masscomp |

(4) chdir to the sctf directory.

(5) Run the Makefile to create the sctf as follows:

```
make ANCHOR="`pwd`"
```

Note - the sctf can only be used to test local and remote access capabilities between 4.2 UNIX systems.

# Appendix C: IBIS Manual Pages

The manual pages follow this page.

**NAME**

System calls with remote access:

access, chdir, chmod, close, creat, dup, dup2, fchmod, flock, fork, fstat, fsync, ftruncate, link, lseek, lstat, mkdir, open, read, readlink, rename, rmdir, stat, symlink, truncate, umask, unlink, write

Stdio functions with remote access:

clearerr, fclose, feof, ferror, fflush, fgetc, fgets, fileno, fprintf, fputc, fputs, fread, fscanf, fseek, ftell, fwrite, getc, getchar, gets, getw, printf, putc, putchar, puts, putw, rewind, scanf, setbuf, setbuffer, setlinebuf, sprintf, sscanf, ungetc

Other library functions with remote access:

closedir, getwd, opendir, perror, popen, readdir, rewinddir, scandir, seekdir, telldir

**SYNOPSIS**

Same as those in UNIX Programmer's Manual (2) and (3).

**DESCRIPTION**

The functions listed in the first paragraph above mimic the system calls for file manipulation. The semantics are the same as described in UNIX Programmer's Manual (2), except for accepting remote file names or remote file descriptors.

Open/creat a remote file or dup a remote file descriptor returns a remote file descriptor, which can be used later on in read, write, lseek, fstat, dup, ... in the same way as an ordinary file descriptor. Remote file descriptors are inherited upon fork (in libra.a), vfork and exceve as long as the program to be exceve'd also uses the remote access versions of the system calls/library functions.

When an error occurs remotely during a system call, the errno is copied to the external variable errno of the client process to indicate the error condition.

The functions listed in the second and third paragraphs above are the same as those in libc.a, invoke the functions in the first paragraph instead of the standard system calls.

**RESTRICTIONS**

Most other system calls and library functions are not in the libra.a because they have nothing to do with remote access. There are a few exceptions which might accept remote file name/descriptor but whose remote versions are unavailable. Such system calls include chown, chroot, fchown, fcntl,

mknod, mount and umount.

**AUTHOR**

Zuwang Ruan, Purdue University

**SEE ALSO**

rcsh(1R), commands(1R)

intro(2), access(2), chdir(2), chmod(2), close(2), creat(2), dup(2), flock(2), fork(2), fsync(2), link(2), lseek(2), mkdir(2), open(2), read(2), readlink(2), rename(2), rmdir(2), stat(2), symlink(2), truncate(2), umask(2), unlink(2), write(2)

intro(3), directory(3), getwd(3), perror(3), popen(3), scandir(3)

intro(3S) and the whole section (3S)

Walter F. Tichy and Zuwang Ruan, Towards a Distributed File System, Proceedings of USENIX 1984 Summer Conference.

**NAME**

cat, chmod, cp, diff, ln, ls, mkdir, mv, rm, rmdir - remote access version

**SYNOPSIS**

Same as those in UNIX Programmer's Manual (1).

**DESCRIPTION**

The above 10 commands have the same meanings as the corresponding commands described in UNIX Programmer's Manual (1), except that they accept remote as well as local file names as arguments. The file name is in the form of [/net/<host>/]path. Where <host> is: cv for cinevax, vp for vaxpopuli, thor for thor, hoder for hoder, and tba for rittba.

If /net/<host>/ is missing, file name searching starts either from the root of local host (if path starts with '/') or from the current working directory. The current working directory may be either local or remote. If /net/<host>/ is present, the file name searching starts either from the root in the machine designated by <host>. /net/<host>/ can also be used if host is the local machine. In this case the path, refers to the root directory.

Cp, mv and rm provide the tomb option. The deleted file is actually relinked to the directory /usr/tomb/user-id in the machine where the file was, if /usr/tomb exists.

**RESTRICTIONS**

File searching can cross machine boundaries only once. For example, file searching fails in the following cases: the file name is /net/<host>/path where path contains another /net/<host>/ or path contains a symbolic link to /net/<host>/.

**BUGS**

Ln file1 file2 (hard link) complains "cross-device link" if one of the files is prefixed with /net/<host>/ but the other is not, even if the <host> is the local machine.

**SEE ALSO**

rcsh(1R), libra.a(3R)

cat(1), chmod(1), cp(1), diff(1), mkdir(1), mv(1), rm(1), rmdir(1)

Walter F. Tichy and Zuwang Ruan, Towards a Distributed File System, Proceedings of USENIX 1984 Summer Conference.

Modifications to support network special files (/net/<host>/) were added by Edward Ford.

**NAME**

rcsh - remote access version

**SYNOPSIS**

Same as csh but with remote access.

**DESCRIPTION**

Same as described in UNIX Programmer's Manual (1), except that the remote access facility is provided.

**Pathname searching:**

A pathname is in the form of [/net/<host>/]path where <host> is cv for cinevax, vp for vaxpopuli, thor for thor, hoder for hoder, and tba for rittba. If /net/<host>/ is missing, file name searching starts from the root of the local host (if path starts with '/') or from the current working directory, which may be in the local host or a remote host. If /net/<host>/ is present, file name searching starts from the root directory of the specified host (if path is missing) or from the specified path.

**Current working directory:**

The current working directory may be either in the local host or in a remote host. The builtin commands cd, chdir, pushd, popd and dirs manipulate the current working directory in the same way as described in csh(1), no matter if it is local or remote. Cd or chdir with no argument means to change the current working directory to the local home directory. Cd, chdir and pushd use cdpath to look for a directory when the normal pathname searching (described above) fails. However, only the cdpath in the local host is used, even though the current working directory is remote.

**Name substitution (globbing):**

The characters `*', `?', `[' and `{' in path are expanded in directories of local or remote hosts where the file name searching is going on there. `~' can be expanded in the local host if /net/<host>/ is missing and '~' is the first character of the pathname.

**I/O redirection:**

Standard input, standard output and diagnostic output can be redirected to remote files in the same way as to local files.

**Connection establishment:**

A connection for remote access to a host can be established either explicitly by the builtin command raon, or implicitly on demand, i.e. a connection is established automatically when a process attempts to access to an object in the remote host for the first time. In either case, the connection is closed when the process finishes and inherited by its descendent processes, if any. Connections established by shell can also be closed explicitly by the builtin command raoff.

**Builtin commands:**

There are three new builtin commands:

raon <host> [user]

establishes a connection of remote access to the specified host for the current shell. The remote user name is given by the specified user, or by default (as described below) if user is omitted.

raoff [<host>] ...

closes the connection(s) to the specified host(s) or all established connections of the current shell if host is omitted. The connection to the host where the current working directory is cannot be closed.

rastat [<host>] ...

prints the message:

<host> is RA-on with remote user user, or

<host> is not RA-on

for the connection(s) to the specified host(s), or all connected hosts if <host> is omitted.

**Default remote user name:**

Except the remote user name is given by raon command, the default remote user name is used. If the remote host appears in the /etc/hosts.equiv in the local machine, the local user name is used as the default remote user name. Otherwise, it looks up the file .rhosts in the local user's home directory, in which each line contains a host user pair. The user name in the line of the first appearance of the remote host is chosen as the default remote user name. If no such line exists, the establishment fails.

The server process (rfiled) is responsible for authentication checking. It uses the same method as that used by rsh

and rlogin. In brief, a connection requested by user A from host P to host Q with user name B can be approved only if P appears in Q's /etc/hosts.equiv and A is the same as B (and is not the root), or the .rhosts in B's home directory in host Q contains the P A pair.

The other functions such as command interpretation, history substitution, alias substitution and variable substitution are the same as described in UNIX Programmer's Manual (1).

**RESTRICTIONS**

Rcsh supports remote file access but not remote command execution. The commands are always executed locally. The default input/output is the local terminal. For example, the following commands are legal:

```
cat /net/tba/... /net/thor/... | pg

ls -l /net/tba/... | grep ... > /net/thor/...
```

But

```
ls ... | /net/thor/bin/grep ...
```

does not work.

In this version of rcsh, the current working directory is only used for pathname searching, but not command searching. The commands are always searched according to the local path. Even if the "." appears in the local path and the current working directory is remote, rcsh does not look in the remote directory for commands.

**BUGS**

Rcsh does not guarantee that a chdir to remote directory via a path that contains a symbolic link will work correctly. In particular, the chdir may be succesfull but subsequent attempts to chdir with relative paths (i.e. ../) may not cross back from the remote to local machine correctly.

**AUTHOR**

Zuwang Ruan, Purdue University

**SEE ALSO**

commands(1R), libra.a(3R)

csh(1), rsh(1)

Walter F. Tichy and Zuwang Ruan, Towards a Distributed File System, Proceedings of USENIX 1984 Summer Conference.

Modifications to support network special files (/net/<host>), generalized remote working directories, and heterogenous machines were added by Edward Ford.