A DISTRIBUTED UNIX SYSTEM BASED ON A VIRTUAL CIRCUIT SWITCH

G. W. R. Luderer,
H. Che, J. P. Haggerty
P. A. Kirslis †, W. T. Marshall

Bell Laboratories
Murray Hill, New Jersey 07974

† Current address: University of Illinois,
Urbana, Illinois 61801

ABSTRACT

The popular UNIX^{TM} operating system provides time-sharing service on a single computer. This paper reports on the design and implementation of a distributed UNIX system. The new operating system consists of two components: the S-UNIX subsystem provides a complete UNIX process environment enhanced by access to remote files; the F-UNIX subsystem is specialized to offer remote file service. A system can be configured out of many computers which operate either under the S-UNIX or the F-UNIX operating subsystem. The file servers together present the view of a single global file system. A single-service view is presented to any user terminal connected to one of the S-UNIX subsystems.

Computers communicate with each other through a high-bandwidth virtual circuit switch. Small front-end processors handle the data and control protocol for error and flow-controlled virtual circuits. Terminals may be connected directly to the computers or through the switch.

Operational since early 1980, the system has served as a vehicle to explore virtual circuit switching as the basis for distributed system design. The performance of the communication software has been a focus of our work. Performance measurement results are presented for user process level and operating system driver level data transfer rates, message exchange times, and system capacity benchmarks. The architecture offers reliability and modularly growable configurations. The communication service offered can serve as the foundation for different distributed architectures.

1. INTRODUCTION

The UNIX* time-sharing system is widely known and used [Ritchie 1974]. The virtues of distributed systems have been extolled in many places (for a comprehensive treatment see [Bochmann 1979], [Clark 1978], [Thurber 1979]). Thus, the idea of a distributed UNIX system has appealed to many.

* UNIX is a trademark of Bell Laboratories

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1981 ACM 0-89791-062-1-12/81-0160 $00.75

In spring of 1979, when we began to study possible designs, we saw several major short-term goals for a multicomputer UNIX system arrangement:

i. increased capacity, i.e. being able to give better service to more simultaneous users,
ii. modular growth, i.e. being able to add computers as the load increases,
iii. increased availability, i.e. computer failure should not cause system failure,
iv. faster recovery, in particular file system checking and repair.

Figure 1 shows our first configuration. Connected through a high-bandwidth switch are two kinds of computers. The processors in the top row run the user processes, while those in the bottom row implement a global file system. All the user files are handled by the file server computers.

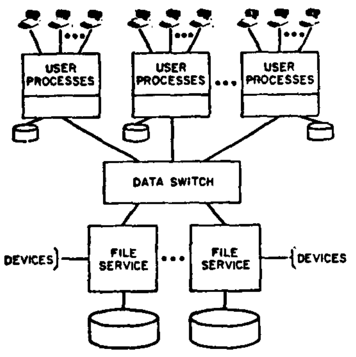

FIGURE 1
System with terminals on computers

Since the disks in the top row hold only local files, such as temporaries or the boot image, whose identities are of no interest to the user, the user processes can run on any of these computers. All files and devices that need be shared are on the bottom row computers. Here, the terminals are connected to the top row computers in a interleaved hunt sequence, which enables some primitive load balancing[1]. For growth, computers can be added in both the top and the bottom rows. For reliability, spares in both rows can prevent total service outage, and operator intervention can allow recovery from a degraded state.

1. This configuration does not allow direct communication between two terminals on different computers as offered by the *write* command.

Figure 2 shows a variation of our configuration with the terminals connected through the switch. It is more flexible and provides potential cost savings.

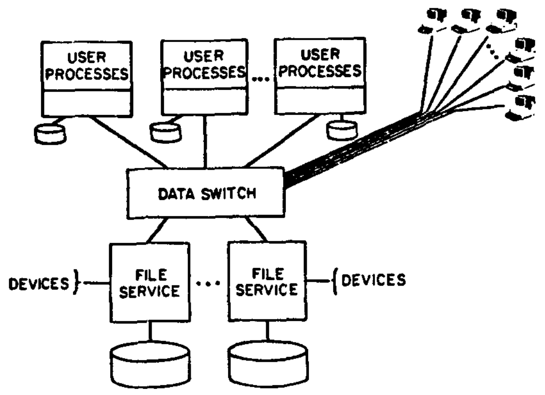

FIGURE 2
System with terminals on switch

The above two configurations model a computer center and not the geographically distributed system that many envision for the future. We feel, however, that our design takes a step in that direction, as shown in Figure 3. The same file servers appear at the bottom, but the top row consists of remote personal computers each serving a single user.

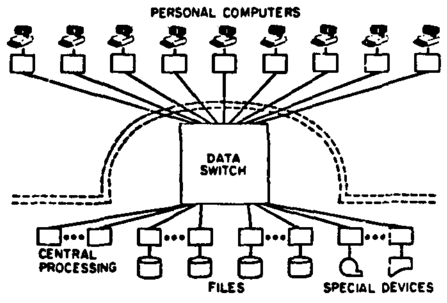

FIGURE 3
System with personal computers

The operating subsystems in the two types of computers are different and specialized, i.e. we have a heterogeneous network. We hope this results in a very important long-term benefit: reduced software complexity[2]. Not having to handle disparate tasks reduces component complexity; being able to replace functional components as user needs change and technology advances helps with the management of system complexity. Of course, the decomposition has to be "right," and the interfaces must be long-lived (like the system bus of a computer family).

S-UNIX is the name we have given to the specialized operating subsystem that runs user processes; it is "stripped" of most files, and models the later-to-be-achieved "single-user" UNIX system. The subsystem running on the file server is called F-UNIX.

The remainder of the paper has the following structure. Section 2 presents some general design considerations. We have separated the UNIX operating system related discussion from a discussion of the communication service subsystem. These topics are treated in Sections 3 and 4 respectively. Section 5 discusses performance, and Section 6 gives some ideas about possible future extensions. Finally, in Section 7 we compare our system with related efforts of others.

2. The emphasis here is on "long-term"; a substantiation follows later under "Potential Extensions".

2. GENERAL DESIGN CONSIDERATIONS

We gave ourselves the strict requirement of preserving the UNIX process environment and file system behavior, i.e. full compatibility with an existing version. We made as few changes to the UNIX code as possible, but we did make radical changes when they became necessary. The division between the file system and the rest of UNIX is not across a well-defined interface and required major redesign. It was also clear at the outset that the success of this project hinged upon efficient interprocessor communication software and hardware, and about half of our effort was applied towards this goal.

The obvious choice for the computer hardware was the high end of Digital Equipment Corporation's PDP-11 line. More unusual is our choice of the interconnection medium. Most contemporary designs of distributed systems use a packet switch and a message or datagram discipline for intercomputer communication. We wanted to explore the suitability of virtual circuits as the underlying communication architecture. One frequently encounters two objections to the use of circuit switches for distributed computer systems: First, fixed bandwidth allocation is particularly wasteful for the bursty kind of traffic common to data communication. Second, there seems to be more algorithmic complexity in programs that have to keep track of circuit states. (As one reviewer observed, "virtual circuits are always necessary, for reliability; the question often asked is at what level of the protocol hierarchy". We decided to push them to as low a level as possible, under the assumption (1) that they are generally useful; (2) that they require a lot of host resources to manage which can be more easily off-loaded to a peripheral if the circuit protocol is a low-level basic service.) One might say that there is intrinsic atomicity in pure message disciplines. We took the second argument as a challenge: if we could develop a workable and efficient communication architecture based on virtual circuits, we expected to see potential advantages in the areas of system management and extension to other services than data communication, e. g. voice, facsimile, etc. The first objection we could easily overcome. We found a switch that combines the desirable properties of both packet and virtual circuit switching: the Datakit switch [Fraser 1979]. It offers the functionality of a virtual circuit switch with dynamic bandwidth allocation, since it uses packet switching (demand multiplexing) in its internal implementation.

As to the second objection, we must leave it to the reader to judge whether the modest amount of added complexity is worth the advantages gained.

3. THE S-UNIX AND F-UNIX SUBSYSTEMS

The UNIX file system name space is a singly rooted tree, with intermediate nodes representing directories and with leaves representing files or devices.

Our initial intent was to remove all files from the S-UNIX side and put them on the F-UNIX side. We ended up keeping local files for four reasons:

a. An S-UNIX subsystem should be able to access more than one file server. To preserve a singly rooted name space tree with no name recognition in S-UNIX would require a file server hierarchy, which is undesirable because of reliability and performance.

b. It is impractical, at least for exploratory development, if one cannot bootstrap an S-UNIX subsystem from a local file.

c. There are potential efficiency gains if some frequently used files like load modules are kept locally.

d. The down-stream model of a personal UNIX system should have the option of local files.

Point (a) actually calls for a local name space, not local file space. For example, one could add a small name space management facility in the operating system. However, we decided to keep the root of the global file system tree on the S-UNIX side.

In the current UNIX system, the file space is extended by "mounting" a properly formated disk volume on top of an existing directory. We have expanded this concept by allowing the S-UNIX user to "mount a file server" in an analogous way. Whereas the existing mount procedure requires a special file representing a properly structured block device (e.g. a formated disk volume), our new procedure substitutes a device communicating with a file server, i.e., a circuit to the switch. Multiple mounts of both kinds can be active simultaneously.

Figure 4 shows the file name space of a configuration of two S-UNIX systems which have mounted F-UNIX file servers F-UNIX$_1$ and F-UNIX$_2$ on mount points F1 and F2, respectively, in their local name space.

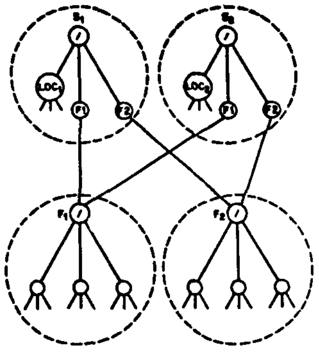

FIGURE 4
File name space of a 2/2 configuration

Processes on the S-UNIX side see no difference between local and remote files, except for performance. All system calls of the S-UNIX subsystem apply to both local and remote files. This includes special files, i.e. devices, on the F-UNIX subsystem. No F-UNIX system can access files of another F-UNIX nor can it get to local S-UNIX files.

Growth and recovery from failure can thus be handled by adding or removing S-UNIX and F-UNIX computers from in-service configurations, subject to appropriate operating procedures. Failure of an F-UNIX system can be handled by moving the disk volumes (or backed-up versions) to a spare computer and mounting that new file server on the failed F-UNIX machine's mount point in each S-UNIX. Failure of an S-UNIX machine disrupts all user sessions on that system, but the other systems remain unaffected.

File access control in the UNIX system depends on the user *id* and group *id*. For philosophical and pragmatic reasons, we have chosen to have a single password file on our systems; thus the *id* numbers denoting file ownership are global to all S-UNIX and F-UNIX machines.

By taking the position that the operating subsystems and the communication mechanism are trustworthy, we avoid having to deal with problems of authentication beyond those present in the current UNIX system. We realize that, once we allow remote "personal" computers, the local operating system cannot be trusted anymore. We shall return to this problem under "Potential Extensions".

3.1 S/F-UNIX Implementation

The following description requires an understanding of UNIX internals [Thompson 1978]. The reader unfamiliar with or uninterested in UNIX internals may skip to Section 4.

3.1.1 The Cut Between S-UNIX and F-UNIX Accessing files from multiple computers and preserving local files turned out to be conflicting objectives. There are two obvious ways of introducing remote files. The first is to have the remote file server look like a block-addressable device [Glasser 1980]. Because blocks contain housekeeping information and we wanted to preserve the shared file access properties of the UNIX system, we would have to introduce an inordinate amount of extra messages for locking and unlocking. The second way is to translate all remote-file-related system calls into appropriate messages. This is complicated because the operating system itself makes file system references, e.g. for core dumps, for writing the accounting file, and for loading programs. Our implementation followed this second choice closely. We introduced changes wherever the name-to-disk-address converter *(namei)* is invoked, to handle remoteness.

3.1.2 Remote Inodes In the UNIX system, each file on a volume is described by a data structure called an *inode*, which is read into memory when the file is opened. The *inode* contains almost all the information about a file, e.g. its type (directory, ordinary file, device, etc.), owner, access permissions, length, and physical address. We introduce an *inode* of type *remote* that is created in memory when a remote file is opened. It contains just enough information for the S-UNIX subsystem to talk about the file:

- a pointer to a data structure identifying the F-UNIX machine holding the file,
- a unique number assigned by that F-UNIX machine.

All other information about a remote file, e.g. its access permission and length, is maintained only by the remote file server. This allows all S-UNIX machines to see a consistent description of the file. The introduction of the remote inode enables us to restrict the number of messages exchanged to one request and one reply per call. The basic algorithm is: if a path name crosses the mount point of a remote file server, stop interpreting the path name and send a message with the remaining path name. If a remote file is being opened or created, the F-UNIX subsystem returns a tag of its choosing to be used in future references. Tags are also returned in response to *chroot* or *chdir* system calls. All absolute pathnames carry the root tag, and all relative pathnames carry the current directory tag. Thus the file server always sees the equivalent of an absolute pathname and does not have to remember the current directory; yet the tag (really an inode number) serves to speed up the search process.

In order to allow several S-UNIX machines to update the same remote file concurrently, the cache of disk blocks in S-UNIX memory had to be restricted to local files only. The F-UNIX subsystem, on the other hand, can use a large part of its memory as a cache, since it does not run user processes.

3.1.3 Special Files Devices are treated like special files in the UNIX system. Peripherals on the F-UNIX subsystem can thus be easily accessed in the usual manner. For example, an S-UNIX machine can write to a tape drive on an F-UNIX machine. Special peripherals like printers or phototypesetters could be handled by F-UNIX subsystems running on small dedicated computers with or without local secondary storage.

A new special UNIX interprocess communication mechanism is the *fifo*, which provides communication between unrelated processes by associating a new special file type with a file name. Since *remote fifos* are legal, they can be used for interprocessor communication between S-UNIX machines or between an S-UNIX machine and an F-UNIX machine.

3.1.4 File Server Details The file server computers are running under the F-UNIX operating subsystem. There is one file server process for each circuit connected to an S-UNIX machine. These processes execute in kernel mode. When started, they are connected to the circuit and obey the S-UNIX controlled file service protocol. starting with a "mount file server" request. Each S-UNIX machine is then handled by at least one server process on each F-UNIX. F-UNIX multitasking is simply implemented by starting several server processes per S-UNIX, each on a different circuit. The degree of multiplexing is thus chosen on the S-UNIX side, where as many requests can be outstanding as there are circuits to F-UNIX systems.

One design decision concerns the amount of S-UNIX state information to be kept in the F-UNIX subsystem. The file server does not keep a count of all open-operations against a file. Rather it keeps track of which S-UNIX machine has the file opened (at least once). Disappearing S-UNIX systems that do not properly close their files are discovered, and the files are closed.

3.1.5 File Service Protocol Interaction between both subsystems at the functional level is handled by the file service protocol, which is strictly a sequential message exchange over one virtual circuit. Error and flow control are supported by the circuit mechanism.

Out of 27 system call types related to files, 18 result in message traffic if remote files are involved. Of these, 10 contain a path name as an argument, and the remaining 8 refer to already opened files. Path names or the data read or written can be up to 64K bytes long. The structure of each message is a type code followed by type-dependent data.

4. COMMUNICATION SUBSYSTEM

The communication subsystem is built on the concept of a virtual circuit service. It is thus independent of the S/F-UNIX architecture and can be used as the foundation for different distributed system designs [Luderer 1981]. In the following, we shall introduce the Datakit switch, then explain how we use it, and finally give details about the protocols and the switch interface.

4.1 The Datakit Switch

Datakit is functionally a virtual circuit switch [Fraser 1979]. Computers and terminals are connected in a star topology to interface modules interconnected by a backplane. Packet switching occurs on the up-link and down-link of a folded bus on the backplane. The switch module at the pivot replaces packet source addresses with destination addresses (Figure 5).

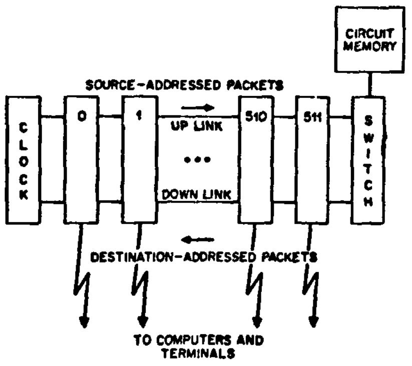

FIGURE 5
Datakit Switch

The aggregate data rate is 7 Mbit/s which corresponds to a payload of 42,000 16-data-byte packets per second. Due to asynchronous time-division multiplexing, it effectively provides dynamic bandwidth allocation on virtual circuits. The subscriber link interface is that of a Digital Equipment Corporation (DEC) DR11-C program-controlled, word-parallel communication device. We converted this interface to direct memory access with the help of a very fast (200ns instruction time) communication front-end processor, the DEC KMC11-B, which also handles our link protocol [Digital 1978]. The Datakit switch, in its largest configuration, can address a quarter million distinct subscriber circuits, which are usually partitioned into 511 physical subscriber links each multiplexed into a maximum of 511 full-duplex virtual circuits.

4.2 Connection Procedure

The switch contains a table that defines the end points of each circuit, i.e. the subscriber's interface module address and one of its virtual circuits. Circuit set-up and take-down are managed by a subscriber computer designated as Common Control. We have implemented a control program that resides in a DEC LSI-11 computer. This computer also holds a monitor program that in addition to other functions periodically receives status information from each subscriber interface module, e.g. a count of packets lost due to errors.

The switch is initialized such that the control program is connected to each bootstrapped subscriber's circuit 1, which is the signaling circuit for all circuit set-ups and take-downs. By convention, the subscriber manages only its odd-numbered circuits, starting with 3; Common Control owns and manages the subscriber's even-numbered circuits.

Common Control contains a simple name server that will establish circuits between any two subscribers. For example, when an F-UNIX computer is restarted, it announces to the name server that it is willing to accept file service requests on a specified service circuit. Likewise, when an S-UNIX computer is restarted, it selects an odd-numbered circuit and asks Common Control to connect it to the file server. Common Control will then allocate and set up an even-numbered circuit on the F-UNIX machine being called, which will be informed of the request and in turn acknowledge it to the requester.

The same mechanism is used to request other kinds of service. For example, small computers with no local secondary storage (e.g. PDP-11/23's) have a program in ROM that requests a circuit to a pre-established boot server, and a higher level protocol used on that circuit down-loads the image of the operating subsystem.

4.3 Circuit Protocol

A great deal of attention has been paid to the design of a simple and efficient circuit mechanism (we actually implemented four different designs). Major guidelines were to take advantage of the switch's hardware properties and to design a protocol that would fit into the front-end processor (4K bytes data, 8K bytes of code). The error behavior of the switch is characterized by very high reliability (we observed one faulty packet in six months) and the fact that the only possible error is loss of a packet (16 bytes). We call our protocol the NK protocol (network kernel). It is unusual in that it places the burden of error control solely on the transmitter site. This greatly simplifies the logic of the receiver, which has only one state and two local counters.

The NK protocol provides an error-free stream of bytes on a virtual circuit. The Datakit interface hardware expects and delivers 17-byte packets. The first byte identifies the virtual circuit. The format of the remaining 16 bytes is dictated by the NK protocol. The first byte is always a control byte followed by 0 to 15 data bytes. The control byte consists of a 3-bit command and a 5-bit argument. We will sketch the protocol by explaining the receiver action for the four commands it recognizes.

a. Initialization: The transmitter bids for a window size (i.e. number of buffers). The receiver accepts it or reduces it to its liking, and returns its decision.

b. Data packet with sequence number: If the number is expected, the packet is accepted and the local count is incremented (modulo window size). Otherwise the packet is flushed. No response is sent.

c. Data packet with sequence number and request for response: Same action as above, except that, in case of success, the sequence number is returned.

d. Enquiry request: A reply command with the last accepted sequence number is returned.

The above protocol has been working to our full satisfaction for data transfer on virtual circuits. However, it had to be modified to serve reliably for communicating with Common Control in the circuit set-up process. Here we operate with short messages only (one packet). The problem was survival of the system in case of failure of Common Control. A workable solution was found by letting Common Control delay the packet acknowledgment (case c above) until an acknowledgment of the reply was received. This causes a circuit set-up message to be re-sent repeatedly, even across crashes, until the required action is complete.

We have investigated the properties of this protocol to determine the choice of window size and acknowledge frequency as a function of the delay parameters of the network. Results will be reported elsewhere.

4.4 The Switch Interface

The driver on the host computer copies the data to be transmitted into a system buffer and gives a write command with a buffer pointer and the circuit number to the front-end processor, which empties the buffer and implements the above circuit protocol. Since buffer allocation is on a per-circuit basis on both ends, multiple circuit transmission is interleaved. For receiving, blocks are assembled in the front-end from packets and then copied into the waiting system buffer.

In order to further improve performance, we have built a peripheral to the front-end processor that eliminates the DR11-C and connects directly (i.e., not over the UNIBUS[3]) to the Datakit switch.

5. PERFORMANCE MEASUREMENTS

We distinguish three major dimensions when characterizing the performance of a computing system:

i. Capacity: what is the rate at which useful work can be performed?

ii. Responsiveness: what is the delay before a desired action takes place?

iii. Overhead: how many system resources are used to perform a specified activity?

For distributed systems, we characterize interprocessor communication by three measurements along these dimensions:

a. the achievable data transfer rate,

b. the time to exchange a null message,

c. the CPU time needed to perform communication.

The measurements given below refer to a configuration of two DEC PDP-11/45 computers[4] connected through the Datakit switch. Unless otherwise stated, the interface is a KMC11-B controlled special line card.

5.1 Data Transfer

Figure 6 shows data transfer rates between two processors as a function of the block size. The two solid curves show user level data transfer, i.e. process to process, and kernel level transfer, i.e. between specially instrumented drivers on each computer, respectively. The latter is more typical of the file server which runs in kernel mode. For comparison, the intracomputer pipe rate between user processes is shown as a dotted line. The data were generated within one computer and thrown away after arrival at the receiver.

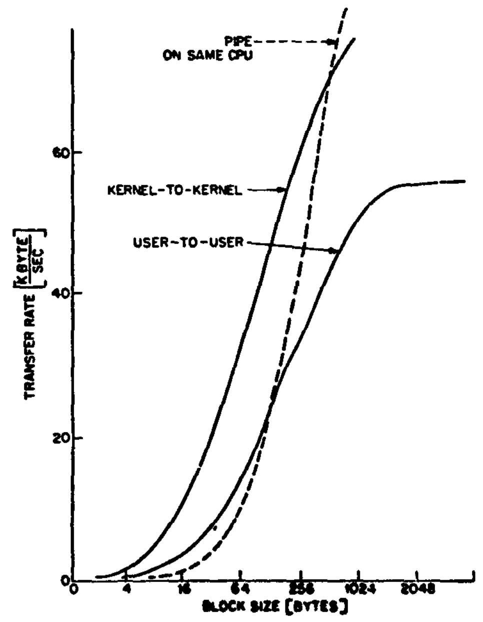

FIGURE 6
Data transfer rates between PDP11-45's

Figure 7 shows the CPU utilization of the host computer for the three experiments of Figure 6.

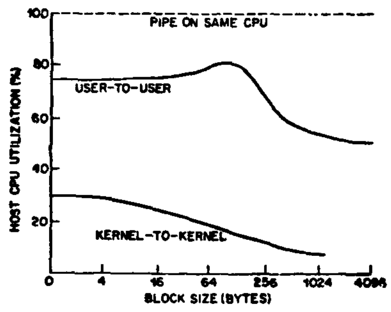

FIGURE 7
CPU Overhead for communication

As with many communication media used for local area networking, the Datakit switch is supposed to be operated in an area of light loading. Notice that, as the load increases, so does the throughput since there are no losses due to collision.

5.2 Responsiveness

A test script that invokes commands which use all of the remote file system functions was executed in two modes, and the total elapsed time and CPU resources used were measured. Figure 8 below shows the results. Notice that all programs had to be loaded from the remote file server. In both cases, CPU time contains 2 sec of user CPU time.

3. UNIBUS is a trademark of Digital Equipment Corporation
4. Neither computer had a cache memory. The core memory on the file server computer slowed it down by about 10% compared to the other computer, which had MOS memory.

Experiment script run with	*Elapsed Time*	*CPU Time (S-UNIX)*
local files only	39 sec	12 sec
remote files only	51 sec	16 sec

FIGURE 8
Response Time Test

Another measure of responsiveness is the elapsed time in which a short message can be sent and acknowledgment received. We measured 11 msec for an exchange between user processes on different processors.

However, at the kernel-to-kernel level we successfully sent a message in 900 μsec. We believe this result bodes well for new computer architectures with low context switch overhead and segment switching instead of copying.

5.3 Capacity

A multithread benchmark script was executed to determine system capacity. The set of application processes in the script is assumed to be a crude approximation of some realistic work load. A series of experiments were performed, increasing the number of scripts beyond saturation. As the script is executed many times, one can determine the maximum number of processes that terminate per hour as a relative measure of capacity.

Measurement	*Local Files*	*Remote Files*
sec/run	1699	1988
scripts completed	20	15
scripts/hour	43	28
sec/script		
real time	419 ± 22	651 ± 46
user time	28 ± 0.4	30 ± 0.5
system time	42 ± 1	69 ± 1
CPU utilization		
local	94%	89%
remote	0%	43%

FIGURE 9
Throughput Test

5.3.1 Virtual Circuit Set-Up We have measured the time for setting up a virtual circuit between two processes in different computers, i.e. the elapsed time from a process issuing a dial request until the called process has opened the new circuit. About half of this time is due to UNIX file system activity. Figure 10 shows the measurements for different computers serving as Common Control.

Common Control Computer	*Elapsed Time*
LSI-11	100 ms
PDP-11/23	62 ms
PDP-11/45	50 ms

FIGURE 10
Circuit Set-up Time

5.3.2 Performance Discussion Many researchers have found the actually achieved data transfer rates across unloaded high-bandwidth communication media to be disappointingly low due to the complexity of the communication software. At the outset, using DR11-C hardware, we observed user level transfer rates in the 10-15 kbyte/sec range with practically 100% host CPU utilization. Similar results have been reported by others using a variety of communication media. For example, going through several levels of protocol (some unnecessary), UNIX systems connected over the 6.25 Mbyte/sec Hyperchannel from Network Systems Corporation achieve an effective rate of 18 kbyte/sec [Goldsmith 1981][5]. Likewise, user level data transfer on a 370 kbyte/sec (2.94 Mbit/sec) Ethernet[6] has been observed as 10 kbyte/sec [Spector 1981], and as 50 kbyte/s on the 10 Mbit/s Ethernet between the much faster Dorado computers [Crane 80]. Even the highly optimized Tandem system with two 13 Mbyte/sec Dynabus connections yields only a rate of 65 kbyte/sec at a buffer size of 512 bytes. [Usas 1979].

In the light of these reported results we consider the performance of our current interface to be an advance in the design of efficient communication interfaces:

i. a user level transfer rate of 48 to 55 kbyte/sec at a host CPU utilization of 45 to 80% with a relatively slow CPU (11/45)[7],
ii. a kernel level transfer rate of 125 kbyte/sec,
iii. a kernel level message transfer time of 900 μs across the switching system.

With the same interface hardware (a 1 Mbit/sec DEC DMC-11), and the DDCMP network protocol, used as a machine-to-machine link, the network in use at Purdue has been able to achieve 31.25 kbyte/s between 2 PDP-11/70's.

System performance observations indicate that command execution times are increased by an average of 55% if remote files are accessed, showing a range of 37 to 75%. Throughput measurements show a decrease of about 35%. Distributing the load of one computer across two specialized computers does not double the capacity, at best we may achieve that two computers carry the load of one in our current architecture. (We have optimized only the lower communication protocol; the file service protocol carries, among other things, the overhead of accommodating computers with different data representations). An explanation for this phenomenon is that the transmission delays can be overlapped with local processing (hence no losses), but the off-loaded file service is compensated for by the communication load (hence no gains). A very promising result is the short circuit set-up time. Once switch interfaces supporting a larger number of virtual circuits become available, a more dynamic use of circuits can be made; e.g., one will be able to afford setting up circuits for a brief message interchange.

5. Hyperchannel is a trademark of Network Systems Corporation.
6. Ethernet is a trademark of Xerox Corporation.
7. The variation is due to different driver designs which allow some trade-off of transfer rate against CPU intensity.

We have conducted extensive communication performance measurements using both software and hardware monitoring techniques. Our results indicate that in order to achieve high transfer rates a fast front-end processor is essential. In our case, the speeds of the host and the front-end are 0.3 and 5 MIPS, respectively. Conversely, the degree of responsiveness is dominated by the speed of the host processor.

6. POTENTIAL EXTENSIONS

6.1 Computer Pool

UNIX process creation uses the "fork" system call. A process calling *fork* is duplicated: the child process shares the code and all the outside connections (open files) with the parent process. Both continue execution following the *fork* call, but each knows about its identity, and, of course, the parent recognizes the *fork* operation's success or failure. We propose to extend this mechanism by supplying an argument to *fork*, the name of the computer on which the child process is to start. The name could be a generic name asking for any free member of a pool of computers. The *fork* call would then result in a service request to Common Control which would forward it to an available server, an operation similar to a hunt sequence on a telephone switch. The servers in this pool are assumed to have announced their availability.

The computer pool could be used in a variety of ways. First, one could assign each user logging on from a terminal a "personal" computer for the duration of the session. The obvious advantages are: no sharing with other users and no dependence on one specific computer. Second, one could assign additional computers to a user as the need arises. For example, the UNIX command interpreter recognizes "&" at the end of a command as a request for background execution, which is implemented via *fork*. In our model, the background work could be executed on another computer. Further, heavily used commands currently resulting in a pipeline of so-called filter processes could run on several processors. However, there are several problems yet to be solved with this approach. First, one would need communication between S-UNIX subsystems, e.g. through pipes. Second, one would have to bequeath the user's identity to the forked-to pool processor. Third, a particular problem occurs due to the UNIX feature of "set-user-id", which allows processes to assume the privilege of acting on behalf of a user different from the invoker ("effective user" as opposed to "real user"). This set of problems would be greatly simplified if the S-UNIX subsystem were not allowed to have local files.

In a nutshell, the access control problem boils down to the following choice: In a multi-user S-UNIX architecture (our current implementation) the file server trusts the S-UNIX side, which is supplying an (presumably unforgeable) user id with each file open request, assuming that the authentication (login, password) has been valid. Privileged processes (set-user-id) supply the changed "effective user id" with each open request and can thus run on the S-UNIX subsystem. On the other hand, with a remote single-user S-UNIX, the file server has to remember the established "real user id" for each subsystem, and a privileged process could not run on the remote system, since there is no way to guarantee that its open requests do not come from an impostor process. One way out would be to restrict privileged processes to run on the file server, which is only possible if it does not require access to files on other S-UNIX or F-UNIX systems.

In summary, the extended *fork* mechanism in connection with a computer pool could be used to exploit obvious parallelism to give users the power of one or more dedicated computers.

6.2 Further Specialization of Subsystems

Redesign of the component subsystems would preserve the protocols but move towards more specialization. For example, the file server could be redesigned to achieve:

a. greater reliability,

b. higher performance,

c. faster recovery from failure,

d. less component complexity.

Specifically, we would redesign the housekeeping algorithms according to the concept of a "stable storage system." This would reduce the damage caused by file server crashes and shorten the recovery time. The latter would be a much welcome improvement, since significant time is currently spent in checking and repairing crashed file systems. The fact that the F-UNIX subsystem offers a full UNIX process environment is an advantage; all that is needed is a special message from the S-UNIX side to start a check and repair process. (Currently, the F-UNIX local operator console has to be used for this activity.)

For extreme reliability requirements, mirrored writing or incremental back-up could be implemented. There are several obvious avenues for performance improvement: a large disk buffer cache, improved in-memory and on-disk search techniques, lower overhead for task switching, contiguous storage and read-ahead for executable files, etc. Reduced complexity would result from restricting the file server to be a computer that handles one disk drive (including name management) and nothing else.

We see significant potential gains in performance and simplicity if the virtual circuit interface we are using could be further exploited by supporting a large number of virtual circuits on each computer, by offering short circuit set-up times, and by implementing the protocol up to the user process level in the hardware (front-end processor). Given these opportunities, we would map each open file into a circuit, which would let all the (de)multiplexing work be done in the front-end.

An architectural revision towards economy of mechanisms and less algorithmic complexity is to merge our concept of "mounting a file server" by extending the UNIX concept of mounting a file system to encompass remote file systems implemented as file servers.

7. RELATED WORK BY OTHERS

The literature on distributed systems is extensive and surveyed elsewhere [Bochmann 1979]. Issues of remote file service have been thoroughly discussed in [Sturgis 1980]. We shall therefore restrict ourselves here to UNIX-related efforts.

The idea of interconnecting several UNIX systems has been explored by several people. The most widely used file transfer utility is known as *uucp* (UNIX-to-UNIX Copy). It uses the dial-up network and spools explicitly named remote/local file transfer requests for asynchronous execution [Nowitz 1980].

The above scheme is the least transparent of a number of heterogeneous approaches, which all deal with a network of autonomous and more or less equal UNIX systems. To access remote files in addition to local ones, they have been augmented to also recognize remote file names. The remote files are either identified explicitly with a system prefix [Chesson 1975], [Lu 1979], [Antonelli 1980], or the names are integrated into a global name space [Glasser 1980]. The latter, however, provides only read access to remote files and uses user-level daemon processes for intercomputer communication.

Another approach is heterogeneous: computers are specialized to handle only a subset of the operating system functions. At the two ends of the scale are the satellite processor systems [Lycklama 1978], [Barak 1980], which off-load all system calls including file service to a mother system, and the file server in the Spider Network [Fraser 1974], which is a full-fledged UNIX system restricted to and modified for safe keeping of valuable files.

Virtual circuits on multiplexed direct links have been used in the Purdue network [Croft 1977]. Virtual circuits and the Datakit switch have been used to connect autonomous UNIX systems [Fraser 1975], [Chesson 1980]. The use of pipes and I/O redirection for connection to remote resources was proposed by

[Holmgren 1978], who has master and slave processes on each computer manage the interaction.

In the light of this prior work, our architecture combines the following attributes: a heterogeneous system with specialized computers, a global file system view with complete read/write transparency, full integration into the operating system kernel with no communication daemon processes, use of virtual circuits within the operating system, and an emphasis on performance that makes operation without local files feasible, since remote files can be accessed almost as fast as local ones.

8. CONCLUSION

We have built a distributed system with functionally specialized computers connected by a high-bandwidth virtual circuit switch. The two operating system components allow us to construct configurations with differing degrees of file system access and terminal-to-terminal communication. All configurations share the attribute of presenting a single global file system implemented by a varying number of file server computers. Local files accessible only from the home computer are optional. Terminals can be connected in several ways: to local hosts, through the switch, or through a front-end. Communication between users on different hosts is more restricted than in uniprocessor UNIX systems: terminal-to-terminal communication and interprocessor pipes are not implemented, but remote *fifos* can to some degree replace the latter.

Besides preserving as much of the UNIX capabilities as possible, we concentrated on performance issues. The performance degradation due to the separation of file service from user process handling has been made tolerable by an efficiently designed communication service with novel hardware and software. We have shown that communication based on virtual circuits can achieve high transfer rates, fast response, and low local overhead.

The resulting configurations offer modular growth and potentially more reliability. In spite of advances made in the efficiency of communication, the cost per user of such systems cannot yet compete with a multiplicity of smaller systems each serving smaller communities. A rough and safe estimate is that the CPU and memory resources ought to be doubled to achieve comparable capacity and responsiveness in an aggregate configuration.

However, we believe that with the cost of hardware steadily decreasing and with the efficiency of communication interfaces increasing, the extra costs of such architectures will become affordable when weighted against the additional advantages:

i. the ability to provide a single service to a larger community;

ii. the opportunity for modular growth with increasing load;

iii. less variability in the level of service;

iv. increased service availability;

v. faster recovery from failures;

vi. the potential to react faster to changes in user needs;

vii. the potential to introduce new technology in a less disruptive manner than in the past.

9. ACKNOWLEDGMENTS

A. G. Fraser and G. L. Chesson gave us access to their Datakit hardware and software [Chesson 1980]. E. Sirota from Brown University built the KMC/Datakit interface under the direction of R. C. Haight during a summer assignment. M. J. Bach converted the S-UNIX operating system to a more recent release. M. D. Beck conducted several of the measurements reported here, using facilities developed by J. Feder and D. A. De Graaf.

D. L. Bayer, R. H. Canaday, C. F. Simone, E. N. Pinson, J. M. Scanlon, and B. A. Tague deserve our thanks for encouraging and furthering this work.

Finally, this work would not have been possible without Ken Thompson and Dennis Ritchie having supplied us with a foundation to build upon: an efficient and understandable operating system, the UNIX system.

REFERENCES

[Antonelli 1980] C. J. Antonelli, L. S. Hamilton, P. M. Lu, J. J. Wallace, K. Yueh, "SDS/NET - An Interactive Distributed Operating System", *COMPCON Fall '80,* 21st IEEE Comp. Soc. Conference, pp. 487-493.

[Barak 1980] Amnon B. Barak, Amos Shapir, "UNIX with Satellite Processors", *Software Practice and Experience,* Vol.10, pp.383-392, May 1980.

[Bochmann 1979] G. von Bochmann, *Architecture of Distributed Computer Systems,* Lecture Notes in Computer Science #77, Springer-Verlag, New York, 1979.

[Chesson 1975] G. L. Chesson, "The Network UNIX System," *Operating Systems Review,* Vol. 9, No. 5 (1975), pp. 60-66. Also in *Proc. of the 5th Symposium on Operating Systems Principles.*

[Chesson 1980] G. L. Chesson, "Datakit Software Architecture", *Proc. ICC 79,* June 1979, Boston Ma., pp.20.2.1-20.2.5

[Clark 1978] D. Clark, "An Introduction to Local Area Networks," Proceedings IEEE, Vol.66, No.11, November 1978, pp.1497-1517.

[Crane 1980] R. C. Crane and E. A. Taft, "Practical Considerations in Ethernet Local Network Design", Hawaii International Conference on System Sciences, January 1980,

[Croft 80] W. J. Croft, "UNIX Networking at Purdue", UNIX Usenix Conference, University of Delaware, June 1980.

[Digital 1978] Digital Equipment Corporation, Maynard, Mass., *PDP11 Peripherals Handbook,* 1978, pp. 331-339.
—, "KMC11-B Unibus Microprocessor", YM-C093C-00, January 1979.
—, "COMM IOP-DUP Programming Manual", No. AA-5670A-TC.
—, "Terminals and Communications Handbook", 1979.

[Fraser 1974] A. G. Fraser, "Spider - an Experimental Data Communication System," *Proc. IEEE Conf. on Communications,* June 1974, pp. 21-30; IEEE Cat No. 74CH0859-9-CSCB.

[Fraser 1975] A. G. Fraser, "A Virtual Channel Network", *Datamation,* Vol.21, No.2; February 1975, pp. 51-58.

[Fraser 1979] A. G. Fraser, "Datakit - A Modular Network for Synchronous and Asynchronous Traffic", *Proc. ICC 1979,* June 1979, Boston, Ma., pp.20.1.1-20.1.3

[Glasser 1980] A. Glasser and D. M. Ungar, "A Distributed UNIX System", *Proc. of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks,* Feb. 3-5, 1980, p. 241

[Goldsmith 1981] S. Goldsmith, private communication, measurement at Bell Labs Murray Hill Comp Center

[Holmgren 1978] S. F. Holmgren, "Resource Sharing UNIX," *Seventeenth IEEE Computer Society International Conference,* Washington, D.C., September 5-8, 1978 (New York: IEEE, 1978), pp. 302-305.

[Kaufeld 1980] J. C. Kaufeld, D. Russell, "Distributed UNIX System", *Workshop on Fundamental Issues in Distributed Computing, acm SIGOPS and SIGPLAN,* Dec. 15-17, 1980, Fallbrook, Ca.

[Lu 1979] P. M. Lu, "A System for Resources Sharing in a Distributed Environment - RIDE", *Proc. IEEE Computer Society's Third International COMPSAC*, 1979.

[Luderer 1981] G. W. R. Luderer, H. Che, W. T. Marshall, "A Virtual Circuit Switch as the Basis for a Distributed System", Seventh Data Communications Symposium - 1981, ACM, IEEE Computer Society, IEEE Communications Society, October 27-29, 1981, (accepted paper).

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

[Nowitz 1980] D. A. Nowitz, M. E. Lesk, "Implementation of a Dial-up Network of UNIX Systems," *COMPCON Fall '80*, 21st IEEE Comp. Soc. Conference, pp. 483-486.

[Peterson 1979] James L. Peterson, "Notes on a Workshop on Distributed Computing", *ACM Operating Systems Review*, Vol. 13, No.3, July 1979, pp.18-30

[Ritchie 1974] D. M. Ritchie and K. L. Thompson, "The UNIX Time-sharing System," *Communications of the ACM*, Vol. 17, No. 7, July 1974, pp. 365-375.

[Spector 1981] A. L. Spector, Stanford University and Xerox Corporation, private communication

[Sturgis 1980] H. Sturgis, J. Mitchell, and J. Israel, "Issues in the Design and Use of a Distributed File System", *ACM Operating Systems Review*, Volume 14, Number 3, July 1980, p.55-69.

[Thompson 1978] K. Thompson, "UNIX Implementation", *Bell System Technical Journal*, Vol. 57, No. 6, Part 2 (July - August 1978), pp.1931-1946.

[Thurber 1979] Kenneth J. Thurber and Gerald M. Masson, *Distributed Processor Communication Architecture*, Lexington Books, D. C. Heath and Company, Lexington, Mass., 1979.

[Usas 1980] A. M. Usas, private communication of measurements done under Tandem's Guardian operating system at Bell Laboratories.