THE NETWORK UNIX SYSTEM

Gregory L. Chesson[†]
University of Illinois
Urbana, Illinois 61801

Abstract: A Network Interface Program (NIP) is that part of an operating system which interfaces with similar entities in a network. Normally, the NIP is a collection of software routines which implement interprocess communication, interhost protocols, data flow controls, and other necessary executive functions. This paper discusses the organization of the NIP currently being used with the Unix operating system on the ARPA network. The Network Unix system is noteworthy because of the natural way that network and local functions are merged. As a result the network appears as a logical extension to the local system - from the point of view of both the interactive terminal and user program.

Key Words and Phrases: ARPANET, operating system, protocol, Unix

CR Categories: 4.3

1. Introduction

The Unix time-sharing system [1], developed at Bell Laboratories for the Digital Equipment Corporation PDP-11/40 and 11/45 computers, has been installed at 100 or more sites. It has proven to be an efficient and powerful tool for numerous applications. At the University of Illinois, Unix has been placed on the ARPANET by adding a Network Interface Program (NIP) to the standard Unix system. Other sites that are using or have received copies of this software include UCLA, UC at Berkely, MIT Lincoln Laboratories, Purdue University, the Rand Corporation, and the Stanford AI Laboratory.

The architecture of standard Unix simplified many aspects of the design of the Network Unix. As a result, the Unix NIP enjoys several properties which are not often found in a single networking executive: (1) the system will work with a variety of network hardware interfaces - it is not limited to operation on the ARPANET alone; (2) the resident core overhead is low - about 3.5K 16-bit words; (3) protocol state machines are implemented by a natural mechanism - thus tending to be easy to maintain; (4) user interfaces to the NIP are clean and simple; (5) the NIP is written entirely in a high-level language, as is Unix; and (6) a network Unix can easily operate as a link between the ARPANET and other networks.

The network Unix system is being used to build sophisticated network mechanisms. Some of this work is described in section 5 below. Other applications are mentioned here in passing. In particular, "stripped" versions of the current system can easily be used as satellite processors for larger systems. Also, it is possible to connect several Unix systems together to achieve multiprocessing as well as the sharing of resources.

Although this paper is primarily an exposition of the architecture of the NIP in Unix, some familiarity with ARPANET protocols and Unix features is necessary. Therefore, a summary of appropriate terminology and protocols is included in the next two sections in order to simplify the presentation of the NIP design.

2. Terminology

2.1 Unix

The operating system uses the standard DEC address relocation hardware to partition the physical memory into two parts - kernel space which is reserved for the resident portion of Unix, and user space which is available to user programs. The user's keyboard interface to the system is a program called the shell. The system forks (i.e., creates a new process) a copy of the shell for each terminal logged onto the system. Commands typed by the user are read and analyzed by the shell which then starts up system programs for the user. Filenames are part of the command syntax recognized by the shell. Since the directory structure of the Unix filing system is a tree, a pathname is an ordered list of directory names that makes up a path from the root of the tree to a particular file. Directory names are separated by / in a pathname. For example,

[†]Work supported in part by NSF DCR 72-03740 A01, the Department of Computer Science of the University of Illinois, and the Center for Advanced Computation at the University of Illinois.

/usr/greg/f specifies file f in directory greg, which is a subdirectory of usr.

The file system and directory structure are implemented by a simple pointer system. A directory entry contains only a name for the associated file and a pointer. The pointer is an integer called the i-number (index number) of the file. The i-number is used as an index into a system table (i-list). The indexed element of the i-list is a 16-word data block known as an inode. An inode contains all the information necessary to access a file. In particular, an inode indicates whether a file is an ordinary data file (containing file names and pointers to inodes), or a special file.

Within the Unix kernel, each I/O device controller is assigned a major device number. Each controller may be associated with one or more actual devices, such as disks or tapes. These are distinguished by minor device numbers. The major device numbers are used by the system to select I/O device drivers, and the minor device numbers are used by the drivers to select particular devices. Each I/O device in a Unix system is also represented by at least one special file. These files are usually located in directory /dev. For example, /dev/tty4 would be the special file associated with a particular terminal attached to the system. The inode associated with a special file contains the major and minor device numbers and the flag bit which marks the inode "special." Read, Write, and Seek (when appropriate) commands to special files are passed directly to the device drivers by means of the information contained in the special file inodes. Hence I/O commands on /dev/tty4 would activate that terminal accordingly. This feature of Unix facilitates the treatment of network I/O as standard I/O.

2.2 ARPANET

The Interface Message Processor, or IMP, is the packet switching computer which provides the basic data transmission facility of the ARPANET. A host is a computer attached to an IMP. Each host is assigned a unique host number. A message is the unit of transmission (up to 8095 bits) between a host and an IMP. A leader is the first 32 bits of a message. It specifies, among other things, the destination host and a link number. The link is used to demultiplex messages entering a host into 256 possible channels. Link zero is assigned as the control link used for host-host protocol exchanges. Links 2 through 71 are available for general use, links 196 through 255 are available for experimental use, and the rest are otherwise assigned or reserved. A socket number is a 32-bit value that identifies a software I/O port belonging to a process in a host system. A process is uniquely identified in the network by its host number and socket number. Even numbered sockets are defined to be "read" sockets that receive data from the net; odd numbers designate "write" sockets. A connection is a simplex (unidirectional) data path between processes consisting of two sockets (one read socket and a write socket) and a link. When incoming data arrives at a host, the NIP program uses the incoming link number to determine which local socket should receive the message.

3. Protocols

A protocol is a set of conventions that cooperating systems agree to observe. In this case, the ARPANET protocols [2] specify the form and content of messages that are exchanged between the various elements of the network. The actual transfer of user data between network hosts is supported by a hierarchy of protocols. These protocols are related to the various logical levels of data transfer between hosts: bit transfers between IMP's, regular message transfers between hosts, communication between software processes, and file transfers between systems. These are outlined in the following paragraphs, beginning with the lowest level.

3.1 IMP-to-IMP

Each IMP may be connected to as many as five other IMP's and up to four hosts. The low-level Imp-to-Imp operations do not affect the NIP design since they are transparent to a host system. Therefore, they will not be discussed here.

3.2 IMP-host (first level protocol)

The IMP and host communicate through an IMP interface. Interfaces in current use range in complexity from simple data channels to small computer systems depending on the size and nature of the host. However, for the purpose of protocol definition, the IMP interface is merely a data path. Given a suitable IMP interface and device driver, the IMP-host protocol is based on the 32-bit message leaders. Control bits in a leader indicate whether or not the leader is followed by additional data. If there is additional data (up to the maximum message size) then the leader plus the data constitutes a regular message. If there is no additional data then the leader is an IMP-host control message. The most commonly occurring IMP-host signal is the Ready-For-Next-Message (RFNM) control message. A RFNM is a positive acknowledgment from a distant IMP that is sent to the local IMP when the distant IMP begins copying a regular message from the local IMP. All other IMP-host messages are diagnostic in nature or indicate error conditions.

3.3 Host-host (second level protocol)

The control messages used for this protocol consist of a 72-bit (9 byte) header followed by additional control information. These control messages are the basis for opening and closing connections between hosts, transferring data across connections, and for performing several other auxiliary functions.

3.3.1 Connections

Request-For-Connection (RFC) commands are exchanged between hosts for the purpose of establishing connections between processes. There are actually two RFC commands, one for a prospective receiver and one for senders. Each RFC contains a pair of socket numbers (the pair desired for the connection); the receiver's RFC also specifies the link to be associated with the read socket. The NCP in a host must compare the RFC's that it sends

to other hosts with those it receives. When the socket pair in an incoming RFC from some foreign host matches a pair sent to that same host, then the connection is considered to be open. Fine points in this process which have been ignored in this discussion include timeout and queuing policies to be observed during the connection process, as well as that part of the protocol which defines the byte size to be used in subsequent data transmissions over the connection.

The RFC commands for setting up simplex connections are used by higher level protocols (see 3.4, 3.5) to establish duplex connections.

3.3.2 Flow Control

Hosts are required to maintain a message counter and bit counter for every open connection. These counters are initially set to zero. No data can be sent over a connection until the receiver sends an allocate command to the sender. The allocate tells the sender the maximum number of messages and the total number of bits that can be sent. Every time data is transmitted over the connection, both the sender and receiver decrement their message counters by one and the bit counters by the number of bits in the message. No data transfers may take place that would cause either the message counter or bit counter to become negative. Thus the receiver must continuously send allocates to the sender. This technique guards against the possibility of a fast sender overrunning a slower receiver with data.

3.4 Initial Connection Protocol (third level protocol)

The ICP, as it is called, is the standard ARPANET mechanism for connecting a process in one host with a process in another host. The ICP uses the host-host protocol to establish a pair of connections between hosts. The result is a bidirectional data path consisting of a read and write socket for each process and a link for each read socket.

3.5 Higher Level Protocols

The Telnet protocol allows a user at a terminal on one host to log on to a foreign host system as though his terminal were attached to the foreign host. This is accomplished by using the ICP protocol to connect to a Telnet "server" process in the foriegn host. The Telnet protocol itself consists of the data and control commands that are passed over the duplex connection established by the ICP.

The File Transfer Protocol (FTP) is used for transferring files between hosts. An FTP exchange consists of opening a Telnet connection to a foreign FTP socket, carrying on an initial conversation, opening a simplex data connection, transferring the file over the data connection, and then closing the connections.

Larger hosts on the ARPANET support a Remote Job Entry (RJE) protocol which enables a distant user to submit jobs to a batch job stream. Although there is a prototype official ARPANET RJE protocol [4], existing network RJE implementations are local adaptations.

There exist numerous experimental or proposed higher level protocols. Examples include schemes for file access (as opposed to file transfer), interactive graphics, procedure calling, and internetwork communications, i.e., communications between ARPANET and nonARPANET networks. These examples are given for completeness only and will not be discussed since a survey of protocol development is outside the scope of this paper.

4. NIP Structure

The Network Unix system is a standard Unix augmented by a Network Control Program (NCP), protocol programs, and network special files. The NCP is implemented in two parts: an NCP kernel which is made part of the core-resident Unix code; and the NCP daemon which is a continuous background user-level process in the system. The NCP daemon implements the host-host and ICP protocols; the NCP kernel services the IMP, the NCP daemon, and user programs; the protocol programs execute in user space utilizing the NCP kernel and daemon to implement higher level protocols; and network special files provide the basis for the interface between user programs and the NCP. The NCP kernel (about 3.5K words) is the only resident software in the NIP. The NCP daemon (about 8K words) and other programs are only brought in to memory (as user programs) when they are needed. Since the NCP is primarily needed for opening and closing network connections, and since the NCP kernel manages network data flow, this "split" organization conserves memory without sacrificing performance. In fact the NIP was developed and will run on a PDP-11 with only 32K words of memory - the minimum required for standard Unix.

4.1 Network Special Files

In Unix, special files map a character string representation of physical device names into their assigned integer names within the system (refer to section 2). Network special files perform a similar function in that they map network host names into host numbers. The network special files are found in directory "/dev/net." For example, "/dev/net/harv" represents the Harvard PDP-10, and "/dev/net/london" represents the PDP-9 front-end in England. Each network special file has a major device number of 255 which distinguishes it from the standard Unix device numbers which are assigned starting from zero. The minor device number of a network special file is the assigned network identifier for the corresponding host.

4.2 Unix-NCP Interface

Programs access the network software by applying standard Unix I/O calls (Open, Close, Read, and Write) to network special files. Thus, the network can be accessed by a program written in any language that provides the standard interface to the file system. Usage of these commands with network files is just what one would expect. That is, the Open command establishes a connection between the calling program and a server process on the foreign host. Read and Write commands transfer data between the two processes, and a Close terminates the connection.

The form of the Open call is:

```
fd = open("/dev/net/hostname," mode);
```

The Open command returns a file descriptor (fd >= 0) if the connection is opened successfully, and minus one otherwise. The first argument is the Unix pathname of the desired network special file. The second argument is normally 0, 1, or 2 signifying that a read-only, write-only, or a read-write (i.e., standard Telnet) connection is desired. This interpretation coincides with standard Unix usage when the value of "mode" is 0, 1, or 2. However, any other value is interpreted as the address of a control block in the user program during a network Open. The control block fields and their meanings are given below:

type	-	indicates (1) whether a connection or a "listen" for a foreign RFC is desired on a local socket, (2) simplex or duplex connections, (3) absolute socket numbers or numbers relative to a base, and (4) whether an ICP or a direct connection is desired.
file id	-	file descriptor used when the open refers to an already open network file.
local socket	-	refers to a local socket number.
foreign socket	-	specifies a foreign socket number.
host	-	specifies a foreign host.
byte	-	specifies the connection byte size.
alloc	-	specifies the nominal size of an allocate command sent to a foreign host.
time	-	the time in 60ths of a second to wait for the foreign host to fulfill the request before canceling it.

If any fields in the control block are zero, the NCP will use default values in their place. The flexibility in opening network connections afforded by this control block scheme greatly simplifies the task of implementing higher level protocols.

The Read/Write/Close calls are equivalent to the standard Unix calls. They have the following form:

```
nbytes = read(fd,buffer,count);
nbytes = write(fd,buffer,count);
status = close(fd);
```

In each of these calls "fd" is a file descriptor returned by an Open call, "buffer" is a buffer address, and "count" is the number of bytes requested for transfer. On data transfers, "nbytes" is set to the number of bytes actually transferred, and in all three cases a -1 is returned on an error.

It should be pointed out that although the ARPANET does define an INTERRUPT signal on a link, the Unix NCP does not currently implement such a mechanism. Indeed, file system commands such as those mentioned here do not provide a natural interrupt mechanism. However, there does exist within the standard Unix a signal call by which a user program specifies the address of a software procedure that the system will invoke in response to an external event. Normally these events are abnormal occurrences such as illegal instruction traps, or loss of carrier on a data connection. Nevertheless, this mechanism could be pressed into service for the purpose of handling incoming interrupt signals. Outgoing interrupt signals can be generated with ease by a variety of software mechanisms. However, we have not felt obliged to implement these interrupt facilities, and mention the subject here for completeness only.

4.3 NCP Kernel

Referring to Figure 1, the NCP kernel includes everything below the dotted line. The principal data structures associated with the NCP kernel are the Read and Write connection tables, the network file table, and data buffers. These structures refer to standard Unix structures that already exist in the kernel; specifically as inodes, file blocks, and kernel buffers. The NCP kernel uses existing Unix procedures for managing these structures (see 4.3.2 and 4.3.3).

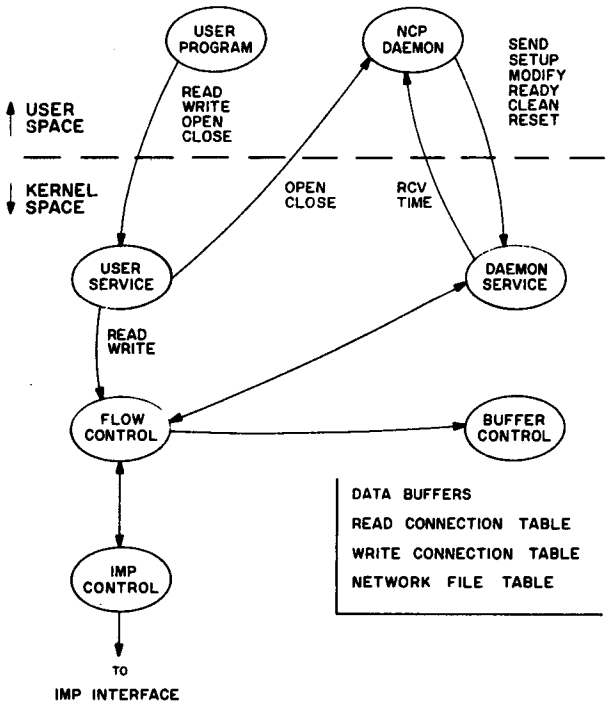

Figure 1. NCP Data and Control Flow

4.3.1 User Service

Communication between user programs and the user service routines is accomplished through the existing Unix system call mechanism. User I/O calls on a network special file are detected by four conditional statements that augment the standard Unix file system. These four statements are the only changes to standard Unix code required by the NCP - and all they do is check for I/O calls on special files having major device number 255. System calls distinguished by this simple mechanism are diverted to the NCP kernel. There the

Open/Close requests are sent to the NCP daemon while transfer requests are processed in the kernel. Communication between the NCP kernel and NCP daemon is implemented by a special file (/dev/ncpkernel). Unlike the network special files described in 4.1, the ncpkernel file has a normal Unix major device number. However, the device driver for /dev/ncpkernel is actually the NCP kernel. That is, Unix is set up so that Read/Write calls on /dev/ncpkernel are processed by routines in the NCP kernel. These routines essentially copy data between daemon buffers in user space and kernel buffers in kernel space.

4.3.2 Connection and File Tables

Whenever a file is opened in Unix, the system sets up certain data structures in the kernel that describe the file and which are updated as the file is modified. This is true as well for network special files. However, since standard Unix I/O calls on network files are diverted to the NCP kernel, most of the space in these data structures can be used by the NCP kernel for its own purposes. In particular, Unix inodes are used to represent sockets, and each standard Unix file control block can point to as many as three of these sockets. The socket inodes contain message and bit counters (refer to 3.3.2), host and link data, and other parameters that are part of network data flow control.

For each local socket there exists an entry in one of the connection tables (i.e., the read table for read sockets; write table for write sockets). A table entry consists of a pointer to the socket, the foreign host number and a link number.

The network file table contains a one word pointer to a Unix file control block for each open network file. The file table is the basis for communication between the NCP kernel and the NCP daemon - open network files are referred to by their index number in the file table. The sizes of the file table and connection tables are compile time constants - they are all set for 32 entries in the current system.

4.3.3 Buffer Control

The buffer control section of the NCP kernel manages a pool of 64-byte buffers that are obtained from 512-byte buffers allocated by standard Unix. The NCP kernel will take up to 8 512-byte buffers from Unix, returning them when free. Also incorporated into the buffer control section are procedures for concatenating messages, appending data to messages, and copying messages to and from user space. A "message" in this context is an IMP-host regular message, i.e., network data, and may occupy several of the small 64-byte buffers.

4.3.4 IMP Control

The IMP control section handles the IMP-host protocol and other mechanics of transferring data between the host and IMP.

4.3.5 Daemon Service

The daemon service routines process commands (SEND, etc.) from the NCP daemon, and send messages from the net to the daemon (RCV). The SEND command, as the name implies, is used by the daemon to send protocol messages to other hosts. The other commands recognized by daemon service procedures are used to update kernel data structures (connection and file tables) as directed by the NCP daemon.

4.3.6 Flow Control

The key to the operation of the split NCP is really the flow control section. This part of the kernel implements user data flow control according to the host-host protocol. This entails (1) sending allocate commands to foreign hosts (refer to section 3.3), (2) accepting allocates from foreign hosts, (3) maintaining message and byte counters affected by allocate commands and data structures, and (4) implementing the reallocation protocol. Since user Read/Write and flow control processing routines are core resident at all times, user data transfers to and from the net are efficient.

Flow control as implemented in the network Unix is constrained by a design requirement that the system operate on a PDP-11 with only 32K words of memory. The algorithm is as follows:

a) a process writing to a foreign host will be buffered by the NCP kernel up to a limit of 4096 words. When the limit is reached the sending process is put to "sleep" until some of the buffered data is sent to the foreign host.

b) the message header of every message coming into Unix from the IMP is examined in a fixed buffer dedicated to that purpose. If the header indicates that it is part of a regular message, then additional buffer space is allocated from the buffer pool as required. If space is not available, then the kernel process that reads from the IMP blocks until awakened by a space-freeing primitive.

Faster algorithms than the one described here require more memory than is consistent with a minicomputer installation. Large hosts with virtual memory can allocate large virtual buffers for every open connection. This kind of scheme can be implemented to an extent with the memory management unit of a PDP-11/45, but not with an 11/40. Since the bandwidth of the current system is more than adequate for our needs, and is in fact often greater than the available bandwidth of the ARPANET, the algorithm which is compatible with both 11/40's and 11/45's is preferred over a faster algorithm which would not be compatible.

The NCP daemon is a continuous background process in Unix, running as a user program. Inputs to the NCP daemon consist of Open, Close, or RCV commands from the NCP kernel which are read from the communication file /dev/ncpkernel. As explained in 4.2 and 4.3, the Open and Close commands arise from Open and Close requests on network special files generated by local user programs. The RCV command indicates incoming network traffic for the NCP daemon.

Most of the time the NCP daemon is "asleep" waiting for a read on the communication file to be satisfied. However, when input commands do arrive, the program responds in a number of ways. It can (1) update its internal data structures, (2) send protocol messages to other hosts, (3) send commands to the NCP kernel, and (4) log statistics and events in external files. Depending on the state of the sockets and files associated with an incoming command, the NCP kernel may take any, or all, or none of the above actions. In this sense the NCP daemon is simply a finite state machine - for each input it computes a next state and an output function depending on the current state. Actually the transition functions in the program are specified for a single socket and network file. When an input command is decoded, it will specify the particular network file or socket to be affected. Thus the state machines in the NCP daemon consider one network event at a time.

The most complicated state machine in the NCP daemon is the "socket machine." There are nine possible states for each socket (2 listen states, 2 rfc states, 4 states associated with closing, a socket open state, and a null state) and nine operations that the socket machine can accept as input commands:

two listen commands
local rfc command
foreign rfc command
foreign close command
local close command
ncp daemon close command
timeout command
foreign host died signal

This could lead to an 81-state machine. However, the implementation is reasonably compact since there are only 25 unique actions that are performed at the 81 possible states. Each of the possible actions is implemented as a function, and the state table is a nine-by-nine array of function addresses. The state table indicates which function to call for a given configuration, and the next state is determined by code in each function.

4.4.1 Outputs

The commands that the NCP daemon can send to the NCP kernel are given below:

- send - transmit data to the network
- reset - clean up all table entries and processes related to a specific host
- clean - release a kernel socket (inode)
- ready - wake up any processes that are waiting for the specified network file
- mod - change the state of a kernel socket
- setup - initialize a kernel socket

4.4.2 Daemon Data Structures

The NCP daemon maintains several arrays that each have one entry for every possible host on the network, and file and socket structures that relate to local processes. These are:

- hostup - an array of 256 bits, one for each possible host. A one indicates that the host is available.
- rfnm - an array of 256 bits. A bit set indicates that an rfnm is outstanding from the indicated host.
- retry - an array of 256 counters. Each one keeps track of the number of times a message to a host is retransmitted.
- probuf headers - an array of 256 pointers to protocol buffers. The NCP daemon assembles host-host messages in buffers which are allocated as needed. The header array pointers map host numbers into the addresses of these protocol buffers.
- socket struct - the NCP daemon data structure for a socket. It indicates the local socket, foreign socket, host, link, byte size, network file, and socket state of a particular socket.
- file struct - the NCP daemon data structure for a network file. It contains the kernel's id for the file (i.e., index into kernel file table), a file state indicator, and the location of NCP socket structs associated with the file.

4.4.3 NCP Daemon Main Loop

The algorithm given below closely paraphrases the main loop in the actual code of the NCP daemon. Note that 3 sockets are allocated on an open because the ICP uses one socket as it establishes two others. Note also that statistics are kept on all incoming host-host messages (RCV).

```
procedure ncpdaemon;
  {
    Open communication file /dev/ncpkernel;
    while not end-of-file on /dev/ncpkernel
      do {
          Read next command;
        if (command = OPEN) then
                {
                 allocate file, 3 sockets, and
                    link;
                 call socket machine with open
                    command
                }
        if (command = CLOSE) then
                {
                 if file is in use then call
                    socket machine with close
                    command
                }
        if (command = RCV) then
                {
                 decode host number from leader;
                 update statistics;
                 call specified host-host
                    procedure;
                }
        if (protocol was generated) then
                send protocol;
      }
  }
```

5. Current and Future Work

Unix currently supports the Telnet and FTP protocols to the extent that local Unix users may log into a foreign host with Telnet or transfer files between Unix and a foreign host that has an FTP server. However, Telnet and FTP servers are planned for Unix. They are expected to simplify the sharing and updating of software by Unix users on the ARPANET.

Because the Unix interface to the NIP uses much of the standard Unix file system, networking programs are easy to write and a significant portion of the Unix shell command syntax becomes meaningful in a network context. Current work involves modifying the Unix shell so that pathnames of the form "/hostname/pathname" will be interpreted as pathnames on the specified foreign host. In general, it would be desireable to be able to replace any Unix pathname in a command line with a "network" pathname. However, the full generality of Unix shell commands would tax the present abilities of the ARPANET, although the results would be worthwhile. For example, the command

```
/host1/prog1 /host2/path2 | /host3/prog2 | prog3 | lpr &
```

would require prog1, prog2, prog3, and lpr to be run concurrently. The & symbol specifies that this four program system is to be run as a separate process - that is, "forked" off as a batch job, thus returning the terminal to the user. Each of these programs has a "standard" input and output (i.e., the user's terminal) in addition to any others it may have. However, in command lines of the type given above, the standard output of a program on the left side of a vertical bar is connected to the standard input of the program to its right. The result is to chain the standard I/O from left to right across the line. So in the example, prog1 executes on host1. It receives an input file from host2. The standard output from prog1 is the input to prog2 running on host3. Prog3 runs on the local system and directs its output to the local line printer process (lpr).

Although there are many practical problems involved with supporting command lines like the one given above, there are some general techniques that look promising. Basically, the Unix shell could open up Telnet connections to the foreign hosts mentioned in a command line. Then "canned" messages and other commands derived from the original line typed by the user would be sent to the foreign hosts. A reasonably simple "daemon" process could be run on each foreign system that could set up the standard input and output connections across the network for the coupling convention expressed by the vertical bars in the Unix shell syntax. The proposed daemon would need the ability to start processes on the foreign host in response to communications from the local Unix. However, unless the foreign host is Multics or another Unix this may not be an easy thing to do. However, there is a simpler approach which can be made to work using only FTP and Telnet. To whit, instead of trying to start concurrent processes the command line could be considered as a sequence of job steps, with some steps taking place on different hosts. The Unix shell could easily start a program on some host, collect the output into a file, and when one program completes use FTP to supply the input for the next program. The control steps would be initiated automatically by Unix on behalf of the user.

The immediate applications that are seen for the "intelligent" shell are in providing simple network RJE mechanisms, in providing useful and efficient distributed data access techniques, and in connecting multiple Unix systems together. The RJE work is fairly straightforward since RJE protocols already abound and protocol programs can be easily implemented given the interface structure of the NIP. Techniques involving cross-network file access techniques have yet to be perfected, but preliminary work indicates that distributed file concepts can be worked into Unix. Experimental work involving multiple Unix systems is currently in progress. From the users' point of view, a multiple processor Unix system is accessed via the shell syntax outlined above. Given a local mini-network of Unix systems, network-wide password authentication is possible; thus a login at one processor can also be a login at the other processors. A small network is being organized along these lines. Although the primary purpose of the mini-network is to provide ARPANET access to other systems through the Unix NIP, distributed processing experiments are also planned. These include the natural coroutine scheme exemplified by the shell syntax already described, as well as experiments where a processor and its resources may be "slaved" to another processor. The object of this work is to provide a test bed for distributed control and resource management algorithms.

6. Acknowledgment

The programmers who developed the NIP design and performed the initial coding are Steve Bunch, Gary Grossman, and Steve Holmgren. Special credit should be given as well to the developers of Unix, Ken Thompson and Dennis Ritchie.

References

[1] Dennis M. Ritchie and Ken Thompson, "The UNIX Time-Sharing System," CACM, Vol. 17, No. 7 (1974), pp. 365-375.

[2] Network Information Center, "Current Network Protocols," NIC 7104, Augmentation Research Center, Stanford Research Institute, Menlo Park, California, Revised June, 1973.

[3] Jonathan B. Postel, "Survey of Network Control Programs in the ARPA Network," Mitre Corporation Technical Report MTR-6722.

[4] Bob Bressler, Rich Guida and Alex McKenzie, "Remote Job Entry Protocol," ARPANET RFC #407, Network Information Center document #12112.