A Virtual Circuit Switch as the Basis for Distributed Systems

G. W. R. Luderer
H. Che
W. T. Marshall

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

A communication system is presented which consists of a switch (Datakit [Fraser 1979]) with associated control and interface hardware and software. The switch offers virtual circuit service which is internally implemented through packet switching. The users of the communication system are operating systems; using our communication system, we have implemented a network of UNIX™ systems and a new distributed operating system derived from UNIX. Our work concentrated on the performance of the subscriber-switch interface; on the reliability and recoverability of the switching service; and on the exploitation of the circuit concept for operating system design.

Introduction

The last years have seen many research efforts in local area networks and distributed systems (see e.g. [Thurber 1979] for an overview of current work; or [Shoch 1980] for an annotated bibliography.) Most software architects building such a system arrive at the conclusion that some message discipline (i. e. a higher level protocol) has to be defined. This emphasis on message exchange has, in many cases, led to the conclusion that the underlying data transport mechanism better be of the same kind, that is datagram service, sometimes also called packet switching in this context. We follow Pouzin's distinction between packet switching as a **service** offered to the user and packet switching as a transport **mechanism** employed at lower protocol levels [Pouzin 1977], [Pouzin 1978].

Our observation has been, that for applications we are familiar with, the message traffic is highly localized, and thus we should be able to carry it over virtual circuits. For example, a process in the UNIX[1] time-sharing system communicates with the outside world (files, peripherals, terminals or "pipes" to other processes) via an abstraction called "file descriptor", which handles an unformated byte stream. We think that the main objection to circuit switching as a transport mechanism stems from the observation that data traffic is much more bursty than voice traffic, and hence setting up a circuit implies reserving bandwidth in the communication medium, much of which will not be used most of the time. Fortunately, we had access to a data switch called Datakit [Fraser 1979] which provides virtual circuit switching service but implements the data transport internally by way of packet switching, i.e. no bandwidth is reserved, rather a contention scheme effectively offers dynamic bandwidth allocation. This switch allowed us to explore the development of network operating systems based on virtual circuit communication.

The Datakit switch leads to a star topology for the node. Although most other local area network research concerns more distributed topologies using busses, loops or rings, the virtues of a star topology have been pointed out in one [Closs 1980] of the two star proposals known to us (the other is [Rawson 1978]): the network interfaces are simpler, and the maintenance and fault isolation are easier.

Our study of other research efforts revealed further, that, no matter how high the bandwidth of the interconnection (including prior Datakit based architectures [Chesson 1979]), the computer interface to the switch or network was always the performance-limiting factor. It is not uncommon to

---

[1] UNIX is a trademark of Bell Laboratories.

CH1694-9/81/0000/0164$00.75 © 1981 IEEE

observe maximum data transfer rates that are two orders of magnitude lower than the idle network could carry. We therefore decided to concentrate our research effort on the network interface, and we were willing to forego, if necessary, the use of standardized protocols. Moreover, we set out to investigate the use of a fast front-end processor as the network interface.

The Datakit switch uses a control computer for circuit management.. Our second concern was to make this control as simple and reliable as possible and to deal with the problem of recovery from control failure.

The communication subsystem we built has been used in two different environments. First, we have been using it since early 1981 to interconnect a network of eight conventional autonomous UNIX systems in our laboratory[2]. Second, we built a distributed system with two kinds of specialized computers. These are computers with the conventional UNIX process environment running under the S-UNIX subsystem which obtain fast file service from computers running under the F-UNIX subsystem.

Our current research has several near-term goals: first, to connect terminals and dedicated work-stations to the network, second to ease connection of network components built by others including connection to other networks, and third to further improve the network interface by exploiting the virtual circuit concepts.

In the remainder of this paper, we shall first describe our architecture from a functional viewpoint. Then follows a description of the Datakit switch hardware. Our switch interface including the data and control protocol is next, and that is followed by a report on performance measurements and analysis. Finally, we present two operating system environments implemented on top of our communication subsystem and conclude with a perspective on future extensions.

Architecture

A high level view of our architecture is shown in Figure 1. It shows two nodes, A and B, connected by a trunk. Although we have so far restricted ourselves to a single node, Datakit-based systems with glass fiber trunks between nodes are in operation elsewhere at Bell Laboratories.

A node has a star topology. The switch control computer and up to 510 subscriber computers can be connected to the switch via dedicated subscriber links.

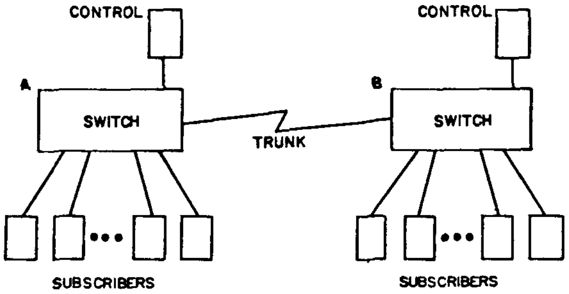

Figure 1. Network Architecture

Links are currently multiwire cables limited to the same room, but coaxial cable and glass fiber links for longer distances are being investigated. Each physical link can be multiplexed into up to 512 virtual circuits. Such a computer link can be replaced by several terminal connections. Hardware exists to connect four terminals in place of one computer link, although we have not used it yet. For the moment, view the switch as a backplane equipped with one interface module board per computer or trunk, or per four terminals. The address space of a single node is therefore 510 x 512 virtual circuit terminations, each ending in a computer or terminal. All circuits are full duplex.

The communication subsystem operates as follows: As each subscriber computer is bootstrapped, circuit 1 is automatically connected to Common Control [Chesson

---

[2] For a report on a network of UNIX systems using the same interface hardware as we did, but direct connections in place of a switch see [Croft 1980].

1979]. This is the signaling circuit. All the even-numbered circuits (2 through 510) of a subscriber are owned and managed by Common Control. Each subscriber owns and manages its odd-numbered circuits (3 through 511). Circuit 0 is for maintenance and circuit 1 for signaling.

Common Control holds the other end of each signaling circuit, one circuit per physical link. Signaling for circuit set-up proceeds as follows: The calling subscriber allocates one of its odd-numbered channels, say x, and sends a message to Common Control asking

connect circuit x to subscriber m

where m is the link address (one of the 511 interface module addresses). Common Control then informs the subscriber on link m over its signaling circuit 1 that it has a call on its circuit y, which is picked from one of the unused even-numbered circuits. If subscriber m has no more free even-numbered circuits, the caller would get a denial message on its circuit x. If the call is successful, the called subscriber is expected to reply on its circuit in whatever protocol the two parties want to use. The caller would thus receive the expected ready message from the called subscriber or a busy message from Common Control on its circuit x.

The above scheme could suffice only for intra-node traffic; moreover, the physical module address must be known to the caller. Therefore, call set-up can be by name rather than address. Common Control will help in the following way. A subscriber will allocate an odd-numbered circuit x and send an "available message" over its circuit 1 to Common Control:

accepting calls for name N on circuit z

Common Control will verify uniqueness of name N and enter the name into a table. Subsequently, somebody may call

connect circuit x to name N

and Common Control will try to allocate one of N's circuits and inform N on its circuit z that it has a call on the allocated channel, and so on as above. If the service is provided on another node, Common Control will connect to a trunk circuit leading to the service-providing node.

The important point here is that this scheme allows a **name server** hierarchy. The level 0 name service provided by Common Control is very simple and need never be modified (we restrict such "names" to one byte number). The application builder can implement an application-specific name server and announce it to Common Control in the above fashion. Calling parties would then as a first step ask to be connected to this name server which could use more elaborate schemes to map a name into an address. For example, the name server could subsume some scheduling function. Imagine, e.g., a caller asking for service of type S. The name server could manage several servers of type S and return the address of a free server. Notice that this is analogous to providing a hunt sequence in the telephone system. Other services modeled after the modern telephone system could be implemented, e.g., call forwarding to another node or a "call waiting" signal message to allow flexible priority scheduling. The important point is that such features can be put on top of our communication subsystem.

Once a circuit has been set up, we guarantee error-free transmission of a byte stream. The message protocol superimposed on the byte-stream transmitted through the circuit is entirely up to the conventions established between the communicating parties. If a circuit is broken, due to one party becoming inoperative, the switch discovers this fact and informs the live party on its signaling circuit.

Having explained the functional architecture of our communication subsystem, we shall now turn to a description of the Datakit hardware.

The Datakit Switch

The Datakit switch [Fraser 1979] consists of one or more interconnected backplanes holding interface module cards into which the subscriber link cables are plugged. Besides signaling and power supply connections, the backplane provides access to a folded serial bus, as shown schematically in Figure 2.

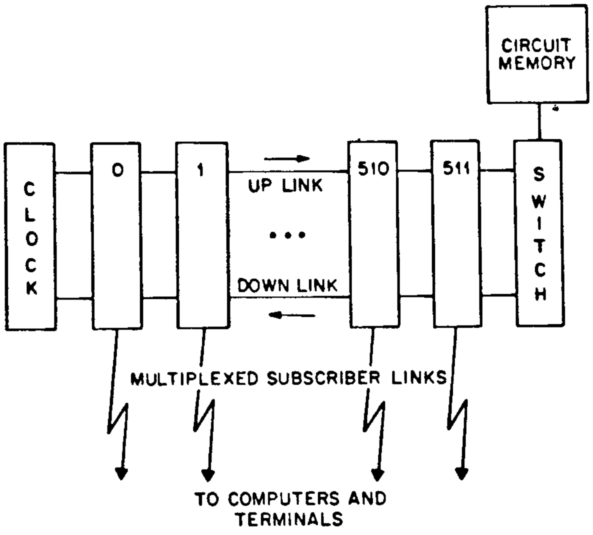

Figure 2. Datakit Switch

Each module connects to both the up-link and the down-link of the serial bus. A switch module sits at the pivot between the two links. A clock module terminates the bus. It runs at 7.5 MHz and defines a 180-bit packet cycle of 24μs. At the start of a cycle, modules contend for the up-link by putting their address on the bus and monitoring the result. One of the contending modules always wins and the others, if any, defer. Thus, module addresses determine a priority order[3]. The winning module uses the packet cycle to send a packet on the up-link to the switch module. As shown in Figure 3, the packets arriving on the up-link at the switch module, carry the **source address** in their header. The switch module does a double-index table look-up in its circuit memory, mapping the source address into a destination address during one packet cycle.

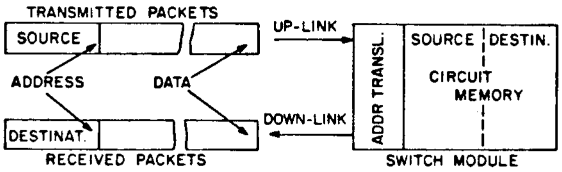

Figure 3. Packet Address Transformation during Transmission

As the packet is shifted out of the switch module onto the down-link of the bus, its source address is replaced by the destination address. All modules monitor the down-link and take in packets addressed to them. Up-link and down-link transmissions overlap, and the switch can handle about 42,000 packets per second.

The packet format on the Datakit bus is shown in Figure 4. There are 18 ten-bit envelopes. The tenth bit in each envelope is used for even parity checking. Addresses consist of two components, which are carried as nine-bit values in the first two envelopes: a module address and a virtual circuit number (within the module). Since the module address is redundant inside the interface module, it is automatically appended to outgoing and stripped from incoming packets. The remaining 16 envelopes carry data or control (signaling) information, with bit 9 determining whether it is a control or data envelope.

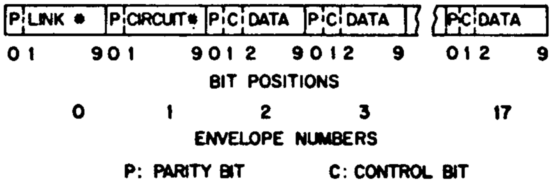

Figure 4. Packet Format on Datakit Bus

Thus, the payload at this level is 16 eight-bit bytes. (Our link protocol uses one of these

---

[3] There are several easy ways approximate to a more equitable round-robin-like scheduling. For example, each access circuit could maintain a "justice" flip-flop, which would be set if the last contention cycle is lost and which would control whether the link is contending or deferring. In case of contention, its setting would be prepended to the address.

bytes for error and flow control, as explained later).

The switch module recognizes specially formated control packets for reading or writing its circuit memory. A control program in one of the subscriber computers accepts circuit set-up and take-down requests and manages the circuit memory.

All Datakit modules perform certain maintenance functions. The clock polls each module periodically for status information: module type and serial number, parity errors, enable-disable state, etc. These packets are sent to a special circuit to which a monitor program can be connected. The current monitor program collects error statistics, and it can be used to interrogate and "manually" manipulate the switch state, an invaluable feature for debugging.

Interface Modules on the Switch

The Datakit-computer interface module matches the Digital Equipment Corporation (DEC) DR11-C parallel interface [Digital 1978]. It provides 16 bit input and output registers with additional status information which is used to read or write 10 bit envelopes. The module buffers up to 16 packets coming from the subscriber and up to 4 packets going to the subscriber from the switch. The subscriber can poll the status of the input (to the switch) memory to avoid overflowing it. There is no flow control at the output side, as this task is entirely left to the subscriber. If more packets arrive before the subscriber has emptied the output buffer, output is lost. Packets with parity errors are discarded.

There exists a terminal interface board which contains an INTEL 8085 microprocessor and interfaces to four terminals via RS232C connectors. Modems may be used or direct connections at 9600 baud. There is a also a trunk interface board for twisted pairs or glass fiber cables to interconnect Datakit switches. We are planning to use these boards later in 1981; they have been in use elsewhere at Bell Laboratories for some time.

Switch Interface on the Host

The Datakit switch has been in use at Bell Laboratories for connection of autonomous computer systems running under the UNIX operating system [Chesson 1979]. The computers have been connected via the DR11-C parallel interface which is a program-controlled I/O device (non-DMA). Its performance, though tolerable for the application, was of concern to us when we decided to investigate the design of a distributed UNIX system with more tightly coupled computers. Performance measurements revealed a data transfer rate in the 10 to 20 Kbyte/sec range with close to 100% CPU utilization of the host. Standard DMA devices like the DEC DR11-B could not be used due to too low intelligence for dealing with issues of packetizing, timing, and protocols. We had prior experience [Long 1981] with a DEC front-end processor, the KMC11-B, a specialized UNIBUS[4] device with a 200ns instruction time, 8 Kbytes of program memory and 4 Kbytes of data memory [Digital 1978]. Our intent was to convert the DR11-C program-controlled interface into a DMA interface using the KMC front-end processor. We went through four design versions with extensive measurement, analysis and modeling efforts (see below). Figure 5 shows the hardware evolution of our interface. Starting with the non-DMA interface (a), we next used the KMC to drive the DR11-C interface (b) and ended up building a special peripheral to the KMC processor, called KDI, which looks like a DR11-C to the Datakit switch (c).

---

[4] UNIBUS and PDP are a trademarks of the Digital Equipment Corporation.

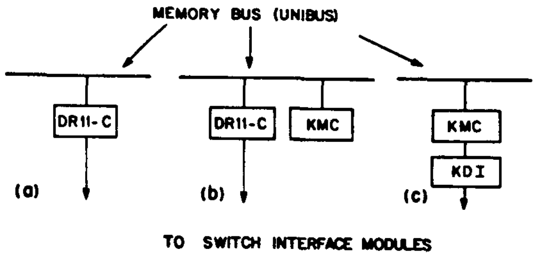

Figure 5. Evolution of Interface between Switch and Computer

Link Protocol

We assumed that any operating system would set up a virtual circuit and then ask for for either transmission of a block of data or reception in a supplied buffer. Our communication subsystem therefore had to provide these capabilities. In terms of a protocol, that means supplying circuit set-up and take-down and a link protocol to handle error and flow control between the subscriber computer and the switch-sided interface.

Our measurement and analysis of earlier implementations led us to the following insights.

i. The receiver part of the transmission job tended to be substantially more complex and time-consuming than the transmission part.

ii. The error behavior of the transmission medium is very low. (We have observed one lost packet in six months.)

iii. The flow control (buffer management) ought to be of paramount concern.

The Network Kernel (NK) protocol we designed takes these considerations into account; in addition the limited code memory in the front-end processor was another inducement towards keeping the protocol simple.

To the operating system, the NK protocol provides an error-free stream of bytes. To the switch interface, it communicates via packets of 17 envelopes (Figure 6). Since the Datakit switch checks for errors and discards bad packets, there is no need for a checksum (or CRC) in the packet.

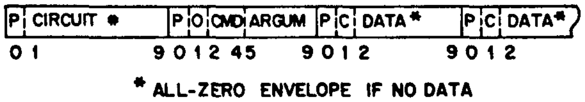

Figure 6. Packet Format of NK Protocol

Envelope 1 carries the 9-bit circuit number. Envelope 2 holds control information: a 3-bit command code and a 5-bit argument. Envelope 3 through 17 carry up to fifteen 8-bit data bytes. Absence of data is indicated by an all-zero envelope. We shall now describe the protocol algorithm.

1 The transmitter sends an initializing command offering a window size, i.e., asking the receiver to reserve a certain number of packet-size buffers.

2 The receiver responds with a buffer size that is equal or less than the offered one and resets its sequence counter. The transmitter accepts the offer.

3.1 The transmitter may now send a data packet with a sequence number as command argument.

3.2 Alternatively, the transmitter may send a data packet with a sequence number and, in addition, ask for a response.

3.3 Alternatively, the transmitter may just ask for a response without sending data.

4.1 In case 3.1, the receiver accepts the data packet only if its counter agrees with the supplied sequence counter modulo window size, which is subsequently incremented. Otherwise, the packet is flushed. There is no reply.

4.2 In case 3.2, the receiver acts as in case 3.1, except that the sequence number of the packet is returned, if the packet is accepted.

4.3 As response to case 3.3, the receiver just returns the sequence number of the last successfully received packet.

Notice that the receiver has only one state; it handles two registers: window size and sequence count, and it responds to only three types of commands. All time-out processing is left to the transmitter, to the extent that the receiver even won't answer to repeated enquiries of type 3.3 if it is busy.

Control Protocol

When this protocol was used to transmit a circuit set-up request or any other control message (a one-packet message) to Common Control, the following situation could arise. The packet is successfully transmitted and acknowledged, but thereafter the Common Control computer could fail. Although the message was received, the required service will not be carried out even after the Control computer is restarted. We therefore varied the receiver in the Common Control in the following way. Control delays the acknowledgement of a control request packet until after it has replied to the request. Only then will the request packet be acknowledged. The caller will repeat its control request across control failure events.

Host-Front-End Interface

One of the critical design issues turned out to be the separation of functions between the host CPU and the front-end processor. We experimented with various implementations, including the one used by the vendor-supplied software. Our final implementation achieved almost a 50% throughput improvement. There are two circular buffer queues:

i. The *command queue* receives commands from the host destined for the front-end.

ii. The *status queue* receives status information from the front-end destined for the host.

The head and tail pointers of the queues are addresses common to both host and CPU (control registers). Notice that the flow of communication guarantees that head or tail pointer updating is the exclusive responsibility of either host or front-end. Pointers to buffers for open read and write requests are kept in the front-end.

The above actions take place when the host and front-end are in the *operational* state. There are three states: *idle* when powered up, *initializing,* which is entered under command from the host, followed by *operational,* reported to the host after successful initialization, which includes acquiring the above two circular queues.

Performance Definitions

We think that architects designing a local network or distributed system should define performance goals for their system and moreover, they ought to report performance measurements of their implementation. We know of no standards of performance and have tried to define some experiments that could be reproduced by most systems. We assume an idle network with two communicating subscribers. The performance measures we suggest are the following:

1. Data transfer rate: measuring the error-free transfer of a large enough amount of data to achieve steady state and avoid boundary effects; e.g. bytes/sec to transfer 100,000 bytes from main memory across the network to main memory.
2. Message turn-around times: sending a short (say 1 byte) message and receiving a reply message of the same length; measured typically in msec.
3. Host CPU use: subscriber CPU time per message exchange and per transferred byte (instruction counts may be more descriptive but are

usually harder to measure). Alternatively, the percentage of the CPU used during data transfer may be reported.

4. Circuit set-up time: (this measurement applies only to systems using virtual circuits) the time it takes to establish a virtual circuit between two subscribers.

It is evident that these measures depend on both hardware and software. We found that the software enters the picture in two ways. First there is a network component determined by the architecture of the communication subsystem. Second, there is an operating system component which depends on the structure of the user environment offered within the subscriber. As the operating system using the communication subsystem is changed, the network component typically remains unaffected. We are therefore reporting all our measurements at both the **user level** which includes the operating system overhead, and at the **kernel level,** which does not.

This is significant in several ways. If an **existing** operating system is augmented to provide communication facilities, it may in most cases not be optimized for this purpose. (This was true in the two cases reported later). On the other hand, **new** system architectures may be designed with efficient and easy communication in mind. The kernel level characterization is more typical for such tasks as down-line loading of programs or whole operating subsystems (remote boot-strapping).

Performance Goals

We did not know what performance goals would be achievable when we started our work. As a general objective, we took the file system of a uniprocessor time-sharing system (UNIX) as a reference point.

The analog of the data transfer rate across the network is the maximum sustained rate at which the file system can supply or absorb data. A UNIX system on a DEC PDP-11/45 can attain about 25 Kbytes/sec in an ad-hoc experiment; the CPU utilization is then about 50%. This equates to 10ms of CPU time per 512-byte block. To get a short item from the disk (16 ms rotation), provided one knows where it is, thus takes in the order of 20 ms. Such an action may be likened to a message exchange. Finally, a good counterpart to the circuit set-up time is opening a file, which takes upwards from 40 ms depending on the depth of the name search. In summary, our initial performance goals can be stated as follows:

1. a data transfer rate on the order of 25-50 Kbytes/sec,
2. a host utilization of not more than 50% at the above rate or an overhead of about 10 msec per block transfer,
3. a message exchange rate of about 20 ms.
4. a circuit set-up time of about 50 ms.

At the kernel level, we have exceeded all four objectives. At the user level, they have been reached except that we can attain 50 Kbyte/sec at a CPU utilization of about 75%.

Experimental Conditions

All measurements were made between two DEC PDP-11/45 computers which are of the 0.3 MIPS category. One computer is approximately 10% slower because of core memory (the other's is of the semiconductor type). Both computers are connected via KMC 11-B front-end processors and specially designed line cards (KDI) to a single-backplane Datakit switch, except in those measurements where we compared the interface to a program-controlled DR11-C or a KMC 11-B/DR11-C combination. CPU time was measured by using a "soaker" program, that runs at low priority and reports its CPU usage based on a 16ms cycle system clock [Feder 1980].

For more accurate time measurements, we instrumented the systems by planting write-operations to a (another) DR11-C device at strategic points in the code. A multi-channel oscilloscope or logical analyzer provided a make-shift hardware monitor to graphically display event sequences of interest.

It is perhaps worth reporting, that these measurements and efforts to derive a model from the observations led to a series of code corrections and improvements.

Measurement Results

Figure 7 shows the intercomputer data transfer rate at the user level, i.e. a user process on one computer writing data to a process on another computer who discards them. For comparison, two intracomputer transfer measurements have been included. One is the UNIX pipe, which is (on an otherwise idle system) an in-memory transfer between two processes. The other experiment writes data into a local file, i.e. onto an RP06 disk (3600 rpm, 809 Kbyte/s transfer rate). The disk as well as the network curve flatten out beyond the block size of 512 bytes which is not only the native disk sector size but also the system buffer size used by network transfers.

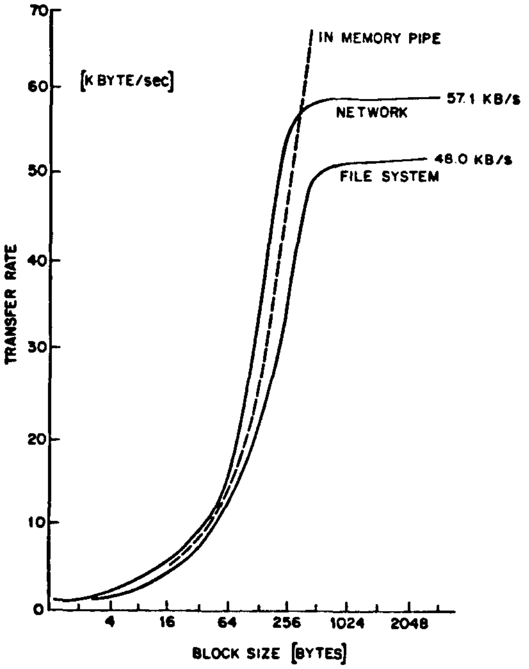

Figure 7. User-Level Data Transfer Rates

Figure 8 shows the data rate as a function of the block size at the kernel level for the three different interface configurations of Figure 5.

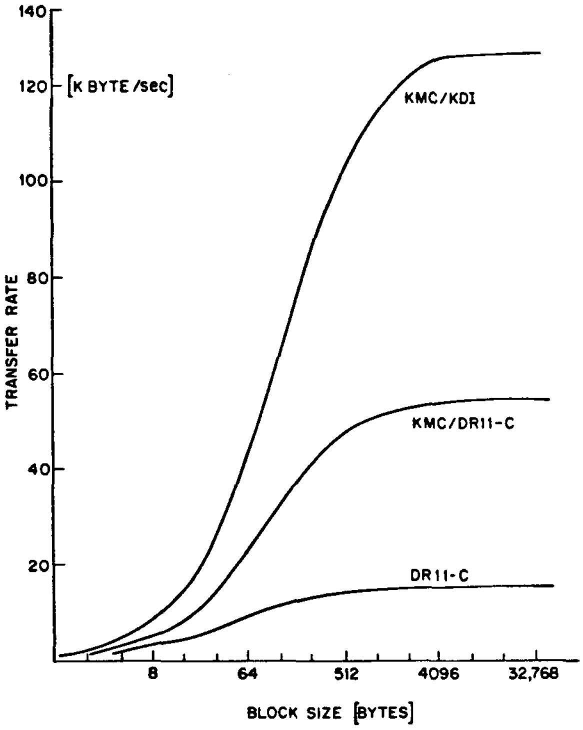

Figure 8. Kernel-Level Data Transfer Rates

The flattening out of the kernel-level KDI curve at 127 Kbyte/s (circa 1 Mbit/s) is due to the front-end being fully busy. As shown below, the front-end needs 116μs for each packet with a payload of 15 bytes).

Let us now turn to the host CPU resource usage. As an integral raw measurement, Figure 9 shows the host CPU utilization at the kernel level for the experiments of Figure 8.

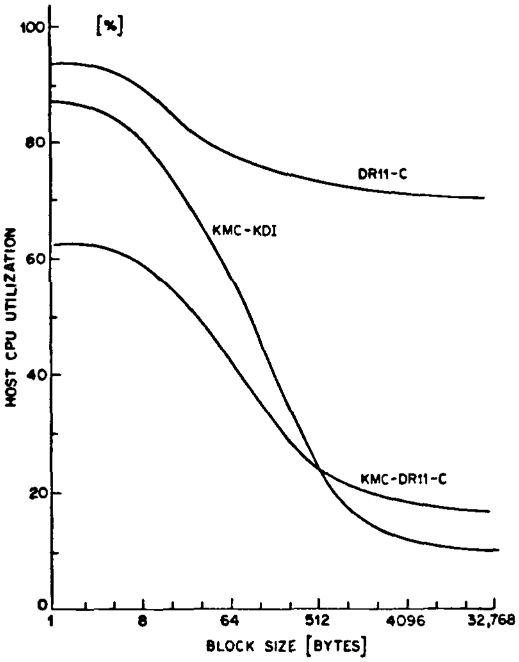

Figure 9. Kernel-Level Host CPU Utilization

The CPU utilization comparing kernel-level transfer and user- level transfer is shown in Figure 10.

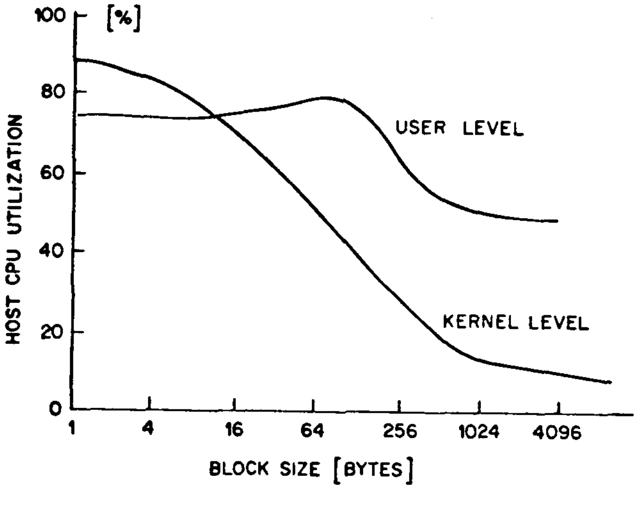

Figure 10. User-Level Host CPU Utilization

The CPU utilization has to be considered in relation to the obtained throughput. For example, one could plot % CPU usage per Kbytes/sec transfer, which would avoid the somewhat misleading intersection of curves in Figures 9 and 10. However, we preferred to present the raw measurements.

This raises the question of the usefulness of the CPU utilization measurement. For purposes of design analysis, one would like a finer breakdown of the resources expended. Such a measurement has been conducted and is reported below under response time measurement. Another viewpoint is that of the system engineer sizing a system for an application. Given a block size and a required data transfer rate, how much of the host CPU is taken up by communication work? Notice that a similar question could be asked with regard to local I/O traffic i.e. file system activity. Figure 11 illustrates both situations.

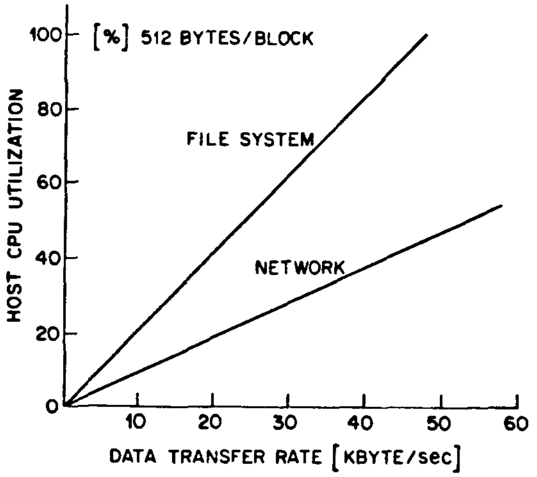

Figure 11. Operating Characteristic: CPU Use Versus Data Traffic

For the block size of 512 bytes, we show CPU utilization as a function of network data traffic or local disk I/O traffic. The latter uses the measurement result that each disk block access requires an average of 10 ms CPU time. One could use this "operating characteristic" for scheduling purposes, for example to set a limit on the amount of permitted network traffic such that a certain fraction of the CPU is available for local work.

Response Time

The time it takes a user process to send a short message to another process on a different computer and to receive an acknowledgement has been measured as 10.6 ms. We have found that 8.8 ms of that time is actually spent in the UNIX file system, i.e. is operating system dependent. Our special attention has been directed at the kernel-level breakdown of the operating-system-independent component. Using the hardware monitoring technique explained above, we obtained the timing diagram (Gantt Chart) of Figure 12.

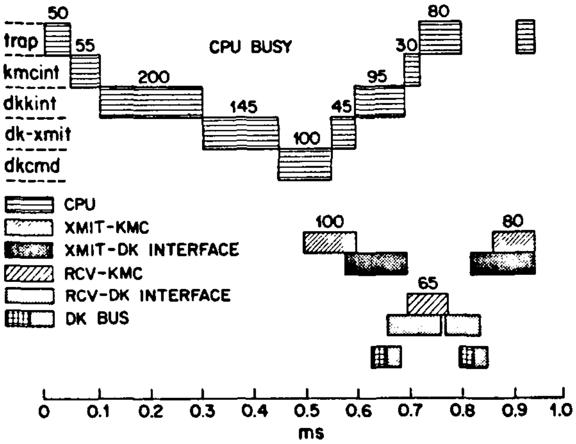

Figure 12. Timing Diagram for Short Message

The Kernel-level exchange takes 1.8 ms. Figure 12 shows a 1 ms interval for the processing of one message going through the following components.

a. The host CPU sending the first message, with five layers of software indicated on the left,

b. the KMC front-end on the above CPU,

c. the Datakit interface logic (KDI) of this KMC,

d. the front-end KMC on the receiving CPU,

e. the Datakit interface logic (KDI) of this KMC,

f. the Datakit bus with up-link and down-link packet cycle.

The sequence of events is as follows. The host enters the trap handler as the result of an interrupt (previous cycle reported complete) and processes the interrupt *(kmc-int, dkk-int)*. It prepares the message to be transmitted *(dk-xmit)* and hands it off to the front-end *(dk-cmd)*. Going from left to right, we see the following devices become active: (1) the sending KMC, (2) its peripheral, (3) the Datakit bus, (4) the receiving KMC's peripheral, (5) the receiving KMC's peripheral, (6) the receiving KMC, (7) its peripheral again for the acknowledging packet, (8) the Datakit bus, (9) the peripheral on the first KMC again, (10) the first KMC, which finally at 900µs interrupts the host to complete the cycle). The message processing is actually completed somewhat later, in *dkk-int*, but we have accounted for that time earlier when we processed the previous message.

Notice that the total cycle takes about 900µs, of which the host CPU is active during 800µs. Transferring a larger message would not increase this activity, since only pointers are passed at the kernel level. However, the activity of the front-ends, of their peripherals, and of the switch would proportionately increase.

To put the above diagram into perspective, notice that it shows the interaction of a 0.3 MIPS host with a 5 MIPS front-end. In such a configuration, we observe that the kernel level transfer rate is mainly determined by the speed of the front-end, whereas the host speed is the dominating factor for the responsiveness of the communication.

Setting Up Virtual Circuits

In order to understand the following measurements of setting up a virtual circuit between two computers, one has to consider the steps involved in this activity. There are two UNIX processes on different computers, one "dialing" the other for a connection. At first the calling process opens a special "dial" file and obtains a file descriptor (the process' end of the circuit to be set up). This step is equivalent to getting dial tone on a phone system. Next the process issues an I/O control command on this file descriptor with the name of the called subscriber as an argument. Finally, the process issues a read command on this file descriptor and waits, i.e. listens. In the meantime, the local driver has allocated a free (odd-numbered) circuit to the file descriptor and sent a message over the signaling circuit (1) to Common Control that contains the newly allocated circuit and the name (or address, as explained above) of the called party. The called party has been listening for signal messages on its circuit 1, and is informed of the call as explained above. Finally, the called process issues an open command with the circuit number taken from the signaling call. This entire sequence has been timed for three different computers in the role of Common Control. Figure 13 shows the results.

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

Figure 13. Circuit Set-Up Times

Again we have noticed that over 50% of the time is spent in the UNIX operating system.

Common Control Reliability

The Common Control of the switch represents a single point of failure in the node, which is an undesirable property. We have taken special efforts to minimize the impact of such a failure. The program runs on a dedicated DEC LSI-11/02 computer with no secondary storage (the computer is actually shared with a monitor and maintenance program, see above). During 15 months of operation, we have not had a single hardware failure of this computer, but many software failures.

The highlights of our Common Control are:

i. Warmbooting: During a Common Control failure, the existing circuits remain unaffected. During the failure, circuits cannot be set up or taken down. Rebooting Common Control preserves the circuits existing before the failure.

ii. Control Protocol: As explained above, pending control requests from subscribers survive Common Control failures; they will automatically be reissued after a restart.

iii. Hot Stand-by: A spare Control computer acting as a stand-by monitors the active Common Control, which has to send periodic sanity messages. When the stand-by notices that the active Common Control is failing, it takes over the Common Control function by rerouting all circuits from the failing to the stand-by computer. The whole process takes one minute; actually, the switchover is accomplished in 2 seconds, but the new Common Control has to make sure that it knows about all active interface modules, even those that may temporarily or inadvertently have no terminating circuit. To that end, Common Control has to wait one or two complete maintenance polling cycles (the clocks poll every module for status periodically).

Connection of UNIX Systems

In our laboratory, we are operating a network of eight computers: DEC PDP-11/23's, PDP-11/45's, a PDP-11/70, and a VAX 11/780. The last two provide time-sharing service; the others are used for special development. All run under the UNIX Version 7 operating system, which is unchanged except for the addition of drivers to interconnect the computers through the Datakit switch. A separate DEC LSI-11/02 computer holds the Common Control and Network Monitor programs (not under a UNIX system). We shall now explain how our communication subsystem is used for this purpose.

After each UNIX system has been booted, a network server process is started in each computer as part of the initialization procedure. Each server opens circuit 1 on the network link and reports to Common Control that it is willing to accept service of type n on its circuit 3. N is a different number for each computer. The network server stays alive permanently waiting for service requests. At the higher protocol level, it understands two commands; *dcon* for direct connection, and *rexec* for remote execution.

When a user on any of the UNIX systems types the command

dcon system-name

where *system-name* is the name of a remote system in the network (ours have have bird's names like Eagle or Owl), the user's terminal appears to be connected to the remote system until a log-out command is given, which returns the session to the local system.

Alternatively, a user may type

rexec system-name command-line

and the named *command-line* only will be executed on the remote system, with the network server acting as the remote user. Eventual output from the command's execution will be returned to the user.

Having presented the functionality of remote connection and execution, we need to digress into some of the UNIX architecture to explain our implementation. The UNIX process is exceptionally well suited to exploit the virtual circuit concept. All process I/O is modeled after the file I/O where a file is an unformated stream of bytes. This includes terminal and device I/O. For a process (i.e. command) to communicate with the outside world, it usually opens a standard input file and a standard output file. The command language interpreter known as the *shell* allows great freedom in redirecting these streams [Ritchie 1974]. When a process is created, it inherits all the open files of its parent. Normally, the shell is the parent of executing commands.

The *dcon* command acquires a local odd-numbered circuit and issues a service request on circuit 1 to Common Control, who forwards it to the named remote network server. The server is informed via its circuit 3 that it has a service request on a newly-allocated even-numbered circuit. The server converses with *dcon* across this circuit to identify the remote user and then logs this user into the remote system (accounting and current-user file, etc.). The server then spawns a *shell* which is connected with its standard input and output to the newly allocated circuit. The *dcon* command takes the user's terminal input processing (i.e. line editing) and forwards the input to the remote *shell* which in turn automatically redirects all the standard output back to *dcon*.

The remote execution is much simpler. When the *rexec* command is issued, similar action as above is taken except that the server receives a whole command line instead of the *dcon* request. The server simply executes the command and returns the resulting output.

Besides the network servers in each system, one system is initialized with a **boot server,** which announces its service to Common Control as of type N=0. We have a ROM bootstrap program on the PDP-11/23 computers which asks for service type 0 and understands the higher level bootstrap protocol managed by the boot server.

A Distributed UNIX System

We have also built a distributed operating system based on UNIX which uses our communication subsystem. Figure 14 shows the configuration.

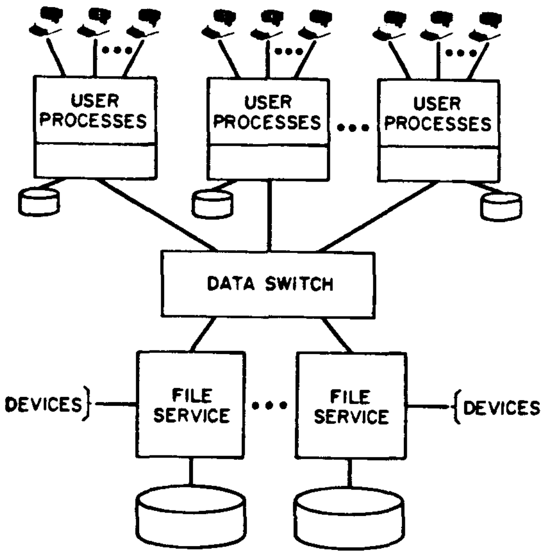

Figure 14. Distributed UNIX System

Rather than having a network of autonomous UNIX systems as described above, the S/F-UNIX system combines computers which receive file service (top-row) and file servers (bottom row) in a single system. Users are logged into the top row computers which run under the S-UNIX operating subsystem. The file servers in the bottom row run under the F-UNIX operating subsystem and together present a global file system view, i.e. a singly-rooted tree as the file name space (Figure 15).

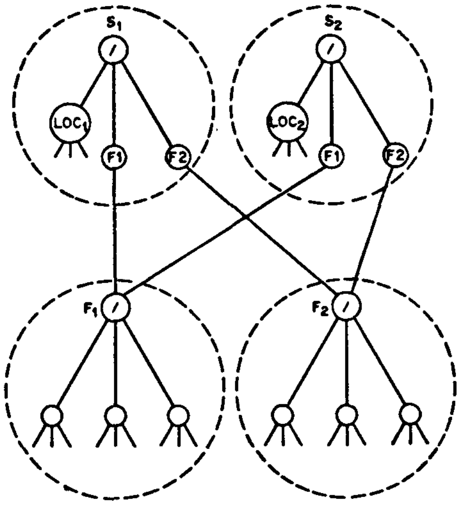

Figure 15. File Name Space of a 2/2 System

The name space is constructed at initialization time when each S-UNIX subsystem issues a "mount file server" request to each of the F-UNIX subsystems.

The details of the architecture and implementation are reported elsewhere [Luderer 1981]. We will restrict ourselves here to issues of using the communication subsystem.

In our first implementation, we established a single permanent virtual circuit between each S-UNIX subsystem and each F-UNIX subsystem. In the file servers, the endpoint of each circuit was served by a file server process. Thus, each file server computer had as many server processes as there were S-UNIX subsystems having mounted its file space. We experimented with various ways of multitasking file service requests, on both the S-UNIX and the F-UNIX side, until we found that the circuit facility itself could help with this problem.

In the current implementation, the S-UNIX side sets up several circuits, each with a file server process on the F-UNIX side, which offers us a cheap way of multitasking in the file server.

One yet unattainable objective is to dedicate a virtual circuit to each open remote file.

We are prevented from currently doing this since our interfaces do not support a sufficiently large number of virtual circuits. New hardware interfaces would be required, and we would also like to see an operating system architecture that supports more flexible memory management than the current PDP-11 family. With the era of VLSI upon us, we are looking for opportunities to identify functions that could be implemented in hardware. An efficient interface for a large number of circuits that "demultiplexes" incoming data right into user process space would be our objective.

Conclusions

We have built a data communication system around a virtual circuit switch. We have used this system to interconnect a local network of autonomous systems and also to form the foundation for a distributed system which relies on fast file transfer. We believe that our contributions lie in three areas: First, the achieved performance of the host-to-switch interface has been increased and knowledge about the operation of this interface has been added. Second, we have shown a number of steps that can improve the reliability and recoverability of a central switch. Third, we have shown how virtual circuit switching service can be used to connect existing systems or to provide a foundation for a distributed system.

Acknowledgments

We acknowledge gratefully the help provided by A. G. Fraser and G. L. Chesson to give us access to Datakit hardware and software. Special thanks is due our colleagues J. P. Haggerty and P. A. Kirslis, who designed and implemented the S/F-UNIX system thus becoming the first users of our communication system. We have also benefited from the work of L. E. McMahon and D. M. Ritchie. E. Sirota from Brown University built the KMC/Datakit interface under the direction of R. C. Haight while here on a summer job. Our measurement work was aided by facilities developed by J. Feder and D. A. De Graaf. P. F. Long and C. Mee III helped us with learning about the intricacies of the KMC font-end processor. We had several enlightening discussions with A. S. Tanenbaum. J. F. Reiser provided helpful suggestions on the design of the host/front-end interface.

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

Finally, we appreciate the efforts of the reviewers and of V. Walsh, who gave us excellent turn-around time when the final version of this paper had to be typed in a rush.

References

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

[Closs 1980] F. Closs, R. P. Lee, "A Multistar Broadcast Network for Local Area Communication", IBM Research Report, RZ 1052 (#37705) 12/31/80, 20pp. Yorktown Heights, NY

[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.

[Feder 1980] Communication with J. Feder of Bell Laboratories

[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

[Long 1981] Conversations with P. F. Long and C. Mee III of Bell Laboratories

[Luderer 1981] G. W. R. Luderer, H. Che,J. P. Haggerty, P. A. Kirslis, W. T. Marshall, "A Distributed UNIX System Based on A Virtual Circuit Switch", to be presented at 8th Symposium on Operating Systems Principles, acm SIGOPS, December 14-16, 1981.

[Pouzin 1978] L. Pouzin, H. Zimmermann, "A Tutorial On Protocols", Proceedings IEEE, Vol.66, No. 11, November 1978, pp. 1346-1370.

[Pouzin 1977] Louis Pouzin, "The Pop Art of Public Data Networks", in "Data Communication Networks", Online Conferences, Uxbridge, Middlesex, England, 1977

[Rawson 1978] E. G. Rawson, R. M. Metcalfe, "Fibernet: Multimode Optical Fibers for Local Computer Networks", IEEE Trans. Comm. Vol. COM-26, No.7, 1978

[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.

[Shoch 1979] J. F. Shoch, "An Annotated Bibliography on Local Computer Networks", IFIP WG 6.4, Working Paper 79-1, Xerox PARC Technical Report SS-79-5, 1979

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