40

1561

Bell Laboratories Cover Sheet for Technical Memorandum

The information contained herein is for the use of employees of Bell Laboratories and is not for publication (see GEI 13.9-3)

Title: **A Distributed UNIX™ System**

Date: **May 9, 1980**

Other Keywords: **Network**
Datagram
Remote File System

TM: **80-3168-6**

Author(s)	Location	Extension
Alan L. Glasser	**HO 1E-335**	**6569**
David M. Ungar	**HO 1E-335**	**3892**

Charging Case: **49408-120**
Filing Case: **40324-2**

ABSTRACT

This paper describes a distributed UNIX system that supports a community of several hundred computer programmers. Individual UNIX systems are connected with network software and vendor-supplied communications hardware to provide *transparent* read-only access to all files on all processors, and a set of commands for explicit distributed data manipulation and resource sharing (e.g., line printers and remote job entry connections).

The system can withstand a number of hardware failures by automatically routing data around malfunctioning processors and links. Also, no more human intervention is required than would be if the same number of processors were operated in a non-distributed mode.

Network file accesses take four times longer than local file accesses, providing adequate performance. The system is compared to a Programmer's Workbench provided on a large computer. Experiences with the system and future work directions are also presented.

The attached paper has been submitted to COMPSAC 80 (IEEE Computer Society's Fourth International Computer Software and Applications Conference, Chicago, October 27-31, 1980).

Pages Text: 1	Other: 13	Total: 14
No. Figures: 4	No. Tables: 0	No. Refs.: 5

BELL TELEPHONE LABORATORIES, INC.

TM-80-3168-6

DISTRIBUTION
(PAPER GEI 13.9-3)

COMPLETE MEMORANDUM TO

CORRESPONDENCE FILES

OFFICIAL FILE COPY
PLUS ONE COPY FOR
EACH ADDITIONAL FILING
CASE REFERENCED

DATE FILE COPY
(FORM E-1328)

40 REFERENCE COPIES

AITCHESON,E J
ABRAM,THOMAS D
BARTON,M E
BAUER,H C
BAVISH,RICHARD J
BEAUMONT,LELAND R
BECKETT,J T
BLOSSER,PATRICK A
BOEHM,KARL W
BREITHAUPT,A H
CAMPBELL,JERRY H
CANAC,F.EDLD H
CHEN,ROBERT
CICON,J F
CRISIOFOR,EUGENE
DE GRAAF,D A
DOCK,G A
>DOLOTTA,T A
DOMPIERRE,J A
DONOHUE,E F,JRE
DOUGHERTY,E J
DUSMAN,E R
DWORAK,F S
ELDRIDGE,GARY P
FISHMAN,DANIEL H
FREEMAN,K G
GARST,ELAINE,JR
GIBSON,H T,JR
GILLON,ALEX C
GIMMELLI,RALPH T
GLASSER,ALAN L
GUIDI,PIER V
>HAIGHT,E C
HAISCH,H F,JR
HALLER,K M
HALL,ANDREW D,JR
HANSEN,R C
HARKNESS,CAROL J
HARRIS,BRENDA L
HAYDEN,DONALD F,JR
HEATER,ROBERT J
HEFFRON,W GORDON,JR
HEIDER,BRUCE H
HERGENHAN,C E
JAASKA,E G
JONES,E E
KANE,J RICHARD
KAPLAN,FRANK
KAUFELD,J C,JR
KNEUER,JOSEPH G
KOZAREK,J F
LANLESS,WILLIAM J
LAYTON,H J,JR
LEEPER,EVELYN C
>LESK,MICHAEL E
LUDERER,GOTTFRIED R E
MACRI,P P
MACUR,H P
MAIK,JOSEPH M
MARIONE,JOHN F
MARASCO,ROBERT S
MARTELLOTTO,E A
>MASHEY,J R
MASA,C A
MAURSELL,H I
MC CABE,F S
MC DONALD,J F
MC INTIRE,R E
MCCULLOCH,JOHN W
MILLER,JC ANDE H
MITCHELL,J C
MITZE,ROBERT W
MIRDFELDT,R G
MOWITZ,D A
ONCHUNDRO,WAYNE E
OSCHARD,E A
PASTERNACK,G
PETERSON,RALPH W
PILLA,M A
PRIEVE,BARBARA G
RALEIGH,T M
REDERER,D L
REED,R A
>RITCHIE,D M
RODRIGUEZ,ZERESIC J
ROVEGNO,HELEN C
ROWLAND,BRUCE R
ROY,SUMIT
RUBINSTEIN,PETER
>SASSEVITZ,A L
SALOMON,FRED A
SCHMITT,A A
SCHWARTZ,R C
SHAER,N R
SLANA,M F
SNYDER,BERNARD E
STOREY,THOMAS F
STREETER,LYNN A
STUBBLEFIELD,R W
SON,YIH-SHYONG
TARLOSKI,THEODORE F,JR
TAGUE,BeSELEY A
>THOMPSON,K
TING,DENNIS WAY
UNGAR,DAVID M
VEENA,SHIV P
WEIT,M J
WENDT,TIMOTHY A
WONSIEWICZ,B C
YACOBELLIS,ROBERT E
YOSK,W L
ZIMMER,DAVID A
ZISLIS,PAUL M
143 NAMES

COVER SHEET ONLY TO

CORRESPONDENCE FILES

4 COPIES PLUS ONE
COPY FOR EACH FILING
CASE

AAGESEN,JOHN
AAFONSON,STEVE M
ABATEMARCO,TERESA M
ABATE,JOSEPH
ACKERMAN,A FRANK
ACKERMAN,J E
AHO,ALFRED V
AHRENS,RAINER B
AHUJA,SUDHIR R
ALBAGLI,V A
ALBERTS,BARBARA A
ALCALAY,D
ALEXIS,A D,JR
ALKENS,FREDERICK
ALLEN,JAMES R
ALLES,H G
ALLISON,C A,JR
AMABILE,GEORGE A
AMITAY,N
AMMIRATA,P T
AMOSS,JOHN J
ANDERSON,C R
ANDERSON,KATHRYN J
ANDERSON,A G
ANDERSON,MILTON M
ANDERSON,R E
ANDERSON,R E
ANDREWS,W J
ANTICLICH,DAVID R
ANTONELLI,CHARLES J
ARCHER,RUSSELL A,JR
ARNSTEIN,D R
ARMSTRONG,F O,JR
ARNOLD,GEORGE W
ARNOLD,JAMES Q
ARNOLD,PHYLLIS A
ARNOLD,THOMAS F
ASPIN,LEE J
ASTHONS,A
ASTIS,H P
ATVIDSON,W F
ATELTINE,EDWARD G
ATSMUTH,MICHAEL L
ATSTHANA,VISHAYA
ATAL,BISHNU S
AXELSON,A L
AYLWARD,ELIZABETH
BACCASH,JEANNE M
BACH,MAURICE J
BACKUS,C F,SR
BAGLEY,JOHN D
BAILEY,CATHERINE T
BAKER,BRENDA S
BAKER,DONN
BAKER,MITCHELL B
BALDWIN,GEORGE L
BALENSON,CHRISTINE M
BALLANCE,ROBERT A
BANG,SUNG YANG
BARBATO,ROBERT R
BARNHARDT,KARL R
BARONSKY,ALLEN
BARON,ROBERT V
BARRESE,A L
BARR,DAVID I
BARR,W J
BATISTONI,F J
BATTAGLIA,FRANCES
BAUER,BARBARA T
BAUER,HELEN A
BAUER,HOLFGANG F
BAUGH,C R
BAXTER,LESLIE A
BAYER,D L
BECKER,JACOB I
BECKER,RICHARD A
BEDNAR,JOSEPH A,JR
BEIGHLEY,KEITH A
BENCO,DAVID S
BENISCH,JEAN
BENNETT,RAYMOND W
BENNETT,RICHARD L
BENNETT,WILLIAM C
BENOWITZ,P
BENSING,JAMES EDWARD
BERENBAUM,ALAN
BERGERON,R F,JR
BERGLAND,G D
BERKEY,K A
BERKOWITZ,PAUL R
BERK,DONALD A
BERNHARDT,RICHARD C
BERNOSKE,BEVERLY G
BERNSTEIN,DANIELLE R
BERNSTEIN,L
BERNSTEIN,PAULA R
BERRYMAN,R D
BERZINS,ALEXANDER H
BHATIA,RAJIV
BIANCHI,M d
BICKFORD,NEIL R
BILLASH,TIMOTHY D
BILOWOS,R M
BIREN,ISHA R
BISCHOFF,R BARRY
BISHOP,J DANIEL
BISHOP,THOMAS P
BITTNER,R R
BITTRICH,MARY E
BLAKE,GARY D
BLAZIER,S D
BLEILE,JOSEF
BLINCER,S
BLINN,J C
BLUM,MARION
BOCKUS,ROBERT J
BOCK,NANCY E
BODDIE,JAMES R
BOEHM,F J
BOHNER,H A
BOGART,F J
BOGART,THOMAS G
BOLVIE,RICHARD H
BOLINSKI,NANCY V DEVLIN
BONACHEA,R N
BONANNI,I E
BOND,F C
BOND,HOLTON C,JR
BOND,KEVIN E
BOSISOY,ZILPA A
BOSKAR,SURESH R
BOSZ,DEBASISH
BOSTON,RONALD E
BOSWELL,PAULA S
BOTHUN,R H
BOURNE,STEPHEN R
BOYER,L RAY
BOYCE,R M
BOYER,PHYLLIS J
BOYLE,GERALD C
BOYSEN,A F,JR
BRADFORD,EDWARD G
BRADLEY,M HELEN
BRANDAUER,C M
BRANDT,RICHARD B
BRAUN,A R
BRAUN,DAVID A
BRAUN,E J
BREILAND,JOHN R
BRENSKI,EDWIN F
BRESLER,RENEE A
BRITT,WARREN G
BROAD,MARTHA M
BROWNSTEIN,N
BROOKS,CATHERINE ANN
BROSS,JEFFREY D
BROYMAN,INNA
BROWN,C W
BROWN,ELLINGTON L
BROWN,JAMES N
BROWN,LAURENCE MC FEE
BROWN,MARK S
BROWN,STUART G
BROWN,W R
BROWN,W STANLEY
BRUZ,G C
BRUECKNER,DOUGLAS E
BRYANT,DAVID J

1443 TOTAL

+ NAMED BY AUTHOR > CITED AS REFERENCE < REQUESTED BY READER (NAMES WITHOUT PREFIX WERE SELECTED USING THE AUTHOR'S SUBJECT OR ORGANIZATIONAL SPECIFICATION AS GIVEN BELOW)

MERCURY SPECIFICATION.....

COMPLETE MEMO TO:
316-SUP 364-SUP 3168-TM 3633-SUP 3121-SUP 3122-SUP

COVER SHEET TO:
1712-SUP 323-SUP 324-SUP 3491-SUP 932-SUP 933-SUP 37-CSH 313-CSH 127-MTS 1353-MTS
135-DSH

COCSSI = COMPUTING SYSTEM INTERCONNECTION, NETWORKS: SOFTWARE ASPECTS
UNOS® = UNIX OPERATING SYSTEM: GENERAL OR SURVEY DOCUMENTS

NO CORRESPONDENCE FILES
NO TM47

TM-80-3168-6
TOTAL PAGES 11

TO GET A COMPLETE COPY:

1. BE SURE YOUR CORRECT ADDRESS IS GIVEN ON THE OTHER SIDE.
2. FOLD THIS SHEET IN HALF WITH THIS SIDE OUT AND STAPLE.
3. CIRCLE THE ADDRESS AT RIGHT. USE NO ENVELOPE.
4. INDICATE WHETHER MICROFICHE OR PAPER IS DESIRED.

PLEASE SEND A COMPLETE

( ) MICROFICHE COPY ( ) PAPER COPY

TO THE ADDRESS SHOWN ON THE OTHER SIDE.

Bell Laboratories

subject: **A Distributed UNIX™ System**
Charge Case 49408-120
File Case 40324-2

date: **May 9, 1980**

from: **Alan L. Glasser**
HO 3168
1E-335 x6569

David M. Ungar
HO 3168
1E-335 x3892

TM 80-3168-6

MEMORANDUM FOR FILE

The attached paper has been submitted to COMPSAC 80 (IEEE Computer Society's Fourth International Computer Software and Applications Conference, Chicago, October 27-31, 1980).

Alan L. Glasser

David Ungar

Alan L. Glasser

David M. Ungar

HO-3168-ALG/DMU-alg

Att.
One memo
References
Appendix
4 Figures

A Distributed UNIX™ System

Alan L. Glasser
David M. Ungar

Bell Laboratories
Holmdel, New Jersey 07733

1. INTRODUCTION

1.1 The Need for A Distributed UNIX System

The Programmer's Workbench UNIX system [1, 2] is a specialized computing facility dedicated to supporting large software development projects. The system is usually run on a DEC PDP-11/70 minicomputer. Unfortunately, the number of users that a single PDP-11/70 UNIX system can support[1] is not sufficient for most sites, and multiple system configurations are quite common. Each system usually supports an independent user community. Inter-system file transfer facilities are used extensively. These facilities typically rely on dial-up telephone ports with automatic call units [3] or remote job entry facilities to a common host [4]. When each system's users form an independent community such facilities suffice. However, when a *single* user community must span multiple systems, *transparent* access to all files is far superior to file transfer commands.

The authors have been members of such a single, spanning community for the past few years. Originally, it was small enough to reside comfortably on a single system. As the number of users grew, it became necessary to add a second system, connected to the first one by a high speed file transfer facility. More systems were added, and the difficulties that were caused by the fragmentation of the community prompted an investigation of techniques to unify the systems. This investigation led to a definition and a *read-only* implementation of a remote file system.

A remote file system possesses the following characteristics:

Location independence	The user does not have to know where a file resides; a command produces the same results if a given disk pack is mounted on system A or system B.
Homogeneity of file names	Remote file names have the same syntax as local file names. Furthermore, the name of a file is independent of the system the user resides on and independent of the system the file resides on.
Homogeneity of access	A user process can "open", "read", "write", "close", etc. a remote file in exactly the same way as it a local file.

Additionally, it is important that all *existing* programs be able to access remote files without modification. In summary, a remote file system is *transparent;* one neither needs to know, nor does know, on which system a given file resides.

1.2 The Users' View

We have implemented a read-only remote file system. Each user has read/write access to files on the local processor and read-only access to files on every other processor. The greatest benefit that has accrued from this system is the ability to share data without duplicating files. Project source code is maintained in one place, and the remote file system provides access to it for all users. An ability to execute remote commands provides the basis for source update facilities. These facilities are built on the underlying network software and the remote file system. Other facilities, that actually predated this work, such as file transfer service, line printer or remote job entry service (where the printer/CPU or RJE/CPU hardware connections

1. The maximum is 48, but a total of 32 simultaneous users is more typical.

are hidden from the user), and inter-system mail service are still provided. The syntax of the commands to use these services, which is independent of the hardware connections, has been maintained. The Appendix gives some examples of usage.

We have attempted to partition our community in a way that minimizes remote file system traffic. Similar partitioning is often done for a single system community to balance disk traffic or to ease the sharing of disk files. The result of our efforts is that we have constructed a viable, large-community, distributed UNIX system out of smaller, single computer systems.

1.3 The Underlying Network

The facilities described above need to send data from one system to another and to perform actions on remote systems. A datagram-based network has been developed to fill these needs and to realize such transport functions as routing, network recovery, and link multiplexing.

The design of the network has concentrated on three issues: recovery, maintainability, and performance.

Recovery — No human intervention is required to inform or reconfigure the network when systems crash or are bootstrapped. Each node monitors its neighbors and configures itself accordingly. In addition, the daemon processes possess enough intelligence to automatically recover from most errors. Finally, each datagram contains a checksum that facilitates the rejection of noise.

Maintainability — No constraints are placed on the interconnection topology, with the exception that communications links must be bidirectional. The routing algorithm automatically chooses the shortest path.

The network can be supported on a wide variety of bidirectional communications devices. We currently use the DEC Parallel Communications Link (PCL-11) interprocessor, time division multiplexed bus. In the past we have also used the DEC DMC-11 synchronous, point-to-point network link. Device dependent software is confined to the device's driver and a simple listener process.

The operating system kernel (PWB/UNIX 2.0) has not been changed, although device drivers have been added.

Performance — Rapid transport is attained with off-the-shelf hardware. A remote transaction takes about 60 ms. of real time. This results in remote file accesses that are roughly four times slower than local file accesses.

The functionality of the network has been limited to the bare essentials. Only capabilities needed for the remote file system are implemented. No attempt has been made to provide virtual circuits, datagram sequencing, or end-to-end assurance.

2. THE REMOTE FILE SYSTEM

2.1 The UNIX File System

To appreciate many of the problems of implementing a remote file system, a basic understanding of the UNIX file system [5] is necessary. The overall structure of the file system is that of a rooted tree composed of *directory files*, *regular files*, and *special files*. Directories provide the mapping between the names of files and the files themselves. Special files are like ordinary disk files, but requests to read or write result in the activation of an associated device.

The file system is device independent, and portions may reside on several different types of devices. The device containing the root of the file system tree is typically specified in the system configuration data, but this is usually a small portion of the entire hierarchy. Each

device contains its own, independent file system tree. The UNIX *mount* command replaces a leaf of the hierarchy tree by the tree on the designated device. All requests for blocks within this sub-tree are given to the device driver specified in the command.

At the lowest level of the file system software is a buffer cache. Higher level routines either request blocks of data from the buffer cache, or return data to the buffer cache. Blocks are addressed with a device number—block number pair. The buffer cache routines invoke the appropriate device driver routines to fill or empty cache blocks as appropriate. These routines have no knowledge of the file system structure. In particular, they cannot distinguish file system control blocks (such as free list blocks) from data blocks. The relationship between a process, the file system software (including the buffer cache), and a disk device driver is illustrated in figure 1a.

It is very difficult to implement a remote file system with full read/write capabilities. The locking and mutual exclusion mechanisms in the UNIX time-sharing system operate at a higher level of abstraction, and thus become ineffective if copies of any particular control block are cached on multiple processors. Operations that modify the file system rely heavily on the buffer cache to allow the efficient modification of control blocks.

Implementing a read-only remote file system at this level is straightforward. A mechanism is needed to translate a request for a remote file system block into a request to the remote machine. This mechanism can be realized in a simple *pseudo-device* driver, and the datagram network software. This interposition is illustrated in figure 1b.

2.2 The Remote File System Pseudo-device Driver

The driver consists of a *block* driver and a *character* driver.[2] The novelty of this implementation is that, unlike traditional combined block and character devices (such as a disk driver), this driver assigns very different semantics to a block read versus a character read. Also, a character write is supported (with unusual semantics), while a block write is explicitly prohibited.

The block interface simply queues requests for remote blocks. The queueing discipline is strictly first-come-first-served.

The character read interface allows a server process to retrieve an entry from the request queue and forward it to a remote machine. That is, the character read interface returns a *request*, not a block of data as does the block interface. The character write interface allows a server process to satisfy a request by returning a block from a remote machine to the original requester. Thus, the write interface does *not* write remote files, but provides a mechanism for returning data from a remote machine to the kernel of the local machine.

A request consists of a device number and a block number. This *tag* is returned with the data via the write interface to match the returned data with the original request. The data are then passed to the process that initiated the request, which may then resume execution (the read has completed). The buffer cache mechanism must be disabled for the remote file system driver because a block fetched from another system and cached locally can become obsolete when it is subsequently changed by the remote processor (see section 4, below).

2.3 Error Recovery

The driver has been designed to be robust and not require perfect underlying network software and hardware. Datagrams are used for both requests and responses. Some important aspects of

---

2. In the UNIX time-sharing system, a block device consists of randomly addressable, secondary memory blocks of 512 bytes each. A character device is any device that does not fall into the block device model. Disk drivers usually support both a block device and a character device interface (the character interface allows I/O to be done in arbitrary block sizes). However, the buffer cache interfaces to block devices only.

the driver's error recovery strategy are:

— Each request for a block is an independent atomic transaction. Thus, the driver can perform error recovery on one request at a time, without concern for other pending requests. *The network need not preserve the ordering of datagrams.*

— Pending requests are timed out after fifteen seconds. Thus, the driver provides for the possible loss of a datagram. *The network need not ensure the delivery of a datagram.*

— An unavailable file system can be locally disabled in order to abort all pending and subsequent requests for the its data. This mechanism provides immediate feedback to a user who attempts to access files on a system that has crashed. The remote file system network software discovers which remote file systems are available by inspecting the routing table.

3. NETWORK SOFTWARE

3.1 Terminology

All network traffic is in the form of *datagrams*, where a datagram is an indivisible and complete request for the destination to perform a specific action with a given set of data. The destination of a datagram is defined to be a *virtual endpoint*, or *VEP*. Each processor hosts at least two VEPs: its name, and *here*, a pseudonym for its name. A module that actually consumes a datagram and performs some useful work is called a *server*.

3.2 Software Architecture

The architecture of the network software was influenced by the desire to keep each process simple. Each process reads some input data, performs the appropriate action, and writes out a response if necessary.

The architecture was constrained by the need to multiplex traffic arriving from other processors, the remote file system, and user programs into one datagram stream for the switch. Additionally, datagrams must be broken up for transmission via communications devices and reassembled at the destination.

After considering the design goals and constraints, an architecture was adopted that consists of separate processes to assemble and switch the datagrams, and a pseudo-device driver in the kernel to multiplex them. Incoming datagrams are reassembled by *listener processes*, multiplexed into one stream by the *datagram multiplexor* device driver, and switched out or serviced by the *switch process*. (See figure 2.)

3.2.1 The Listener Processes. Each incoming line has a communications link listener process that reassembles the datagrams and writes them onto the datagram multiplexor port dedicated to the datagram switch.

An additional listener process reads remote file system requests and encapsulates them into datagrams. The destination of these datagrams is *here*, and the action requested is to map the remote file system device number to a file system name.

3.2.2 The Datagram Multiplexor. The datagram multiplexor is a pseudo-device driver that supports a number of *ports*. Each port is a channel—group pair. Datagrams are written on a channel, and read from the corresponding group. No limit is placed on the number of processes that may have a channel open. Groups, however, are exclusive-use.

The multiplexor is designed specifically for datagrams, and is therefore not stream oriented. No more than 1024 bytes can be written at a time. One read returns the data from exactly one write.

Datagrams are queued first-come-first-served in the UNIX buffer cache. Stale datagrams migrate out to disk until they are needed.

Disk space is allocated from the swap area in one big chunk when the multiplexor is invoked for the first time. This prevents the datagram multiplexor from fragmenting the swap area.

3.2.3 The Switch Process. Each processor on the network has a switch process that reads datagrams from the multiplexor and either switches them out to another processor or services them. Each function performed by this process is realized in a separate module.

The *switch module* is responsible for transporting a datagram to the proper server on the desired processor. Thus it performs two functions: switching and local server dispatching. It includes the main loop of the switch process, which reads in a datagram and switches it.

The switch subroutine looks up the datagram's destination in the routing table. If the destination is not in the routing table, or if the datagram has been switched too many times, it is serviced as a *routing failure*. Otherwise, if the routing table indicates that the distance to the destination is nonzero, the datagram is written out on the outgoing link named in that entry. If the distance is zero, the datagram has arrived at its destination and the server dispatcher subroutine is called to invoke the appropriate server.

The server dispatcher subroutine employs a server table to map a datagram's *request type* to a server. Each entry in the table has a request type and the server's four entry points: initialization, periodic poke, service successful datagram, and service a datagram with a routing failure. If the request type starts with a slash, the remainder of the request type is taken as the name of a multiplexor port (channel) on which to write the datagram. This convention permits datagrams to be sent to servers in other processes.

Finally, the switch module includes the subroutines that invoke the initialization and poke portions of the servers. The remainder of the modules in the switch process are servers.

The *routing server* maintains the routing table. (See section 3.3 below.) This module also provides a subroutine to the other modules in the switch process that adds a local VEP to the routing table.

The *remote file system pseudo-device server* supports the remote file system on the node that originates disk requests. This module possesses the knowledge of the correspondence between pseudo-device numbers and file system names. There are several tasks that require these data.

— Pseudo-device numbers are translated to file system names for outgoing requests.

— A remote file system pseudo-device is enabled if and only if it appears as a VEP in the routing table.

— A *mount* process is spawned for every pseudo-device that is in the routing table but not already mounted.

This module also passes data blocks in incoming responses to the remote file system driver.

The *remote file system disk request server* reads the data blocks on the processor that actually possesses the specified disk. This is a three step process.

— The file system name in the request is translated to a disk device number via the mount table.

— The desired data block is then obtained by doing a seek and read on the appropriate disk block device.

— The data block is encapsulated in a response datagram which is switched out by recursively calling the switch subroutine.

Disk errors and routing failures are handled by manufacturing a response datagram with the appropriate error indication. This server also adds local file systems to the routing table as local VEPs. Non-unique file system names are prefixed with the name of the processor.

The *remote command* server permits users to run commands in remote processors. Commands are executed by invoking an instance of the command interpreter and passing it the contents of the datagram. The switch process does not wait for the command to terminate. For instance, if it is desired to copy a file *to* another processor, a datagram can be sent to it that will cause this server to execute a copy command (remember that all files are readable). Any command that confines its writing to a single file system can be run remotely if simply addressed to that file system's VEP. Most UNIX commands fit this model.

The *switch maintenance server* implements the following functions:

Debug Mode Toggle	All significant events are logged when the switch is in debug mode.
Log a Message	The contents of the datagram are logged.
Profile Toggle	The number of invocations and the percentage of CPU time spent for each subroutine can be measured with the UNIX profiling facility.
Dump	The data space of the switch is written out to a file that can be examined with the debugger. The switch process is not destroyed.
Log Measurements	Counts of various types of datagrams are logged.
Restart	This is used when a new version of the switch process is installed on a remote processor. The switch shuts down gracefully and reexecutes itself.

3.2.4 The Poke Process. The routing software must be directed to initiate a polling cycle every fifteen seconds. The poke process writes a datagram with destination *here* and request type *poke* on the multiplexor every fifteen seconds. These datagrams cause the switch to take the appropriate actions.

3.2.5 Remote File Request Scenario (The following discussion refers to figure 3.) Suppose a user process residing on processor "a" needs to read a file (e.g. "/b1/stuff") in file system "/b1" on processor "b". The user process makes an *open* system call which causes the UNIX file system software to search the root directory for "/b1". Upon locating the table entry for "/b1", the kernel discovers that a file system residing on pseudo-device *n* of the remote file system device is mounted there. The file system software then requests the first block of the mounted file system's root directory from the remote file system device driver. The following operations are the same for every remote file system request.

The device number and block number are inserted in a buffer header which is passed to the remote file system device driver. (The file system treats the remote file system device driver exactly as it would a disk driver.) The remote file system driver passes the request to the remote file system device listener process which encapsulates it into a datagram. The datagram's destination is set to "here", and the request type indicates that it is an unmapped remote file system request. The listener process sends the completed datagram to the switch process via the datagram multiplexor. The switch process and passes it to the remote file system request server, which looks up the device number in a table to discover the name of the file system ("/b1"). It then constructs a datagram whose destination is the file system name. The request type of the datagram indicates that it is a remote file system disk request. The datagram also contains the block number of the block to be read and the name of the originating processor.

The remote file system request server recursively calls the switch module to switch this new datagram. It looks up "/b1" in its routing table and determines that the datagram must be sent out on "/dev/comx/b". The switch passes the datagram to the appropriate communications device driver which sends it to processor "b".

On processor "b", the communications device driver passes the datagram to a communications listener process. This process ensures that the whole datagram is received and then sends it to

the switch via the datagram multiplexor. The switch reads the datagram, looks up "/b1" in its routing table, discovers that "/b1" is local, and passes the datagram to the remote file system disk server module. This server looks up "/b1" in a table to determine which physical disk volume contains the file system, and then reads the desired block of data from that disk. Finally, it constructs a new datagram whose destination is the name of the processor that originated the request ("a"), and whose request type directs the datagram to the remote file system device server. The datagram also contains the block number and the data.

Once again, the switch is recursively called to switch the datagram. It looks up "a" in its routing table and writes the datagram out on the appropriate link. However, this datagram includes a block of data and therefore exceeds the maximum size that may be transferred by the communications driver in one system call, The datagram must be sent in two pieces.

The communications link listener on processor "a" reassembles the datagram before sending it on to the switch through the datagram multiplexor. The switch reads it, discovers that "a" is local, and passes the datagram to the remote file system device server. The server passes the data back to the remote file system driver, along with the volume and block numbers. The remote file system device driver software fills the appropriate kernel buffer with the data and returns it to the file system software.

3.3 Routing

Our current configuration utilizes one time-division-multiplexed bus to interconnect all the processors. Although this fully-connected topology is optimal for our network, length limitations of the interprocessor bus have forced us to use other configurations in the past. The routing software is designed to send datagrams to their destinations via the shortest available path. The only topological constraint is that links be bidirectional.

Recall that the destination of a datagram is defined to be a *virtual endpoint* (VEP). The routing software treats the network as a set of nodes (processors), each node hosting one or more VEPs, and a set of undirected links between the nodes. VEPs are represented as sixteen byte character strings.

Each switch process has a *routing table* that contains an entry for each accessible VEP. An entry consists of three fields: the name of the VEP, the number of hops to the VEP (zero for a local VEP), and, for nonlocal VEPs, the name of the link used to send a datagram to the VEP.

Network applications use the routing software to disseminate knowledge about resources. The application software defines a local VEP for each resource it possesses on its own system. The routing software then enters the local VEP in the routing table and distributes the VEP to other nodes. Eventually each processor's routing table contains all the VEPs in the network. The application software reads the routing table to discover which resources are available on other systems.

For example, consider how the routing software is employed by the remote file system to distribute the names of the available file systems. The remote file system *disk request* server defines a local VEP for each locally mounted file system. The remote file system *pseudo-device* server then searches the routing table for non-local VEPs that are file system names. The resultant list of available file systems is used to enable, disable or mount the remote file system pseudo-devices as needed.

There are currently three classes of VEPs in use:

"here"	a special VEP which always refers to the local node,
processor names,	which are enumerated in a configuration file (e.g. "a"), and
file system names,	which start with a slash (e.g. "/a1", "/a", "/a/usr").

The strategy employed to maintain the routing table as VEPs are added, lost, or moved is the heart of the datagram network. Because a dead system is silent[3], the switch can not be explicitly informed of a VEP's disappearance. Instead, the switch assumes that any VEP that does not appear in the routing table is inaccessible.

Each node periodically rebuilds its routing table by polling its neighbors. Every fifteen seconds, each node sends a request for routing information to its neighbors. The routing table is used to avoid the expense of sending a request to a nonfunctioning system.[4] However, all neighbors *should* be polled; the state of each system changes, and dead systems eventually return to life. As a compromise, the routing software selects one of the nonfunctioning nodes for each polling cycle.

The routing table entries returned in the neighbors' responses are accumulated in a fresh copy of the routing table. At the start of a polling cycle, this copy is cleared and reinitialized with the local VEPs. At the end of the cycle, the routing table is discarded and replaced by this copy. An entry is added only if it would shorten the distance to the VEP (always true if the VEP is not already in the table), and if the distance would be less than or equal to the number of nodes in the network. The entry is incorporated by copying the VEPs name, setting the distance to the VEP to one more than the distance from the neighbor to the VEP, and setting the name of the outgoing link to the name of the link for the neighbor.

4. PERFORMANCE

Performance measurement is an ongoing effort. Execution time profiling has been instrumental in decreasing the CPU overhead of the switch process. Real time measurements suggest that remote file accesses are four times slower than local file accesses. The real time to send a datagram and get a response back has been measured at about 60 ms. The amount of CPU time used by the switch to process a datagram is about 12 ms., of which 3 ms. is spent in user-mode and 9 ms. in system-mode. We note that for a single 512 byte remote block to be retrieved, our implementation causes it to be copied six times. The amount of time spent copying the data, however, is not significant: about 860 μs. per copy for a total of 5.2 ms., or 8% of the *best case* retrieval time.

Utilization of the UNIX buffer cache would yield a significant improvement in remote file access performance. We are measuring cache characteristics in order to develop an algorithm that provides real-time invalidation of *stale* remote data blocks. Other potential performance improvements are: moving the switch or listener processes into the kernel, implementing a more sophisticated flow control policy for the communication device drivers (recall that UNIX device drivers typically deal with 512 byte blocks of data, and that datagrams can be up to 1024 bytes long), and providing more sophisticated routing (e.g., speed-dependent or congestion-dependent routing).

5. SOFTWARE DIMENSIONS

The remote file system driver consists of 400 lines of source code, and compiles to a 2K byte object module. The datagram multiplexor driver consists of 300 lines of source code, and compiles to a 1K byte object module. The user processes consist of 2100 lines of source code, and compile to 200K bytes of load modules.

---

3. Dead men tell no tales.

4. Certain communications device drivers may take up to two seconds to return an error.

6. CONCLUSIONS

A transparent remote file system can solve the problem of supporting a large user community on a UNIX time-sharing system. Also, such a system when run on a configuration with several processors (especially when a spare processor is available as a cold standby) can offer higher availability than a system that provides a UNIX Programmer's Workbench on a large mainframe. Of course, such a system could allocate its resources more flexibly, devoting the full power of a large mainframe to a single process if necessary. However, this flexibility is not as critical for a large software development project as a high level of *availability*, and this is achieved by our implementation of a transparent file system.

One disadvantage of using multiple processors is the lack of a single, system-wide clock. Our source-code database software can only tolerate a discrepancy of about one minute among all the processors' clocks. We currently use manual means (i.e. the system operators) to achieve synchronization.

Our implementation of a remote file system has sacrificed functionality for simplicity and ease of maintenance. The read-only nature of remote files has created a need for special-purpose tools to update them. But, the same limitation made it possible to implement the remote file system without any changes to kernel software and greatly simplified the task of porting the software from the DEC PDP-11/70 to the DEC VAX-11/780.

The simplicity of the software also helped us to get it up and running quickly. We have been providing remote file system service on a four processor configuration since October of 1979. The fail-soft nature of the system has proven itself over a few hardware failures and many software failures.

In the future, we will expand the system to 200 users on seven computers, make optimizations to improve its performance, and implement functional enhancements.

ACKNOWLEDGEMENTS

The first implementation of our remote file system was due, in part, to the efforts of D. Way Ting. Support, suggestions, and encouragement, all of which proved to be essential, came from K. G. Freeman. Special recognition is due C. B. Hergenhan, for the resource-sharing and file copying facilities for both this system and its predecessor. Finally, we gratefully acknowledge the technical contributions of G. L. Chesson, who persuaded us to separate the transport function from the remote file system and construct an underlying network.

REFERENCES

[1] D. M. Ritchie and K. Thompson. 'The UNIX Time-Sharing System,' *Bell Sys. Tech. J.*, 57, 6, 1905-1929 (1978).

[2] T. A. Dolotta, R. C. Haight, and J. R. Mashey. 'The Programmer's Workbench,' *Bell Sys. Tech. J.*, 57, 6, 2177-2200 (1978).

[3] D. A. Nowitz and M. E. Lesk. A Dial-Up Network of UNIX Systems. Bell Laboratories, January, 1979.

[4] A. L. Sabsevitz. Guide to IBM Remote Job Entry for PWB/UNIX Users. Bell Laboratories, October, 1977.

[5] K. Thompson. 'UNIX Implementation,' *Bell Sys. Tech. J.*, 57, 6, 2177-2200 (1978).

APPENDIX

Our users have found many ways to utilize this system. We describe some of them in this appendix.

Each user is assigned a home file system for permanent files. Each file system is normally mounted (locally) on a particular processor; thus a user customarily logs in on the processor with his/her home file system. Our system permits a user to position his/her working directory in any file system on any processor. Thus, our users can log in on *any* processor, and read their files. In fact, users have inadvertently logged on to the wrong processor only to discover that their files were unwritable.

For example, one of our processors lacks a tape drive. A user who wishes to copy his/her files to tape can log on to a processor with a tape drive, and access his/her files transparently.

One of our processors has a high speed connection to a high resolution graphics terminal. With this terminal, one can examine typeset text, graphs and figures before producing hard copy. As any user can log on to any processor, all users can take advantage of this unique terminal.

Without the remote file system, a user who needs to reference *libraries* must execute file transfer commands to copy them to his/her own machine. The actual copying is performed asynchronously, so some form of completion notice is required (typically, inter-system mail service). As the copying is time consuming, the user often tries to use previously copied versions (hoping that they are correct). The transparent access provided by the remote file system allows library sharing across many processors as if there were only a single processor.

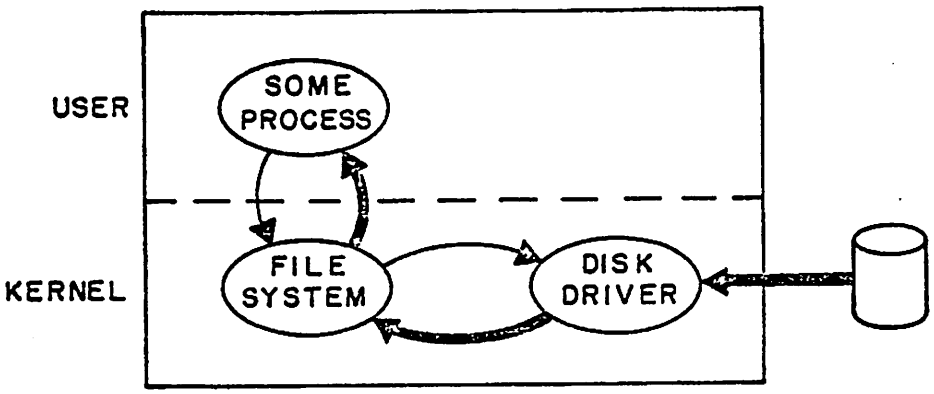

FIGURE 1a

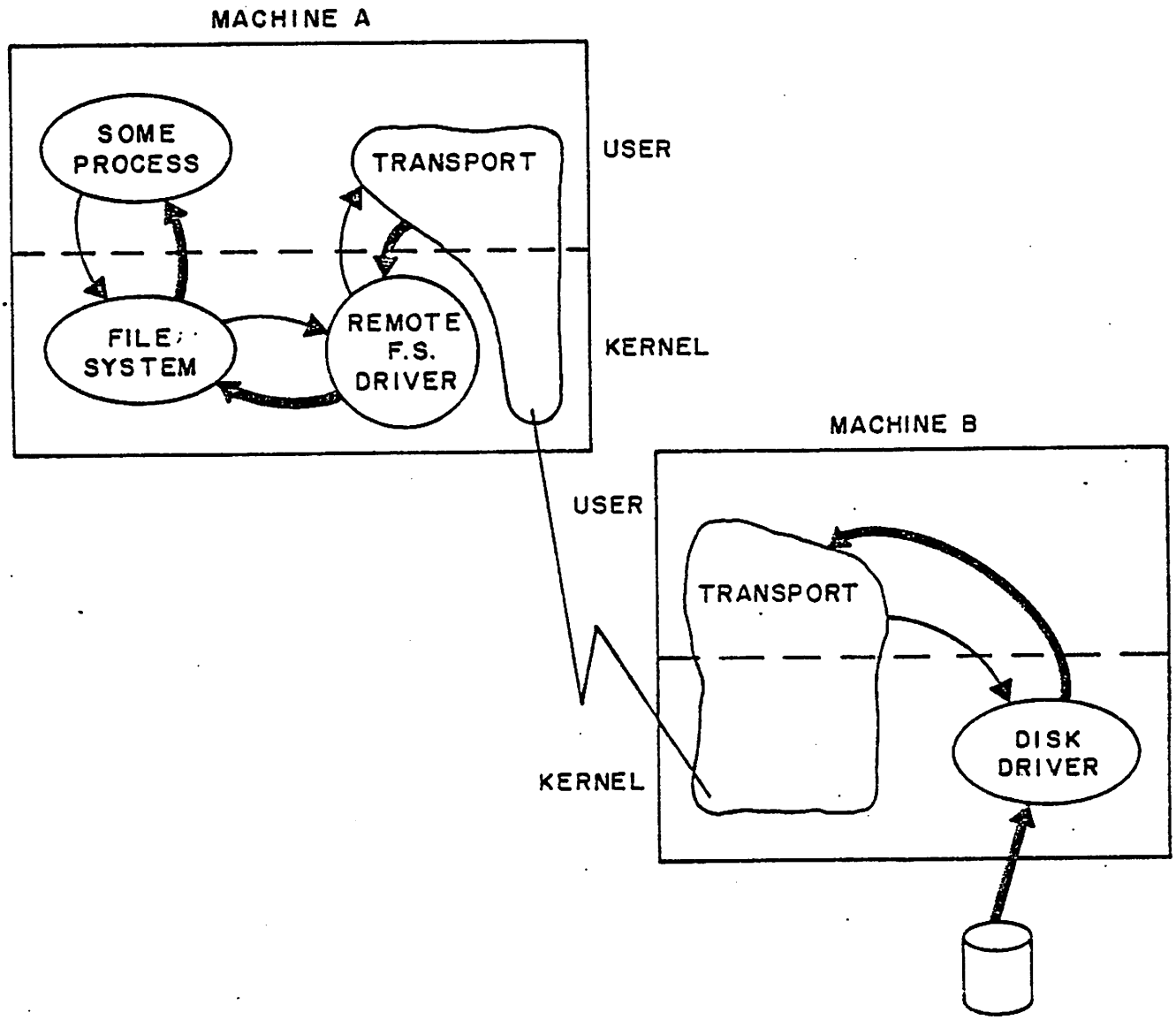

FIGURE 1b

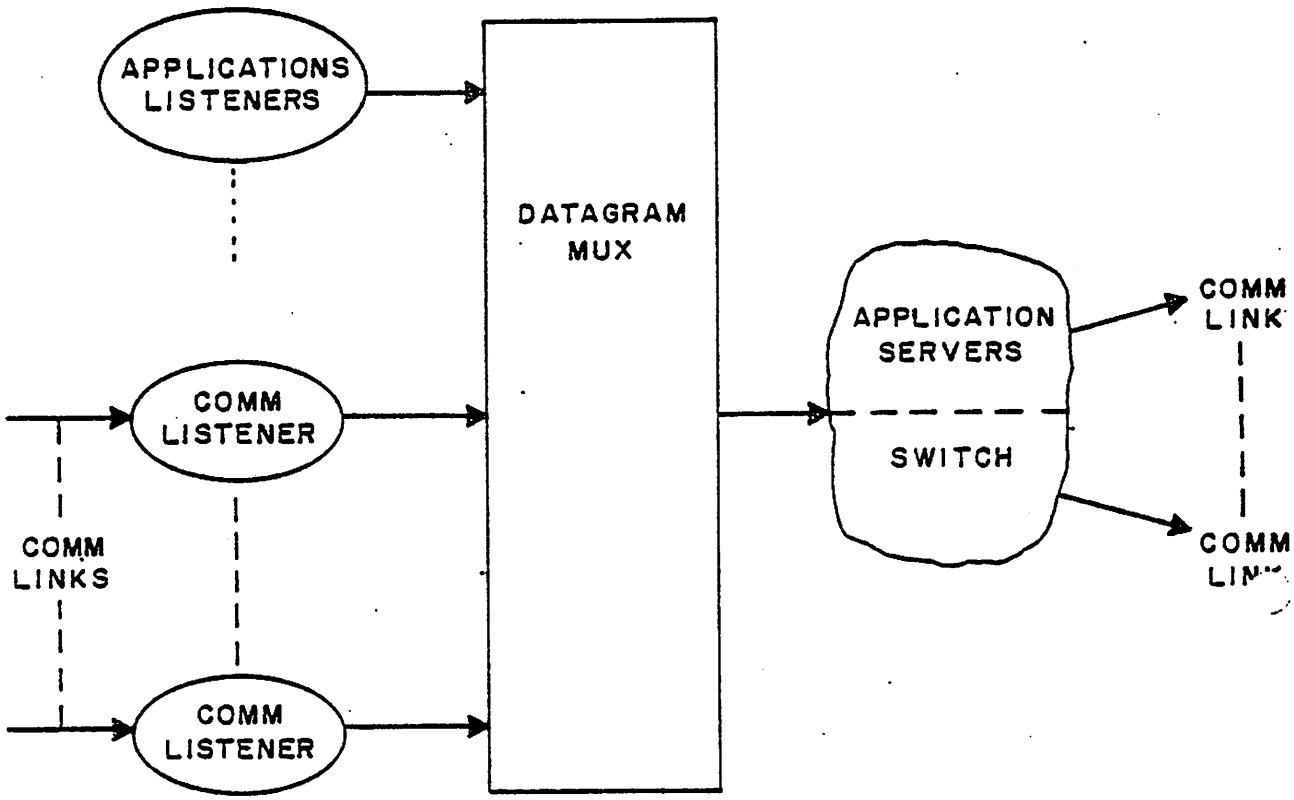
APPLICATIONS LISTENERS
COMM LISTENER
COMM LISTENER
COMM LINKS
DATAGRAM MUX
APPLICATION SERVERS
SWITCH
COMM LINK
COMM LIN

FIGURE 2

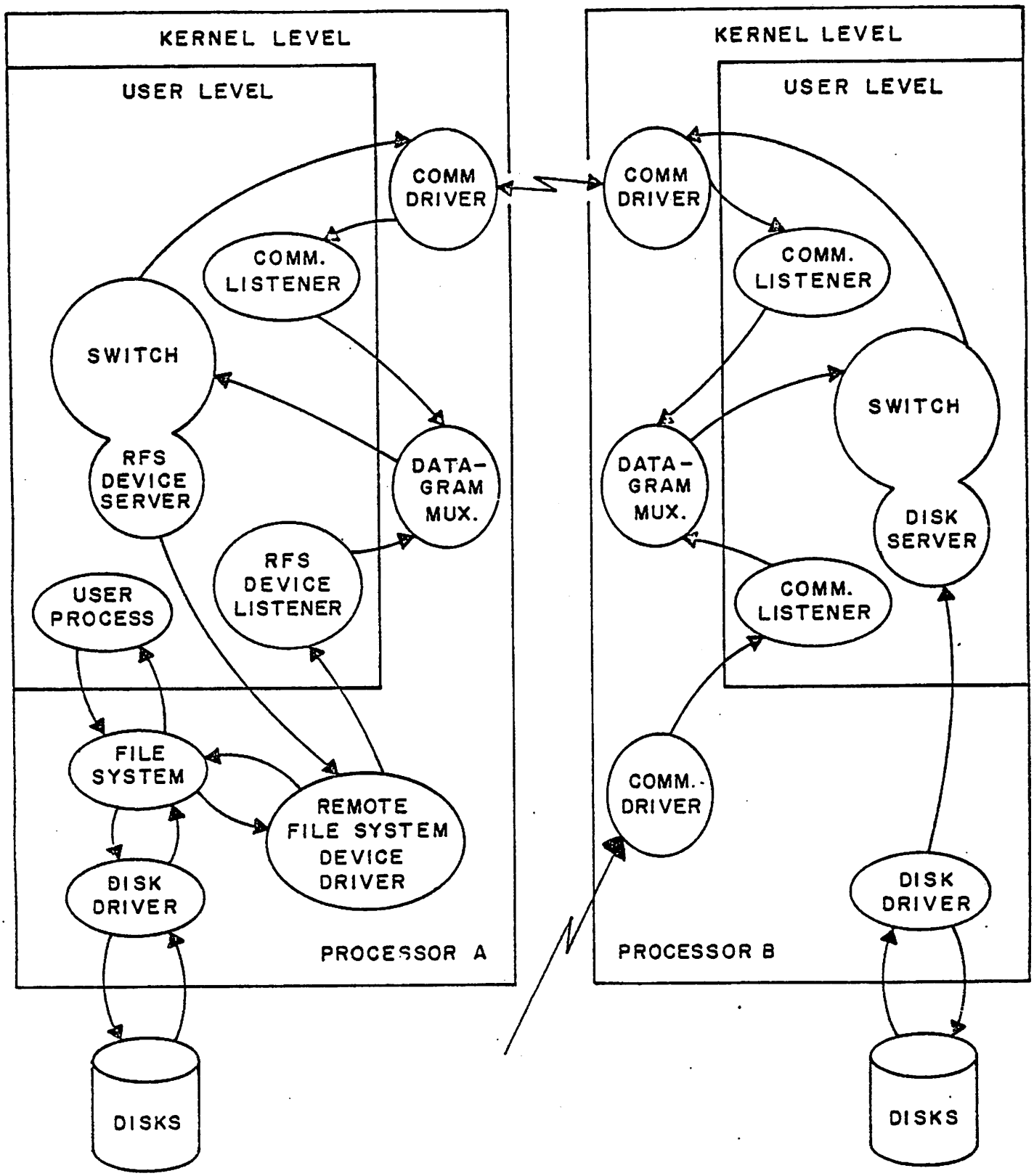
KERNEL LEVEL
USER LEVEL
COMM DRIVER
COMM. LISTENER
SWITCH
RFS DEVICE SERVER
DATA-GRAM MUX.
RFS DEVICE LISTENER
USER PROCESS
FILE SYSTEM
REMOTE FILE SYSTEM DEVICE DRIVER
DISK DRIVER
PROCESSOR A
DISKS
KERNEL LEVEL
USER LEVEL
COMM DRIVER
COMM. LISTENER
SWITCH
DATA-GRAM MUX.
DISK SERVER
COMM. LISTENER
COMM.-DRIVER
DISK DRIVER
PROCESSOR B
DISKS

FIGURE 3