UNIVERSITY OF
ILLINOIS LIBRARY
AT URBANA-CHAMPAIGN
ENGINEERING

NOTICE: **Return or renew all Library Materials! The *Minimum* Fee for each Lost Book is $50.00.**

JUL 06 1988

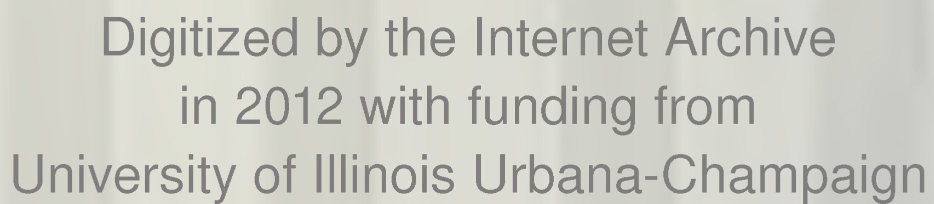
The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.
To renew call Telephone Center, 333-8400

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

ENGINEERING

L161—O-1096


Digitized by the Internet Archive
in 2012 with funding from
University of Illinois Urbana-Champaign

http://archive.org/details/networkunixsyste243kell

0.84
l63c
o. 243

Engin

Center for Advanced Computation

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

224-28

CAC Document No. 243

A Network Unix System for the ARPANET
Vol. 1: The Network Control Program

by
Karl C. Kelley
Richard Balocca
Jody Kravitz

October 1978

THE LIBRARY OF THE
SEP 1 9 1980
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

CAC Document Number 243

A Network Unix System for the Arpanet, Volume 1: The Network Control Program

by

Karl C. Kelley,

Richard Balocca, and Jody Kravitz

Prepared for the
Department of Defense

Center for Advanced Computation and
Computing Services Office of the
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

October 16, 1978

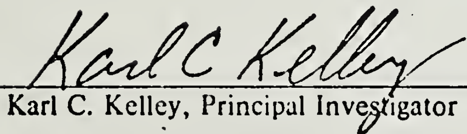
Approved for release: 
Karl C. Kelley, Principal Investigator

Table of Contents

1 INTRODUCTION 1
2 TERMINOLOGY AND OVERVIEW 2
2.1 Unix 2
2.2 Arpanet 3
2.2.1 Protocols 3
2.2.1.1 IMP-IMP (zeroth level protocol) 3
2.2.1.2 IMP-Host (first level protocol) 3
2.2.1.3 Host-Host control link protocol (second level protocol) 4
2.2.2 Sockets 4
2.2.3 Connections 4
2.2.4 Initial Connection Protocol 5
2.2.5 Flow Control 5
2.2.6 Higher Level Protocols 5
2.3 NCP Structure 6
2.3.1 User Program Interface 6
2.3.2 Kernel Functions 9
2.3.2.1 Connection and File Tables 9
2.3.2.2 Buffer Control 10
2.3.2.3 IMP Control 10
2.3.2.4 Flow Control 10
2.3.2.5 Servicing of Daemon 11
2.3.3 Daemon Functions 11
2.3.3.1 Daemon Outputs 12
2.3.3.2 Daemon Data Structures 12
2.3.3.3 NCP Daemon Main Loop 13
3 THE NCP KERNEL 14
3.1 Kernel Parameter Files and Data Structures 14
3.1.1 Kernel Compile-time Parameters 14
3.1.2 Kernel Network Sockets 14
3.1.2.1 Read Socket Format 15
3.1.2.2 Write Socket Format 17
3.1.3 Structure for a Network File 18
3.1.3.1 Network File Structure 18
3.1.4 Connection Tables 18
3.1.4.1 Read Connection Table 19
3.1.4.2 Write Connection Table 19
3.1.5 Network File Table 19
3.2 Connection Establishment in the NCP Kernel 20
3.2.1 User Program Interface 20
3.2.2 NCP Kernel Interface to NCP Daemon 20
3.3 ARPA Network Data Flow within the Kernel 21
3.3.1 User Writes 21
3.3.2 User Reads 22
3.3.3 IMP Device Driver 22
3.3.4 IMP Flow Control Activities 23
3.3.5 Imp Device Driver: Output Side 23
3.4 Sleep-Wakeup Signalling in the Kernel 23
3.4.1 Imp Input 24
3.4.2 NETWORK BUFFERS 24

Table of Contents (cont'd)

3.4.3 Ncpdaemon Input 24
3.4.4 Netfile Opens 24
3.4.5 Clear Host/Host Channels 25
3.4.6 User Reads 25
3.4.7 User Writes 26
3.4.8 Allocation Waits 26
3.5 IMP Input/Output Processing in the NCP Kernel 27
3.5.1 NCP Daemon Interface 27
3.5.2 User Interface 27
3.5.3 IMP Input Process 27
3.6 IMP Output Processing in the NCP Kernel 28
3.6.1 NCP Daemon Interface 28
3.6.2 User Interface 28
3.6.3 IMP Output Process 28
3.7 IMP and NCP Initialization in the NCP Kernel 29
THE NCP DAEMON 30
4.1 Daemon parameter files and data structures 30
4.1.1 Daemon parameter files 30
4.1.2 Daemon Data Structures 31
4.2 NCP Daemon Communication 33
4.3 NCP Daemon Pseudo Special File 33
4.3.1 Daemon Open 33
4.3.2 Daemon Read 33
4.3.3 Daemon Write 34
4.3.4 Exemplary Daemon Communication 35
4.3.5 NCP Daemon Interface 36
5 COMMUNICATION BETWEEN KERNEL AND DAEMON 38
5.1 Kernel to Daemon Communication 38
5.2 Daemon to Kernel Communication 39
5.3 Where and Why the Daemon issues kernel commands: 40
6 SELECTED TOPICS 42
6.1 When the Network is Started 42
6.1.1 Overview 42
6.1.2 Following the Code 42
6.1.3 Imp Input Errors 43
6.2 Initial Host Resetting Algorithm 46
6.3 Opening a Network File 47
6.3.1 Declarations 47
6.3.2 Elements of Openparams 47
6.3.3 The Bits in o_type 49
6.3.4 Examples of Setting o_type 50
6.3.5 The Open Call Itself 51
6.4 The Socket Machine 51
6.5 The File Machine 53
List of References 55

List of Tables & Figures

Figure 1	8
Table 1	15
Figure 2	43
Figure 3	44
Figure 4	45
Table 2	53
Table 3	55

Acknowledgements

The Illinois version of a Network Control Program (NCP) for the Arpanet, and this document describing it, are the products of many contributors. The original design was done by Gary Grossman, Steve Holmgren, and Steve Bunch and each of them had a hand in the original coding. Holmgren carried through for several months thereafter, making changes and additions and bringing up the initial versions of the higher level protocols. Greg Chesson also made important contributions during this time. Jody Kravitz then took over prime responsibility for the NCP and provided major cleaning and clarifying of the code. Particularly important were revisions to make it easier to use other IMP interfaces, and the code for several such interfaces, including the Very Distant Host (VDH) interface. During this period Richard Balocca was also a major contributor. Others who have had a role in either the NCP or protocol software include John McMillian, Jay Goldberg, Paul Jones, Bob Schulman, James Gast, Marsha Conley, and Karl Kelley.

A number of helpful suggestions, revisions, and bug isolations have been contributed by persons using this software or its derivatives at other network sites. Steven Abraham and Steve Tepper, at that time from UCLA and the Rand Corporation respectively, were especially valuable contributors of this kind, and in addition provided some of the early drafts of material in this document. Among the others whose influence can be seen in the system are Yuval Pedual, Mark Kampe, Dennis Mumaugh, John Codd, Bob Greiner, Greg Noel, and Peter Neilson.

This document is built on the shoulders of two earlier documents by Steve Holmgren and Greg Chesson. Major portions were contributed in an earlier draft by Jody Kravitz. The latest contributors have included Jay Goldberg and Richard Balocca. My own role throughout this project has been one of questioning, prodding and cajoling for software, explanations of how it works, and pieces of documentation. Responsibility for the final collection of the document parts, filling in the gaps, checking the content against the code, etc., has fallen to this author. He similarly bears responsibility for any errors of commission and ommision which remain. As they have all told me for years: "When in doubt, read the code; it is the final word on what is happening."

-- K. Kelley

THE UNIX NETWORK CONTROL PROGRAM

1 INTRODUCTION

The Unix time-sharing system, developed at Bell Telephone Laboratories for the Digital Equipment Corporation (DEC) PDP-11 computer (with memory management) as well as the Interdata 8/32, has been installed at hundreds of 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 Advanced Research Project Agency's Network (ARPANET) by adding a Network Control Program (NCP) to the standard Unix system.

The architecture of standard Unix simplified many aspects of the design of Network Unix. Among the architectural features that have aided the design are the modularity of Unix, both within the resident portion of Unix and between user processes. As a result the Unix NCP enjoys several properties which are not often found together in a single networking executive:

(1) the system works with a variety of network hardware interfaces

(2) it is not limited to operation on the ARPANET alone

(3) the resident core overhead is low--about nine thousand bytes

(4) protocol state machines are implemented by a natural mechanism--thus tending to be easy to maintain

(5) user interfaces to the NCP are clean and simple (only one system call added (sendins) and one system call (open) expanded)

(6) the NCP is written entirely in the same high-level language as Unix

(7) the NCP can be used as a basis for connecting multiple Unix systems together as a mini-network.

The function of this document is to describe the Illinois implementation of the NCP. Section 2 will provide an overview of important terms for Unix and of Arpanet concepts. Section 2 also will summarize the NCP. The casual reader will perhaps want to stop there.

Section 3 and Section 4 in turn will describe, in more detail, the kernel and user memory portions of the NCP. Section 5 addresses communications between these split portions of the NCP. Section 6, with the exception of the material about opening a network file, should not be attempted without a copy of the actual source code in hand. It examines indivdually a number of important aspects of the implementation which need to be traced through the code for complete understanding.

2 TERMINOLOGY AND OVERVIEW

This section introduces terminolgy and the conceptual framework of Unix and the ARPANET. It also provides a summary of the structure of the Network Control Program (NCP).

2.1 Unix

The operating system uses the standard DEC address relocation hardware to partition the physical memory into *kernel space* which is reserved for the resident portion of Unix, and multiple *user spaces* which are available to user processes. Each user process has its own address space. It does not overlap with any other user process, nor does it overlap with the kernel space.

The Unix kernel space is divided into a system proper and a set of *device drivers*. Device drivers are usually used to perform Input-Output (I-O). Each device driver is assigned a *major device number* and one or more *minor device numbers*. If a device driver is to be multiplexed among many sub-devices, such as disk spindles on a disk controller, the particular target is distinguished by the minor device number. Most communication between the system proper and a device driver is done via procedure calls indexed by major device number. The device numbers are also passed as parameters to these procedures, allowing the minor device number to be used by the driver to select a particular target.

The file system and directory structure are implemented by an indexing system. A directory entry contains only a name for the associated file and an *i-number*. The i-number indexes into a *i-list* of *i-nodes*. An i-node contains most of the attributes of the file. (Note that the file name is *not* one of these attributes and therefore multiple names can be mapped to a single i-number.) In particular, an i-node indicates whether a file is a *directory* file (containing file names and pointers to i-nodes), an *ordinary* file, or a *special* file.

As a result of this indexing system the directory structure of the Unix filing system is a directed graph with a designated root. A *pathname* is an ordered list of file names (all but the last, *directories* of file names) that makes up a path from the root of the file system to a particular file. File names in a pathname are separated by /. For example, /usr/greg/f specifies file f in directory greg, which is a subdirectory of usr, which is a subdirectory of the root. When a user *open* s a file he uses its *pathname* and the system provides him with a *file descriptor* (an integer). He specifies that file descriptor when performing read and write operations.

In Unix, special files map a character string representation of physical device names into their assigned integer names within the system--the i-nodes of special files contain the device number. User level processes can designate device drivers only by special files. 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 ("terminal number 4"). 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 i-nodes.

The user's command interface to the system is a program called the *shell*. The system *forks* (i.e., creates a *process* that is a copy of) a shell process for each terminal logged onto the system. Commands, interpreted by the shell, usually execute programs for the user.

The *signal* system call allows a user process to specify response to program and external exceptions. For example, there are signals for illegal instruction, loss of carrier on a data connection, and so on.

The special file, device driver, and signal mechanisms mechanism of Unix were used in the treatment of network I-O.

2.2 Arpanet

The ARPANET is composed of an IMP-subnet that performs the bulk of data transference and a set of *hosts*, which drive the data transfers. The IMP-subnet is named after the Interface Message Processor (IMP) which is a computer that performs packet switching for the ARPANET. A host is a computer attached to an IMP. Each host is assigned a unique *host number*. The host numbers are 16 bits wide. Most can be represented by 8 bits. (Unix uses the shorter representation.)

2.2.1 Protocols

A *protocol* is a set of conventions that allow entities (hosts) to cooperate. In this case, the ARPANET protocols specify the form, content, and interplay of messages that are exchanged between the various elements of the network. [1] The ARPANET makes use of a hierarchy of protocols, more complex ones building upon the more primitive. These protocols are related to the various logical levels of data transfer between hosts: transfers between IMP's, transfers between hosts, communication between software processes, file transfer between systems, and so on. These are outlined in the following paragraphs, beginning with the lowest level.

2.2.1.1 IMP-IMP (zeroth level protocol)

The low-level, IMP-subnet protocols do not directly affect the NCP design since they are transparent to a host system. They will not be discussed here.

2.2.1.2 IMP-Host (first level protocol)

The IMP and host communicate through an IMP interface. For the purpose of protocol definition, the IMP interface is merely a data path.

The unit of transmission between a host and an IMP is a *message*. A message may consist of up to 8095 bits. A *message header* is the first few bits of a message. [2] It specifies, among other things, the destination host and a *link number*.

----

[1] See References 2, 3, and 4 for details of the ARPANET protocols which served as the basis for this implmentation.

[2] The number of bits in a message header is either 32 or 96. The 32 bit header carries the short host number and the 96 bit header carries the long header. Unix uses the short (or "old style") header.

The IMP-host protocol has as its vehicle the message headers. Control bits in a header indicate whether or not the header is followed by additional data (up to the maximum message size). If there is additional data, then the header plus the data constitutes a *regular message*. If there is no additional data, then the header is an IMP-host control message.

The most commonly occurring IMP-host control message is the Ready-For-Next-Message (RFNM). A RFNM is a positive acknowledgment from a distant IMP indicating that the distant IMP has begun copying a regular message from the local IMP. All other IMP-host messages are diagnostic in nature.

The link is used to demultiplex messages entering a host into 256 possible channels. Link zero is assigned as the *control link* and is 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 reserved.

2.2.1.3 Host-Host control link protocol (second level protocol)

This protocol, which is responsible for forming data paths, rides on the control link.

Regular messages on the control link are used for opening and closing connections between hosts as well as for performing several auxiliary functions such as Interrupt Sender (INS).

2.2.2 Sockets

The NCP uses the link number to route data. Within a single host system the data path taken by a (link tagged) message to a process is uniquely identified by a 32 bit number known as a *socket number*. The association itself will be called a socket. The host number, socket number pair will be called the *full-socket* to distinguish it from the 32 bit socket number. A full-socket is 48 bits long. (40 bits in the short form.) Sockets are *simplex* (they define a data path with only one direction). Even numbered sockets are, by definition, data paths that receive data from the net ("read" sockets). Odd numbers designate "write" sockets. A *connection* is a unidirectional data path between processes identified by two full-sockets (one read full-socket and one write full-socket) and a link.

2.2.3 Connections

The Request-For-Connection (RFC) message is the Host-Host protocol message exchanged between hosts for the purpose of forming a simplex connection. There are actually two RFC commands, one for a prospective receiver and one for a prospective sender. 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 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 host matches a pair sent to that same host, then the connection is considered to be open. There are mechanisms for initiating a connection to another host and mechanisms for "listening" for other hosts to initiate a connection.

Certain fine points in this process have been ignored: such as timeout and queuing policies to be observed during the connection process; what part of the protocol defines the byte size to be used in subsequent data transmissions over the connection; deadlock problems; and so on.

The RFC commands for setting up simplex connections are used by higher level protocols (see 2.2.4 and 2.2.6 below) to establish more complex connections.

2.2.4 Initial Connection Protocol

The ICP, as it is called, is the standard ARPANET mechanism for opening a bidirectional data path between two processes. The ICP uses the host-host protocol to establish a pair of simplex connections (composed of *two* read full-sockets, *two* write full-sockets, and *two* links) between hosts. This pair is referred to as a *duplex* connection.

2.2.5 Flow Control

NCP's are required to maintain a *message counter* and a *bit counter* for every simplex connection. These counters are initially zero. No data can be sent over a connection until the receiver sends an *allocate* (ALL) command to the sender. The ALL 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 ALL's to the sender. This technique guards against the possibility of a fast sender overrunning a slower receiver with data.

2.2.6 Higher Level Protocols

The Telnet protocol is one of many protocols that makes use of the ICP to establish a duplex connection between a pair of processes. The most common use of the Telnet protocol is to allow a user at a terminal on one host to log on to a foreign host's time-sharing system as though his terminal were attached to the foreign host. This is accomplished by using the ICP protocol to connect from a Telnet *user* process on the local host to a Telnet *server* process on the foreign host. The Telnet protocol itself intermixes data and control commands over the duplex connection.

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 another, simplex connection (a "data" connection), transferring the file over the "data" connection, and then closing the "data" connection when the transfer is complete.

The ARPANET mail protocol rides on top of FTP in order to transfer "memos."

Certain 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 [Reference 4], existing network RJE implementations are local adaptations.

There exist numerous experimental or proposed higher level protocols. Examples include schemes for random file access (as opposed to file transfer), interactive graphics, cross-process procedure calling, and inter-network communications, i.e., communications between ARPANET and other 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 document.

2.3 NCP Structure

The Network Unix system is a standard Unix augmented by a Network Control Program (NCP) Kernel module (refered to as the ncp-kernel), a constantly running user level process or *daemon* (refered to as the ncp-daemon), network special files, a kernel-daemon special file, and other user level service processes. The ncp-daemon implements the host-host and ICP protocols. The ncp-kernel services the IMP, the ncp-daemon, and user programs. All higher level protocols are implemented by network service programs which execute in user space, utilizing the ncp-kernel and through it the ncp-daemon. The ncp-kernel (about 9 kilobytes) is the only resident portion of the NCP. The ncp-daemon (about 22 kilobytes) and other programs are only brought in to memory on demand. Since the ncp-daemon is primarily needed for opening and closing network connections, and since the ncp-kernel manages network data flow, this "split" organization improves memory utilization without sacrificing performance. Because of this split design, Network Unix requires very little additional memory over standard Unix.

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 are overlayed upon standard Unix structures: specifically i-nodes, file table entries, and kernel buffers. The ncp-kernel uses existing Unix procedures for managing these structures.

Network special files provide the basis for the interface between user programs and the NCP. 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 a PDP-9 front-end in London, 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.

2.3.1 User Program Interface

Communication between user programs and the user service routines is accomplished through the existing Unix system call mechanism. Once a network special file has been *Opened* programs access the network software by applying standard Unix I-O system calls (Close, Read, and Write, and so on) to the respective *file descriptor*. Thus, the network is accessible through any language that provides the standard interface to the file system.

The purpose of these system calls applied to network files is just what one would expect. That is, the Open system call establishes a connection between the calling program and a process on the foreign host. Read and Write system calls transfer data between the two processes, and a Close terminates the connection.

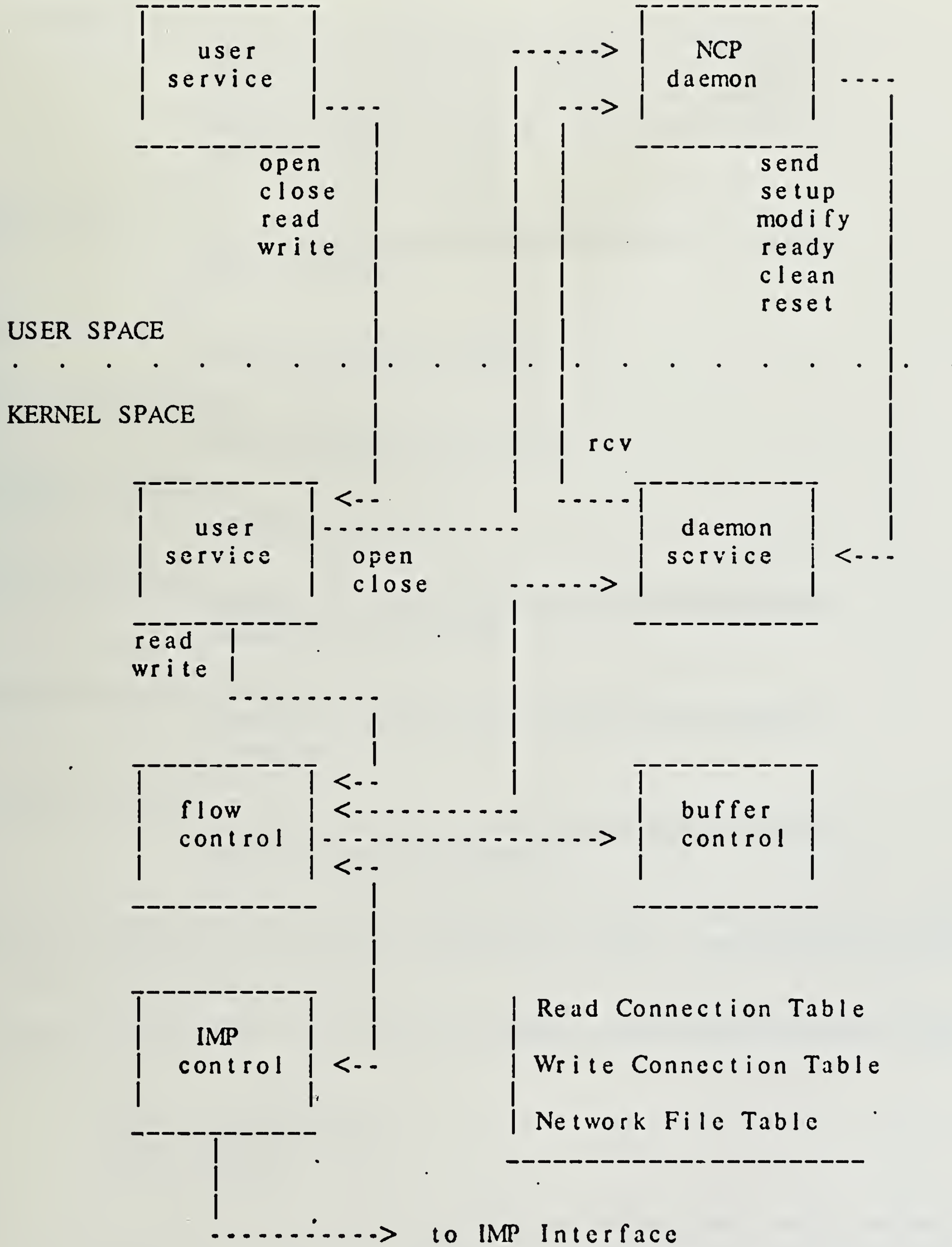
The form of the Open system call is (as available in the C language):

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

The Open system call returns a file descriptor (fd>=0) if the connection is opened successfully, and a 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

Figure 1

NCP Organization



the value of "mode" is 0, 1, or 2. However, any other value is interpreted as the address of a parameter structure in the user program. The fields in this structure, which are refered to as the *open parameters*, are copied into the kernel. The parameters and their meanings are introduced below: (But see Section 6.3 for more detail)

o_type (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.

o_id (file id)
: file descriptor used when the open refers to an already open network file.

o_lskt (local socket)
: refers to a local socket number.

o_fskt (foreign socket)
: specifies a foreign socket number.

o_frnhost (foreign host)
: specifies a foreign host.

o_bsize (byte size)
: specifies the connection byte size (Unix requires that this number be a multiple of 8).

o_nomall (nominal allocation)
: specifies the nominal size of an allocate command sent to a foreign host.

o_timeo (timeout)
: the time in seconds to wait for the foreign host to fulfill the request before cancelling it.

If any fields in the parameter structure are zero, the NCP will use default values in their place. The flexibility afforded by the open parameters greatly simplifies (and in some cases makes possible) the task of implementing higher level protocols.

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

```
nbytes = read(fd,bffr,cnt);
nbytes = write(fd,bffr,cnt);
status = close(fd);
```

In each of these calls fd is a file descriptor returned by an Open call, bffr is a buffer address, and cnt is the number of bytes requested for transfer. The returned value, nbytes, is set to the number of bytes

actually transferred, and in all cases a minus one is returned on an error and a zero returned on end of file (closed connection).

An ARPANET file behaves like a Unix terminal device: it may return a smaller number of bytes than requested.

The Unix signal mechanism was pressed into service for the purpose of handling incoming interrupt signals (INS):

signal(SIGINS,action)

Outgoing interrupt signals are generated with a new Unix system call:

sendins(fd)

User I-O calls on a network special file are detected by four conditional statements ("hooks") that have been added to the standard Unix file system code. These four statements are the only changes to standard Unix code required by the NCP--and all they do is check for I-O system 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 data transfer (Read and Write) requests are processed in the ncp-kernel. Communication between the ncp-kernel and ncp-daemon is implemented by a special file (/dev/ncpkernel). Unlike the network special files described in 2.3, the ncp-kernel file has a normal Unix major device number and is *not* a network special file. However, the device driver for /dev/ncpkernel is actually the body of the ncp-kernel. That is, Unix is set up so that Read and Write system 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.

2.3.2 Kernel Functions

2.3.2.1 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 i-nodes are used to represent the state of sockets (i.e. "one-half" of a connection). Each standard Unix file control block can point to as many as three of these sockets. (These data structures will be refered to as sockets, since they are the data elements that embody sockets--this is in keeping with the "tradition" of the designers of the Unix NCP, and hopefully, no ambiguity will result.) The sockets contain message and bit counters, 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 i-node, the foreign host number and a link number.

The network file table contains a one word pointer to the socket i-node 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 identified by their index number in the file table. The sizes of the file table and connection tables are compile time constants. [3]

2.3.2.2 Buffer Control

The buffer control section of the ncp-kernel manages a pool of 64 byte buffers that are chopped from several of the 512 byte buffers allocatable by standard Unix. The ncp-kernel will request a limited number of 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 may occupy several of the small 64 byte buffers, of course. The number 64 was chosen as a compromise between requirements for connections that would make extensive use of single character messages (typically Telnet connections) and connections that would require larger message sizes. Experience has shown that the average message size for "large message" connections appears to be about 300 bytes. Thus the 64 byte buffer is an effective compromise to the problem of fragmentation.

2.3.2.3 IMP Control

The IMP control section handles the IMP-host protocol and other mechanics of transferring data between the host and IMP (i.e. the IMP device driver).

2.3.2.4 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, (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) when a process issues a write on a network file he will not return until either the entire count of data is sent and acknowledged by rfnms or the file closes. The NCP writes the data to the network in pieces as large as the current allocation allows and always waits for a rfnm before trying to send the next piece.

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

----

[3] The current implementation allows for 32 connection table entries and 16 open network files.

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 most needs, the algorithm which is compatible with both 11/40's and 11/45's was preferred over a faster algorithm which would not be compatible.

2.3.2.5 Servicing of Daemon

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. The specific commands which the kernel must handle for the daemon are the daemon outputs described in Section 2.3.3.1.

2.3.3 Daemon Functions

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 system calls 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 indictes which function to call for a given configuration, and the next state is determined by code in each function. The operation of the "socket machine" is described in Section 6.4.

2.3.3.1 Daemon 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 (i-node)
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
FRLSE	release a network file
TIME	timeout

2.3.3.2 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.
sent	an array of counters of the number of buffers in the current outstanding messages to each host.
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.

2.3.3.3 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 table entry, 3 socket i-nodes,
             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;
     }
  }
```

3 THE NCP KERNEL

In this section we address in more detail the kernel portions of the network control program. We first look at the parameter files and the important structures called sockets, network files, connection tables, and the network file table. We then discuss the procedures which deal with connection establishment in the kernel and describe flow control for data in the kernel. This is followed by a summary of the sleep and wake-up signaling which could be thought of as a control structure for the set of kernel activities. Finally, the procedures used to drive the IMP and to initialize the network are discussed at the end of Section 3.

3.1 Kernel Parameter Files and Data Structures

3.1.1 Kernel Compile-time Parameters

Definitions and declarations of parameters and data structures for the NCP are kept, for the most part, in a set of include files prefaced by the string "net_". [4] These used to be kept in a separate directory "h/net". They are now kept in /usr/include on most systems.

Within limits (which are not well-defined) the system installer could adjust some of these parameters to make kernel memory demands smaller or larger. This should not be done lightly.

The strictly network related include files for the NCP kernel are:

contab.h
hosthost.h
ncp.h
netbuf.h
netopen.h

Table 1 specifies the compile time parameters for the NCP.

3.1.2 Kernel Network Sockets

The Unix file system contains two types of information structures that provide a frame of reference for the Kernel with respect to a standard disk file. These two structures are the inode and network socket structures.

In general terms, an inode serves to contain such information as the absolute length of the file, the related data segment locations, the owners id, last access date, the particular device associated with its data, and finally some indication as to what type of file, whether Special, Data, Directory, or Net.

---

[4] This material is included here only for a sense of completeness. The first time reader can skip the following paragraphs.

Table 1

NCP Compile-time Parameters

Parameter	Value	Meaning	Include file
CONSIZE	32	Number of entries in connection table	contab.h
FILSIZE	16	Number of network files	contab.h
NETPRI	2	Soft priority for sleeps	net.h
NOMSG	10	Nominal message allocation	net.h
B_OVERHEAD	4	Bytes of overhead in a netbuf	netbuf.h
NET_B_SIZE	60	Bytes of data in a netbuf	netbuf.h
kb_hiwat	10	Number of kernel buffers useable by NCP	netbuf.h
net_b_per_k_b	8	number of netbufs to one kernel buffer	netbuf.h

There is an eight word array within each inode which contains the disk addresses for normal files. Since a Special File Inode is really the basis for a mapping between a name and a device number, there are no associated data blocks, and the eight word array is unused. It is this fact that allows for the storage of network information in an inode marked Special. As far as the rest of the system is concerned this portion of the inode structure contains information pertaining to a device. In the net software, this array contains what is referred to as a socket.

A socket has the necessary information for building the various leaders needed for network communication, and for holding allocation information. The socket structs and their flags are declared in the include file net_net.h.

Sockets are initialized in two forms; a Read socket containing the information needed to handle data coming from the network, and a write socket to handle data to be sent to the network.

3.1.2.1 Read Socket Format

word host&link
word byte size
word flags
word messages allocated
word bytes allocated
word message handle
word message size
word reallocation parameter
word reading process

Host&Link

The upper byte contains the number of the foreign host related to this simplex connection. The lower byte contains the specific link number used in communication with the host.

Bytesize

Bytesize contains the number of bits associated with the logical bytesize of the connection. For example, if bytesize was 32, there would be 32 bits to every logical byte sent by the foreign host. This would mean that UNIX would then understand that there is really four times that number of PDP-11 bytes in any message received.

Flags

Flags provides a bit box for the inter-communication between a user initiated Kernel procedure and the NCP Daemon.

Message Allocated

Messages allocated is the number of messages that UNIX has told the foreign host that it may send before it must wait to receive further allocation.

Bytes Allocated

Bytes Allocated is the number of eight bit bytes allocated to the foreign host. It contains the same implied flow control information as does Messages Allocated above.

Message Handle

Message Handle is a pointer to a linked list of buffers containing data from the foreign host. This data is awaiting a user read to finally reach the process in communication with the foreign host.

Message Size

Message Size contains the total number of bytes waiting in the message pointed to by Message Handle.

Reallocation Parameter

In order to keep a constant flow of information coming from a foreign host, for ultimate reception by a user process, the Kernel is constantly sending allocations to the foreign host. To do this, it must have some idea of how many messages and bytes to re-allocate after the user has taken delivery of some data. This is the maximum number of bytes and messages to re-allocate. Hence, the difference between the number of bytes and messages allocated and this parameter is the amount of messages and byted to re-allocate.

Reading Process

The process id of the reading process is put here by nrdwr.cnetread(). This is done in case it becomes necessary to send a signal to the process to notify of network interrupts.

3.1.2.2 Write Socket Format

word host & link
word bytesize
word flags
word messages
word high bit allocation
word low bit allocation
word file id
word tty
word writing process

Host & Link, Bytesize, and flags have the same meaning and function as with a read socket, although the meaning of individual flags are somewhat different. [5]

Messages
: Messages denote the number of messages allocated by the foreign host, and hence the number of messages that may be sent before having to wait for further allocation.

High bit & Low bit allocation
: Both High and Low bit is a denotation of the number of bits that may be sent to the foreign host before waiting for more allocation. This is indirectly the number of eight bit bytes that may be sent. It is stored in this manner because the Allocate command received from the network is in bits, and the remote host is not constrained to do its re-allocation in bytesize bits.

File Id
: File id is the address of the particular network file. It is in fact a pointer to a network file structure as described below.

Tty
: This element was added to hold a pointer to the user's tty for an earlier implementation of server telnet. It no longer has any function.

Writing Process
: As in the read socket, this id of the writing process allows signals to be sent to the process in case of network interrupts.

---

[5] The reader interested in detail with respect to these flags is referred to net_net.h.

3.1.3 Structure for a Network File

A second structure commonly used by UNIX for maintaining order in the available disk storage is referred to as a file.

In the normal sense, a File structure is obtained for each user as he opens a file. This File structure contains information as to its specific type, (Pipe, Network, or Standard Data), a count of the number of processes from the same family that have the file open, and a pointer to an Inode containing information about the file's contents, location, and size. Finally there is a two word offset into the file itself. This is used for sequential reads and determines which physical byte will be returned to the user upon his next read.

In fact it is indirectly a pointer to this File structure that is referenced by the file descriptor specified when a user does a read, write or close.

In the Network sense, the first two parts of the structure have the same meaning.

Throughout the life of a network connection, a file may be associated with up to three sockets. For a large portion of the time only a read and write socket are in use. For a short interval, during an Open, there is also an Initial Connection Socket used to carry on the Initial Connection Protocol. Once the connection is open, this socket is closed and de-allocated.

Again, since there is no local data associated with a network file, the last two word field, the offset, may be used for network purposes. In fact, the last 3 words of the file structure are used as pointers to sockets (Network connection inodes).

3.1.3.1 Network File Structure

byte flag
byte count
word read socket pointer
word write socket pointer
wcrd ICP socket pointer

This represents the total reusage of UNIX structures for network purposes. It is felt that this type of careful piracy allows the network software to utilize not only existing software for management of the structures, but it also results in a savings of data space. Since, at any given instant, a structure may be used for network information, and later for the storage of standard file information, system's data space is used dynamically, according to the particular needs of the current mix of user programs.

3.1.4 Connection Tables

There are two tables that were added to the system specifically for network purposes. They are the Read Connection Table and the Write Connection Table. In principal they are used to route data and control information to user programs.

3.1.4.1 Read Connection Table

When data arrives from the IMP, a message must be created and it eventually must get routed to a user socket. To determine which socket is being referenced by the incoming data, there is a table that contains the Host & Link of each Read connection and an associated socket pointer. This table is searched by the IMP device driver as messages come in, to locate the read socket referenced by the data.

3.1.4.2 Write Connection Table

The Write Connection Table contains the same mapping information as the Read Connection Table, but its contextual use is different. When control information is received from the IMP, a specific class must be routed to the user write socket. This information affects the amount of data that may be sent from the user process to the foreign host before further control information is required. When this information is received, its host and link are looked up in the Write Connection Table to return the address of the associated write socket.

Format of the Connection Table Entry:

word host & link
word socket address
word local socket number
word high bits of foreign socket number
word low bits of foreign socket number

Each of these tables is 160 words long at present (room for 32 network connections), however, their size is a compile time option.

3.1.5 Network File Table

There is one further table dedicated specifically to network use, called the Network File Table. It contains the addresses of all Network files in use at any one time. Each entry is one word in length, and it is used to check file ids associated with NCP Daemon requests for validity. The table is 16 words in length and like the connection tables, its size is a compile time option. [6]

----

[6] The parameter specifying this table is FILSIZE in net_contab.h. It should match the size of "nfiles" in ncpd/files.h. However, for a long time this implementation had FILSIZE=18 and nfiles=16 with no apparent ill effects.

3.2 Connection Establishment in the NCP Kernel

3.2.1 User Program Interface

User programs open connections by opening a network file with the standard Unix open system call. Connections are closed by closing the associated network file. The major events that take place in the kernel are as follows:

netopen

The netopen procedure is invoked by Unix upon receipt of open to network file. It allocates a File structure (see Section 3.1.3.1 above), sets up the open parameter structure (either with default values, or with values extracted from the caller's space), places the open request on the ncp daemon's "rcv" queue, and sleeps on a wait for the daemon to issue a "ready" instruction. When netopen is waked up, if the open was successful, and if there is a read socket, the allocation counts and flags in the socket are set. If there is a write socket, the flags are set. Duplex connections have both types of sockets, but simplex connections would only have one or the other.

netclose

The associated sockets (inodes) are freed (daecls,daedes) and the file reference count is decremented (net_frlse).

3.2.2 NCP Kernel Interface to NCP Daemon

The daemon issues writes to /dev/ncpkernel at various points in the process of opening a network file, specifying the opcode appropriate to the situation. The daemon issues reads to /dev/ncpkernel when it wishes to process information sent it by the kernel. Information which it reads often has an effect on the opening of a network file. The major routines involved in setting up and maintaining connections are as follows:

to_ncp, ncpread

The first is used to queue up data for the daemon. The second is used to give it the data.

wf_ready

This is the process that handles the wakeup from the wait in netopen (see Section 3.2.1 above). It informs the netopen routine as to the outcome of the open request.

wf_mod

This procedure modifies socket information. Specifically, it sets values of "data to go to ncpdaemon" and "socket is open" status bits, the byte size, host, link, and initial allocation values (wf_inc_alloc). Then it places host, link, local skt, and foreign skt information in the associated connection table entry (wf_update_skt).

wf_setup

This procedure sets up data structures to describe a connection. It gets an inode to hold the socket and sets status flags, byte size and allocation values as specified by daemon. It also allocates a connection table entry and place socket information in it (wf_update_skt).

wf_clean

This procedure cleans up data structures associated with a connection when either the daemon is done with the socket or a host is reset (iclean). This will either cause the inode to be destroyed (daedes) or will cause the kernel to tell the daemon the socket is closed (daecls). The daemon can later tell the kernel to destroy it when the daemon is done with the connection. Also, if a user is sleeping on this socket, that user is waked up.

wf_send

Send ncp daemon data into the network (sndnetbytes).

wf_reset

Cleans up all data structures containing information on the specified host. Clean up all sockets (inodes) that were associated with this host (iclean). The sockets are located by following chain from file table to the File structure to the sockets themselves.

wf_frlse

Decrement the use count for a network file, perhaps causing its release (net_frlse).

Other Procedures

An incomplete list of other routines involved: wf_inc_alloc, wf_update_skt, sendnetbytes, daecls, daedes.

3.3 ARPA Network Data Flow within the Kernel

The intent here is to describe in general terms what actually happens within the kernel when various Arpa network related I/O functions take place. The focus of this description will be the various input ports, output ports and queues of the system.

3.3.1 User Writes

When a user *write's* to a network file, a string of procedures decide what allocation considerations are in force, and how many bytes may be sent to the foreign host in in each actual network write. Due to network allocation considerations, a single user *write* may result in several network transmissions. An IMP to Host leader and a Host to Host leader are built from the socket information. Then a message primitive is invoked that copies user data to the kernel, into a message suitable for network transfer. As the data is copied in, buffers are linked to the end of the chain and the "handle" is updated to reflect the new end. Once a message is built, it is linked into the IMP device driver output queue for eventual transmittal to the network.

3.3.2 User Reads

When a user does a network *read*, a message queue is located in his Read Socket. If that queue is zero, he waits for data to arrive. If there is a message, a message primitive is invoked which copies data from the head of the queue into the User's address space, returning any newly freed buffers to the NCP's buffer Pool.

3.3.3 IMP Device Driver

In order for network data to be routed to a User Read Socket in the form of a message, the IMP device driver must build a message as it comes in from the network. There are two approaches that might be taken both resulting in a message for the user. First, a system may declare a fixed area large enough to hold the maximum size network message (1024 bytes for the ARPA Network) and then copy the useful user data into a message and link it to the read socket. Second, a system may take advantage of that fact that the IMP interface will interrupt on "buffer full." This means that the input side of the interface will stop transferring bytes from the IMP when it sees that the next byte will overflow the bounds of the area given it as available for data, even though the whole message has not been transferred. With this feature, a full message can be transferred to the PDP-11 in a series of interface activations. The advantage of the first method is that very little code is needed to manage an input transfer, one simply loads the interface, and waits for a "done" interrupt. The second method requires code to check that the transfer is finished, and if not, to obtain another area and load its address into the interface. The tradeoff is the amount of code required to handle the partial transfers and the extra interrupts versus the fixed 1024 byte area. However, as a byproduct of the message system, there exists a primitive to get a single message buffer (of 60 bytes). Thus there is no extra code required to procure the data areas needed. The code to check for multiple interrupts requires only a single "if" statement to check a bit in the IMP status register. On this basis then the second method was implemented. This single design decision is one of the primary determiners of the "nature" of the Unix NCP.

The actual sequence of transactions is as follows. The first eight bytes of an IMP to Host or Host to Host leader is read into a fixed location in the kernel. The contents are inspected to see if the data pertains to a user or to the NCP Daemon. If the leader indicates this data is a control message, (link field is zero) it is handled by the NCP Daemon. No lookup in the connection table takes place, but an appropriate flag is set. If the leader indicates the data is for the user (link field is non-zero) his read socket address is located in the Read Connection Table. The first buffer for the message is obtained and its address is applied to the IMP interface. (It should be noted that most network messages will fit into one message buffer. A multiple buffer transfer will be presented for completeness.) A sequence of "buffer full" interrupts is taken until the whole message has been transferred. Each of the interrupts results in the appending of a user message and the acquisition of a new buffer. When the message is complete, it is concatenated to any message already in the user's read socket.

In this case, the message with its composite buffers forms a flexible queue of data waiting for the user to do a network *read*. It is flexible in the sense that each message may be from one to 1024 bytes in length and the "containers" will in the worst case waste 59 bytes of unused area; whereas use of the large system I/O buffer would result in a worst case of 511 unused bytes.

Control messages, with the exceptions listed below, are sent to the NCP daemon.

3.3.4 IMP Flow Control Activities

There is a class of data control associated with the ARPA Network protocols referred to as "flow control". This is a mechanism whereby a receiving host may place an upper bound on the size and number of messages received from the sending host. This function is embodied in the ARPA Network Allocate command. This command is transmitted by a receiving host to a sending host to inform him of how much data will be accepted. These allocate control commands and the data itself form the bulk of the traffic passed between two corresponding hosts.

After the IMP to Host or Host to Host leader has been read into its fixed location, a key determination is made. If the message is a data message destined for a user, instruction sequencing continues as described above. If the message is a control message (the IMP to Host link field is zero), it must be searched for Allocates referencing various users local to UNIX, all of whom may be involved in sending data to the foreign host. If the control command is an allocate, the host and link fields are used to access the write connection table. From this the appropriate user's Write Socket is found, and his allocation bits are updated. The user is then notified that an Allocate has arrived so that he may continue sending data up to the restrictions of the new allocation information. Any remaining control protocol is then directed to the NCP Daemon.

3.3.5 Imp Device Driver: Output Side

The output side of the IMP device driver is considerably less complex that the input side. As messages are generated for output to the Network, whether they be user or NCP Daemon originated, an I/O header is initialized and linked to an IMP output queue. If the output side is inactive, it is started.

As application of the output buffers continues, a bit is checked to see whether the container in question is a standard I/O buffer or a network message composed of small 60 byte buffers. If it is a network message, each of the buffers is transmitted before the next I/O header is obtained from the queue. The output side continues this cycle while there are headers in the queue. Once empty, no further action is taken until some party wishes to transmit.

3.4 Sleep-Wakeup Signalling in the Kernel

Calls to the sleep routine are used extensively within the NCP kernel to cause a process to be suspended until it has information to deal with or resources to use. The sleep routine takes two arguments: a unique value on which to sleep, (called the channel), and an optional priority. It causes the routine executing the sleep to be suspended until waked up by some other routine. This is done by storing the process id of the calling routine along with the unique value. The wakeup routine has one argument: the unique value identifying the sleep call to which he corresponds. Wakeup searches for this value in his table of information provided by sleep calls. If he finds it he sets the corresponding process ready to run. Whether he finds it or not he always returns control to the calling program without actually forcing a process switch. Since the waked procedure has now been set to run, it will run the next time its turn comes up in the schedule.

The unique values which are used to match the sleeps with the wakeups are generally procedure addresses or addresses of structures such as sockets, file pointers, or even pointers to unique resources like a buffer freelist pointer. The important thing about the priority is that processes sleeping on negative priority will not be disturbed by signals other than the event they are waiting for. If the priority is positive (greater numbers mean lesser priority), the sleep may be disturbed by other signals and the sleeping process must arrange to check when awakened to be sure that what he is waiting for really happened, or to react to other conditions.

In the sections which follow we summarize the sleep and wakeup cycles for the kernel. Usually there is a single wakeup for each sleep, but in some instances several events need to cause the wakeup.

3.4.1 Imp Input

impopen

A sleep(&imp,-25) done to put imp input process to sleep until there's input to process.

imp_init

wakeup done upon occurrence of imp input interrupt.

ncpclose

wakeup done upon closing of /dev/ncpkernel

3.4.2 NETWORK BUFFERS

getbuf

sleep(&net_b.b_freel,NETPRI) done when getbuf can't get a netbuf for the requesting process.

freeb

wakeup done when a network buffer is freed and "proc_need" flag in net_b is set. The wakeup means "when you wakeup, see if there's any free netbufs...if not go back to sleep".

3.4.3 Ncpdaemon Input

ncpread

sleep(&ncprq,1) done to put ncpdaemon to sleep when it tries to read a kernel command from an empty kernel/ncpdaemon queue.

to_ncp

wakeup done when kernel puts something in the kernel/ncpdaemon queue.

3.4.4 Netfile Opens

netopen

sleep(fp,NETPRI) done to put process which issued the open request to sleep until the daemon tells the kernel that the open is complete. (NETPRI is currently defined to be 2.)

wf_ready

wakeup done when the daemon issues a ready command for the file.

3.4.5 Clear Host/Host Channels

sendalloc

sleep(hostmap + host, NETPRI) done to put process to sleep until the host/host channel over which an allocate is to be sent is clear.

siguser

wakeup done when link 0 RFNM associated with some user's send socket has arrived. wakeup means "RFNM on this host/host channel has come in, but check to see if someone else hasn't taken the channel inbetween the time I did the wakeup and you ran again, and go to sleep again if channel is still busy".

3.4.6 User Reads

netread

sleep(sktp,NETPRI) done to put user process to sleep when its read request has yet to be fulfilled, but there is no data in its input queue.

hh

wakeup done in two cases: first, when kernel is done flushing a msg from the imp, and the msg is associated with the user's receive socket; second, when the kernel is done reading a msg from the imp, and the msg is associated with the user's receive socket.

iclean

in response to call by wf_reset or wf_clean, does wakeup of user process associated with socket to inform user that socket is closed. if the socket was a read socket, this wakeup can match the sleep done in netread.

netclose

wakeup done to let user know a close has been issued if the file ref count is 3 (i.e., two processes have the file open besides the ncpdaemon), and the file has an associated read socket. This is to provide a defacto IPC between the parent and child of user telnet and user ftp. When the parent closes the network file, the child will receive and end of file indication.

User Writes

netwrite
: calls netsl^ep, resulting in sleep(sktp,NETPRI) (netsleep) after sending a message via sendnetbytes. It sleeps until a RFNM, error in data, incomplete transmission, or eof (socket closed) occur.

siguser
: wakeup done when RFNM, incomplete transmission, or error in data occurs.

iclean
: in response to call by wf_reset or wf_clean, does wakeup if there is a user process using the socket to inform user that socket is closed. If the socket is a write socket, the wakeup can match the sleep done in netwrite.

4.8 Allocation Waits

netwrite
: sleep(sktp,netwrite) done to put process to sleep while waiting for an allocation.

allocate
: wakeup done when an allocate for user write socket arrives.

wf_inc_alloc
: Wakeup is done after changing allocation values for write socket. It is called by wf_mod and wf_setup whenever the daemon does setup or mod operation. The only time daemon really changes the allocation values for a send socket is when it processes an ALL for a data (non-ICP) send socket. However, the kernel intercepts and nops (in hh1) all such ALL commands. In any case, the wakeup is done in the kernel whenver a setup or mod for a send socket is issued. Since it seems the kernel allocate routine gobbles all the ALLs that would result in waking the netwrite allocation wait sleep, wf_inc_alloc's wakeups don't seem to ever correspond to a situation where an allocation increase has really occured. The result of its wakeups, then, is that netwrite awakes, sees it still doesn't have an allocation, and goes back to sleep.

iclean
: if iclean is cleaning up a write socket in use by a user process, this wakeup can match the allocation sleep done in netwrite, given that is where netwrite is sleeping (see description of user writes, above).

3.5 IMP Input/Output Processing in the NCP Kernel

3.5.1 NCP Daemon Interface

The daemon gets data by performing a read on /dev/ncpkernel. This causes the next message in the NCP daemon queue to be copied to the daemon's space.

3.5.2 User Interface

The user gets data by performing a read on the affected network file. This causes "netread" to be invoked. It sleeps until there is data present in the associated read socket's input queue, and then copies the minimum of the amount of data present and the amount requested to the user's space. It then sends an allocate for the amount given to the user.

3.5.3 IMP Input Process

This is a genuine Unix Process which is woken up upon receipt of an input IMP interrupt. The major routines involved and their function is as follows:

imp_iint Invoked by Unix upon receipt of an IMP input interrupt. It simply wakes the imp input process (sleeping in imp_open)

imp_open When woken, it calls imp_input to process the imp input interrupt. Then it goes to sleep again.

imp_input Calls appropriate routine to handle the input (hh for host/host message, data or protocol, ih for imp/host message, and flushimp if an illegal leader was read) and asks for next buffer to be read (ihbget) if it wasn't simply imp/host protocol.

hh If end message hasn't been raised, request next net buffer be read (ihbget) or if current message is being flushed, continue doing so (flushimp). When end message has been raised, if we read data (i.e., not host/host protocol) then if its for the daemon, link it to its queue (to_ncp) else link it to associated read socket's input queue. If its host/host protocol, call hh1 to decode it further. Then, start a new leader read (impread).

hh1 Process all host/host protocol. Handles ALL (via allocate) and INS commands. After nooping all ALL commands, if there's anything left, it places the message on the daemon's "rcv" queue.

allocate Updates allocation parameters for affected socket and wakes any users waiting on the allocation.

ih Processes imp/host messages. Specifically handled are rfnms, error in data, and incomplete transmission, by passing the msg type to siguser. All other imp/host messages are put on the ncp daemon's "rcv" queue. In any case, a new imp leader read is initiated (impldrd).

siguser Tell interested parties about rfnms, err in data, or incomplete xmission imp/host. Either the leader is passed to the daemon or the user is woken after settin a flag in the affected socket.

flushimp Call impread to read a netbuff's worth of data into a flush buffer.

ihbget Call impread to initiate reading of next net buffer from IMP.

impread Actually set IMP input registers to cause the next IMP input operation to begin.

3.6 IMP Output Processing in the NCP Kernel

3.6.1 NCP Daemon Interface

When Daemon wishes to send data out to the network, it issues a write to /dev/ncpkernel, specifying a "send" opcode. In the kernel, "wf_send" processes the request. It calls "sendnetbytes" to format the data into an ncp kernel message and call the strategy routine to link the message into the IMP output queue.

3.6.2 User Interface

The user issues a write to the appropriate network file. The sequence of events in the kernel is as follows:

netwrite

Called as device driver write routine, it writes the whole request in the following loop: {wait for allocation; send as much data as allocation permits (sndnetbytes) up to amount left to output; wait for completion (rfnm, error, conn closed); update allocation}.

3.6.3 IMP Output Process

These are the routines that handle formatting data to be output into a form that the imp output routine expects, initiate real output to the IMP, and handle output interrupts.

sndnetbytes

Given a vector of bytes and a byte count, it builds a leader (if not already built; the daemon will build its own), and then constructs an output message in the format expected by the output routines (consisting of a buf.h hdr, the leader, and the rest of the message built as a chain of network buffers). It then calls the output strategy routine, passing it the completed message.

impstrat

It links the message onto the IMP output queue and, if the IMP output side is not active, starts the next output to the IMP by calling imp_output.

imp_output

It actually sets the IMP output registers to initiate real output to the IMP. It causes the next network buffer in the current output message to be output to the IMP.

imp_odone

This is the IMP output interrupt handler. If there is more data to output, it calls imp_output to send out the next network buffer (either the next buffer in the current message, or the first buffer in the next message) in the IMP output queue. It updates all the appropriate queue links too.

3.7 IMP and NCP Initialization in the NCP Kernel

The ncp is initialized when the daemon opens /dev/ncpkernel. It is cleaned up when the daemon closes /dev/ncpkernel. The following routines are involved:

ncpopen

Called when /dev/ncpkernel is opened, it initializes buffers and calls impopen to do the rest of the initialization.

impopen

Forks the imp input process, starts strobing of host master ready (set_hmr), and calls imp_init to initialize the IMP.

set_hmr

Strobes host master ready at 1/6 second intervals. This procedure is used only for the old Illinois Imp Interface.

imp_init

Resets the IMP interface, sends 3 nops to the IMP, and calls impldrd to read in the first imp-host leader pair.

ncpclose

Invoked when user closes /dev/ncpkernel. It frees the buffers allocated to the ncp daemon "rcv" queue and calls imp_dwn to perform the remainder of the clean up.

imp_dwn

Frees all buffers currently in the IMP output queue or in any of the input queues tied to specific sockets, and calls ncp_bfrdwn to release all the ncp kernel's buffer storage. ncp_bfrdwn
gives all usurped Unix buffers back to Unix.

4 THE NCP DAEMON

4.1 Daemon parameter files and data structures

4.1.1 Daemon parameter files

files.h

a) defines template for daemon file structure ("file")
b) allocates array of file structures ("files")
c) defines file machine states

globvar.h

a) defines host/host protocol command codes
b) defines host/host ERR command error codes
c) allocates ("hw_buf"), host write assembly buffer

hhi.h

defines imp/host and host/imp protocol command codes

hr_proc.c

a) allocates transfer vector for host/host protocol decoding ("hr_proc")
b) allocates parallel vector of host/host protocol command lengths ("hh_ilgth")

hstlnk.h

defines template for word containing host and link numbers ("hostlink")

ir_proc.c

allocates transfer vector for imp/host protocol decoding ("ir_proc")

kread.h

a) defines template for structure for kernel->daemon open ("kr_open")
b) defines flags for different types of opens
c) defines open error codes to be passed to user
d) defines template for structure for kernel->daemon rcv ("kr_rcv")
e) defines template for structure for kernel-> daemon close ("kr_close")
f) defines template for structure for kernel->daemon reset ("krr_reset")
g) defines values of kernel->daemon instruction opcodes

kwrite.h

a) defines template for structure holding daemon->kernel commands ("kw")
b) allocates a "kw" structure ("kw_buf")
c) defines values for daemon->kernel commands
d) defines socket status bits (for mod instruction)
e) defines kernel socket indices

leader.h

defines template for structure for imp/host and host/host leaders ("leader")

measure.h

defines structure for gathering of statistics ("measure")

probuf.h

a) defines template for structure for a buffer of host/host protocol ("probuf")
b) allocates array to count protocol buffers in current outstanding protocol msg, for each host ("h_pb_sent")
c) allocates array to count number of retrys that have been attempted for current prot msg, for each hst ("h_pb_rtry")
d) allocates array to hold bit map for alive hosts ("h_up_bm")
e) allocates array to hold bit map for outstanding rfnms on link 0 ("rfnm_bm")

skt_oper.c

a) allocates a two-dimensional array of function ptrs (skt operation X skt state) to govern execution of socket machine ("skt_oper")

skt_unm.c

a) allocates vector of function ptrs (indexed by socket operation) for unmatched socket operations ("so_unm")

socket.h

a) defines template for socket structure ("socket")
b) allocates an array of "socket" structures ("sockets")
c) defines values for socket states
d) defines values for socket operations

4.1.2 Daemon Data Structures

file

template for daemon file structure

files[nfiles]

array of "file" structures

hh_ilgth[]

vector of host/host protocol command lengths, parallel to "hr_proc"

hostlink

structure defining position of host and link within a word

host_status

file of 256*2 bytes;status of each host;host0 really imp status

hr_proc[]
vector of function ptrs for decoding host/host.protocol

hw_buf[12]
buffer in which h/h protocol to be sent is placed

h_pbq[256]
h/h protocol queue list heads, one for each host

h_pb_rtry[256]
# of times we've tried to send current h/h msg for each host

h_pb_sent[256]
of "probufs" in h/h msg currently outstanding for each host

h_up_bm[256/8]
bit map of live hosts

ir_proc[]
vector of function ptrs for decoding imp/host protocol

krr_reset
template for kernel->daemon reset instruction structure

kr_buf
a "kr_rcv" structure

kr_close
template for kernel-> close instruction structure

kr_open
template for kernel->daemon open instruction structure

kr_proc[]
vector of function ptrs for decoding reads from kernel

kr_rcv
template for kernel->daemon rcv instruction structure

kw
template for structure for daemon_kernel writes

kw_buf
a "kw" structure

leader
template for structure for imp/host and host/host leaders

measure
    structure for holding gathered statistics

probuf
    template of h/h protocol buffer, to go into circ. linked queue

rfnm_bm[256/8].
    bit map for outstanding rfnms on link 0 (one bit per host)

skt_oper[][]
    array of function ptrs; rows are skt opers, cols are skt states

skt_req
    a "socket" structure which gets current request for skt machine

socket
    template for socket structure

sockets[nsockets]
    array of "socket" structures

4.2 NCP Daemon Communication

The NCP Daemon is a user program just like any other compiled with the C compiler and is run just as any other job. It is started automatically at system start up (by /etc/init, via /etc/rc) after all necessary system initialization has taken place.

It communicates with the Network code in the Kernel by opening a special device driver file (/dev/ncpkernel). With this mechanism, the NCP Daemon and the NCP Kernel exchange specially formatted commands as data passing thru the special file.

4.3 NCP Daemon Pseudo Special File

The Daemon and the Kernel communicate with requests couched in reads and writes to the special file /dev/ncpkernel.

4.3.1 Daemon Open

When an Open is done on /dev/ncpkernel, a procedure is called that marks the file open. Only one user is allowed to have this open at any one time. Initialization of the message system then takes place, and the IMP is made aware of the system's readiness to accept network traffic. A file descriptor is returned to the caller.

4.3.2 Daemon Read

When a Read is done with the previously opened file descriptor, a queue is checked for any messages. If one is found, it it transfered to the User space address specified, the queue count is decremented and the number of bytes read is returned. If no messages are available, the system puts the NCP Daemon to sleep until one arrives.

The contents of these messages are generated by implicit requests from the Imp, remote host, or User program.

There are three types of requests that the NCP Daemon will receive when it interrogates data returned from a read of the special device driver file /dev/ncpkernel:

OPEN Request

If the user specifies a standard open, the host number is copied in and the request is sent off to the NCP Daemon. The Daemon will perform the connection according to the type requested an will do a "release" on the file address when the connection is opened.

RCV Data

When data is received from the Network, that falls into the realm of the NCP Daemon, Host to Host Resets and the like; the following leader is attached, and it is sent to the NCP Daemon for decoding and response.

byte RCV opcode
byte IMP Type Field
byte Foreign Host
byte Link
byte IMP Sub-type

CLOSE Request

When the User desires that a connection be closed, the following request is sent to the NCP Daemon.

byte CLOSE opcode
byte Socket Index
byte File ID

4.3.3 Daemon Write

In response to the inputs received from reads on the special file, the NCP Daemon has eight requests that it may make of the Kernel. Combinations of these requests allow manipulation of the file system and the network to the extent that the status and control as well as the transient functions allluded to earlier are embodied in these responses.

Each response or Write done by the NCP Daemon includes an eight byte leader from which some or all of the information concerning the request and what is to follow is obtained. The leader includes a request type, a file or socket pointer, a host and link number, a status byte for indicating changes in connection status, and a byte size indicative of the units to send the network data.

Below is a description of each of these responses and what they mean to the Kernel.

Daemon Send

Daemon Send is essentially a request to the Kernel to send any bytes following the leader to the network. This means that the Daemon can carry on protocol conversations with foreign hosts.

Daemon Setup

This command asks the Kernel to procure a Socket from system resources, associate it with the file specified and initialize it with the host, link, status, byte size an allocation passed.

Daemon Modify

Modify is simply a means where by the Daemon may change the parameters of a socket previously allocated by a Setup command.

This is useful both in terms of the Initial Connection Protocol when a socket is going through the various stages of an Open and when the connection is being closed.

Daemon Ready

Ready is used by the Daemon to inform a user process that a response to his particular request has been determined and that he should examine the results.

Daemon Clean

Just as there is a request to setup a socket, there is a request to return a specific socket to the system pool for other use.

In response to either a user request or network request to close a connection, the Daemon will eventually issue a Clean indicating that as far as he is concerned the socket is of no further use. If the user is also done with the socket in question, it is released to the system, if not, the status is set to indicate that the Daemon considers the connection closed, and it is left to the user to indirectly decide when he is finished.

Daemon Reset

Throughout the course of network events, there are times when a foreign host wishs to reset all connections associated with himself. This usually happens when a host first comes "up" on the net, it's effect is to tell other hosts to start with a "clean slate" with reference to it's various network states. The Reset request is issued to the Kernel in response to one of these resets. The Kernel then proceeds to do Cleans on all sockets associated with the host. Essentially then the Reset is a form of multiple Clean on all sockets associated with a particular host.

4.3.4 Exemplary Daemon Communication

With many descriptions of this sort, it is difficult to see how individual pieces or in this case, individual requests make up the whole. With this in mind, what follows is a description of the Initial Connection Protocol on an instruction basis.

The notation is very free form with respect to the specific contents of the Network related data, but the instruction sequences and types are true to form.

In this little scenario, there are two nodes under consideration, the NCP Daemon denoted NCP and the section of the Kernel devoted to direct communication with the IMP, denoted IMP. When a line of the form "->imp:" is found, it means that something was received from the network for the local host. Each of these is usually followed by a line of the form "->ncp:". This is the instruction sent to the Daemon in response to the data received. Likewise when "ncp->:" is found, it means that the Daemon is sending a request to the Kernel.

No effort is made to familiarize the reader with the Initial Connection Protocol, for this a reference to NIC document 7101 is made. [Reference 4]

```
->ncp: open,---,---
ncp->: send, rts ----,---
->imp: str ----,----
->ncp: rcv,type,host,link,subtype,------
ncp->: setup,---,---,----
       send, hh alloc,----,----,---
->imp: data socket base
->ncp: rcv, socket base
ncp->: send, host,0, hh close,------
       modify, id,base,[clswait]
       send, host,0, rts,----,str,-----
imp->: cls ----, str ----, rts -----
->ncp: rcv, host,0,cls ----,str -----, rts ----
ncp->: clean, id, base
       setup, id,base+2,------
       setup, id,base+3,------
       ready, id
```

4.3.5 NCP Daemon Interface

The daemon issues writes to /dev/ncpkernel at various points in the process of opening a network file, specifying the opcode appropriate to the situation. The daemon issues reads to /dev/ncpkernel when it wishes to process information sent it by the kernel. Information which it reads often has an effect on the opening of a network file. The major routines involved in setting up and maintaining connections are as follows:

to_ncp, ncpread
: The first is used to queue up data for the daemon. The second is used to give it the data.

wf_ready
: Wakeup process (waiting in netopen) informing it as to the outcome of the open request.

wf_mod

Modify socket info. Specifically, set values of "data to go to ncpdaemon" and "socket is open" status bits, the byte size, host, link, and initial allocation values (wf_inc_alloc). Then place host, link, local skt, and foreign skt info in associated connection table entry (wf_update_skt).

wf_setup

Setup data structures to describe a connection. Get an inode to hold the socket and set status flags, byte size and allocation values as specified by daemon. Allocate a connection table entry and place socket information in it (wf_update_skt).

wf_clean

Clean up data structures associated with a connection when either daemon is done with socket or a host is reset (iclean). This will either cause the inode to be destroyed (daedes) or will cause the kernel to tell the daemon the socket is closed (daecls) so the daemon can later tell the kernel to destroy it when its ready to ask.

wf_send

Send ncp daemon data into the network (sndnetbytes).

wf_reset

Clean up all data structures containing information on the specified host. Clean up all sockets (inodes) for sockets that were associated with this host (iclean). The sockets are located by following chain from file table to the file block to the sockets themselves.

wf_frlse

Decrement the use count for a network file perhaps causing its release (net_frlse).

netstat

This returns status information about a connection to the caller.

Other Procedures

An incomplete list of other routines involved: wf_inc_alloc, wf_update_skt, sendnetbytes, daecls, daedes.

5 COMMUNICATION BETWEEN KERNEL AND DAEMON

5.1 Kernel to Daemon Communication

The daemon receives "open", "close", "rcv", "timo", and "reset" commands from the kernel. The daemon gets commands from the kernel by doing a read on /dev/ncpkernel. The kernel queues up the following commands for the daemon by calling "to_ncp".

opcode	command	comments
0	open	Passed only by "netopen", it builds the open parameter structure either by extracting it from user space, or building it itself, and queues it up for the daemon.
1	rcv	a) "hh" queues this command to pass data received for a socket flagged for use by by the daemon. The imp leader and the msg constitute the data. b) "ih" queues this command to pass the daemon an imp leader whose type field is other than "regular", "rfnm", "nop", "incomplete transmission", or "error in data". c) "siguser" queues this command if it realizes is signalling a condition (rfnm, error) for a skt flagged for use by daemon or if its signalling a rfnm for link 0 for which no user is waiting (e.g., user waiting for allocate so it can have one sent). The imp leader is passed. d) "hh1" queues this command to give the daemon the leaders and associated link 0 h/h protocol msg if 1) there's an illegal h/h opcode in the msg; 2) there were other than legal ALL commands to process (the legal ALL commands are nooped). e) "allocate" queues this command to give the daemon ALL commands prefaced by the imp leader for sockets flagged for its own use.
2	close	"daccls" queues this command in order to pass the daemon the close parameter structure when a connection has been closed.
3	timeo	A procedure invoked through the clock's callout table sets a flag and does a wakeup on the ncpdaemon. No message is placed in the NCPDaemon's queue since we dare not allocate memory during a clock interrupt.
4	reset	"netreset" queues this command when a process issues a netreset system call.

5.2 Daemon to Kernel Communication

"The Daemon can perform four classes of operations, corresponding to "open"s, "close"s, "read"s and "write"s on /dev/ncpkernel.

open

Causes kernel to be entered at "ncpopen". The network is initialized and "impopen" is called which forks the imp input process. Upon return, the smalldaemon then execs the largedaemon.

close

Causes kernel to be entered at "ncpclose". Causes cleanup of network.

read

Causes kernel to be entered at "ncpread". Causes next message in ncpdaemon's queue (which is filled by kernel via calls to "to_ncp") to be moved to the daemon's space to an area specified in the read call ("kr_buf").

write

Causes kernel to be entered at "ncpwrite". Kernel will fill its daemon request structure ("ncprs") from the daemon's space ("kw_buf") and invoke the appropriate processing routine which may copy additional data from the daemon.

The commands which the daemon can send the kernel are as follows:

opcode	command	daemon routine	kernel routine
0	send	sendpro,fi_ssn	wf_send
1	setup	kw_sumod	wf_setup
2	modify	kw_sumod	wf_mod
3	ready	kw_rdy	wf_ready
4	clean	kw_clean	wf_clean
5	reset	kw_reset	wf_reset
6	file rlse	kw_frlse	wf_frlse

send

send data to the network

setup

tells the kernel that the daemon has created a socket, and the kernel should do whatever it has to do to do the same.

mod

inform the kernel of a change of state in a socket, so the kernel can reflect the changes in its own data structures.

ready

inform the kernel of the outcome of a user-initiated open on a network file, so the user can be woken (informed), and appropriate action can be taken.

kreset

inform the kernel that it should internally reset a host (i.e., reinitialize data structures, etc.).

frlse

inform the kernel that the daemon is done with a particular network file, so the kernel should also do whatever it has to do to release its version of the file.

clean

inform the kernel that the daemon is done with a particular socket so that it can take whatever action it thinks is appropriate.

5.3 Where and Why the Daemon issues kernel commands:

send

1) in "send_pro" to send host/host protocol to the network.
2) in "fi_ssn" to send socket numbers (for server icps).

setup

1) in "lsint_q" when a listen or init operation occurs for a socket in the queued rfc state.
2) in "so_ulsn" when a listen or init operation occurs for a socket that doesn't yet exist.

mod

1) in "fi_sopn" when a user icp, server icp, data rcv, or data send socket is marked open. For user icp, the socket is marked "open and data goes to ncp", and allocation values are set. For server icp, the socket is marked "open and data goes to ncp". For data rcv, the socket is marked "open", and allocation values are set. For data send, the socket is marked "open".
2) in "fi_all" when an ALL command has been processed, and the allocation values for the affected socket are to modified.

ready

1) in "daemon_dwn", error "oc_eicp" is passed for all allocated files. This will wakeup any users waiting for opens to complete.
2) in "fi_sopn" when a file has completely been opened, errorless status is passed.

3) in "fi_sgone", error "oe_eicp" is passed when, due to a socket being closed, an open has failed.
4) in "kr_odrct", for bad parameter ("oe_badpar") and no resource ("oe_nrsrc") errors.
5) in "kr_oicp", a no resource error ("oe_nrsrc") is given if it couldn't get file or sockets.
6) in "kr_ouicp", for bad parameter ("oe_badpar") errors, and if a link # couldn't be assigned ("oe_nrsrc").
7) in "kr_osicp", for bad parameter ("oe_badpar") errors, and if a socket group couldn't be assigned ("oe_nrsrc").

reset

1) in "main" to reset all hosts when the net comes up.
2) in "daemon_dwn" to reset all hosts when the net goes down.
3) in "hdead" to reset a particular host when a host dead imp/host msg is received for that host.

frlse

1) in "daemon_dwn" to cause the kernel to release all of its files when the net goes down.
2) in "f_rlse", at the request of various routines:
a) "fi_sgone", if file closed due to error in a file open.
b) "kr_osicp" if we couldn't allocate a socket group.
c) "kr_ouicp" if we couldn't assign a new link #.

clean

1) in "cls_rfcw" when a cls is received for a skt in the rfc wait state.
2) in "cls_open" when a cls is received for an open socket.
3) in "cls_lsn" when a cls is received for a socket in the listen state.
4) in "clo_rfop" when a kernel close is received for a socket in either the rfc wait or open state.
5) in "so_ut3" when a kill is done on a socket in the rfc wait or open state, or when a timeout occurs for a socket in the rfc wait state.

6 SELECTED TOPICS

The intent here is to describe in some detail selected topics of interest. Following the code along with reading these sections is recommended. Some of the topics will be important only to system maintainers and modifiers. Others, such as opening a network file, will be helpful to persons writing network applications.

6.1 When the Network is Started

6.1.1 Overview

The network subsystem is initially started by executing the smalldaemon. This process opens the file /dev/ncpkernel. In the course of this open, while executing in kernel space, but off of smalldaemon's stack, a fork is done. Thie child of the fork then takes over responsibility for reading from the imp. He is in a "read-dosomething-readagain" loop until the network goes down. At that point he returns to the user level of smalldaemon and exits. The do-something port of the smalldaemon is the imp input process in impio.c. Depending on the information coming from the net he may 1) handle it himself, 2) put it in the largedaemon's queue and wakeup the largedaemon, or 3) put it in a queue of data for another reading process and wake up that process.

The largedaemon gets the file descriptor for /dev/ncpkernel because it was file descriptor zero when he was executed by smalldaemon. After initializing he goes into his main loop which consists of reading commands from /dev/ncpkernel and performing the functions those commands require. In addition he reacts to certain signals. In the course of performin his functions he may send commands back to the ncpkernel by doing writes on /dev/ncpkernel. Actually, he doesn't do a write in the normal way. He uses a routine to_ncp to place the information in a read queue (ncprq) for the ncp daemon.

6.1.2 Following the Code

Small daemon does open on /dev/ncpkernel. This goes to sys2.c|open and then calls sys2.c|open1. The first test that affects this call is for a character special file with major device indicating a network file. Files of the sort /dev/net/<host acronym> will have been set up with major device=255. However, /dev/ncpkernel has major device 15, so it will not satisfy this if statement. The next thing of interest is the call on fio.c|openi(). In this routine the fact that the mode is character special (IFCHR) causes a transfer of control to the open routine associated with a character special file with major device=15 (See conf/c.c). The rest of the open on /dev/ncpkernel will be handled by ncpio.c|ncpopen().

All that ncpopen does is check to see if he's been called twice, arrange for some buffers, and call the imp open routine ncpk/drivers/impio.c|impopen().

The impopen routine immediately forks off a copy of itself. (Keep in mind that the copied process is of both the kernel and user memory portions running on the same stack.) The parent process immediately returns, (eventually) to the smalldaemon just after the open of /dev/ncpkernel, carrying with it the return value equal to the file descriptor for this file. If it were to have returned any kind of error indication it would have to have been detected somewhere along the way prior to the call on impopen.

Now, the child process spawned by the call on newproc (in impio.c|impopen) releases the file descriptors of its parent's files, sets up a "once-only" flag for initialization, and goes into its main loop.

Figure 2

Synopsis of Impopen

```
impopen {
     if newproc {
          Release parent's files;
          needinit <- 1
          Forever {
               Main "for loop";
          }
          imp_dwn();
          exit;
     }
     else exit;
}
```

Note: newproc causes a fork. The child process functions are specified in the "if" portion and the parent process functions are specified in the "else" portion.

Figure 2 describes the fork and Figure 3 describes the main "for-loop" of impopen. On the first pass through he does the initialization and starts a read of 8 bytes from the imp. (The 8 bytes corresponds to the size of a imp leader.) The routine impread is in one of: ill.c, implla.c, or acc.c depending on the interface used. It sets all the appropriate bits in the interface, causing the read to start, and then exits. When control returns to impopen he goes to sleep. He will be waked up at this point whenever the imp interupt routine determines that the read has completed.

When impopen has something from the interface he hands it to impio.c|imp_input(). This procedure will do a number of different things depending on its sate and on what came in, but in all cases it will result in starting another read on the imp and returning. When control gets back to impopen he is in the main "for" loop and will again sleep until the next read of the imp completes. All of the input of information from the net comes through this reading loop.

6.1.3 Imp Input Errors

If an imp input error occurs, he calls imp_reset, which toggles the host master ready bits back and forth once on both input and output sides. Then he increments the "needs initialization" flag. Immediately, next time through the "main for loop", imp_init gets called. Imp_init also calls imp_reset, so this is done twice. Imp_init then enables the interface and returns to impopen in the initialization loop. At that point it starts all over again. For the Illinois interface these routines are in ill.c. Figure 4 represents the routines involved in initializing and resetting the IMP.

Figure 3

Main "for loop" of Impopen

```
Forever {
     Block imp interrupts;
     if need init {
          needinit <- 0;
          call imp_init; /*does not return
                    until imp is
                    ready*/
          For 1 to 3 {
               Send h-h NOP to imp;
               Wait til completes;
          }
          Put a permanent receive command
          in the imp receive buffer;
          Console message: "IMP: Init";
          Start read of 8 bytes from IMP
          into input buffer;
     }/*end of initialization*/
     Sleep until something has arrived from IMP;
     /*will be waked by routine imp_odone*/
     If ncp went down {
          exit from the forever loop;
     }
     else {
          if imp reading error{
               Console message: "IMP: input error";
               Call imp_reset();
               needinit <- needinit+1;
          }/*end of error loop,
          break to forever loop*/
          else { /*no imp reading error*/
               Call imp_input to handle the
               received information;
          }
}/*The loop is left only in case
   of ncp closedown or error.*/
```

Note that in the case of errors like this, the information read from the net never gets handled

Figure 4

IMP Procudures in ill.c

```
imp_init(){
     set the reset bit in interface;
     clear extended memory bits;
     HMR_interval <- 60;
     set_HMR;     /*starts continuous strobing
                  of ready line. Causes a periodic
                  timeout. Stops if somebody has
                  set HMR_interval to 0 then
                  wakes up at that point. */
     while imp master ready not set {
          sleep on lbolt, check every 4 secs;
     }
     /*now the imp is ready*/
     enable imp interupts;
}


set_HMR(){
     if HMR_interval non-zero {
          set output host-master-ready;
          /*the following causes this routine
          to be run every HMR_interval secs.
          If anybody clears HMR_interval and
          goes to sleep on it, they will be
          waked up at the next interval of
          HMR_interval seconds.            */
          timeout(&set_HMR,0,HMR_interval;
     }
     else if HMR_interval is zero {
          wakeup whoever is sleeping
          on HMR_interval; /*(imp-reset will be)*/
     }
}


imp_reset(){
     console message: "IMP:reset";
     set priority;
     HMR_interval <- 0;
     sleep (until HMR_interval timer goes off);
     set reset bit in interface;
}
```

by the imp_input process, and no indication of the error is sent either.

6.2 Initial Host Resetting Algorithm

After smalldaemon starts largedaemon1 (1main.c), kw_reset(0) is called. The comment is that it will reset all of the hosts in the kernel. Kw_reset takes the argument (in this case 0) to be the host number. He packages it with the reset op_code and writes it to the kernel (with kw_write at line 304 in ncpd/kwrite.c). The result returned will be a 0 if everything was ok, a 1 otherwise. However, 1main.c does not check this returned value. The presumption is that if anything went wrong it would be already handled before the return.

Since the op_code is a reset, in the kernel it will be handled in routine wf_reset (line 644 of ncpio.c). It got there via ncpotable in ncpwrite (line 393 of ncpio.c). Given a host argument of zero, wf_reset does an iclean on the inodes for each file in use by the ncp. Iclean destroys the inode. If some user still thinks the inode is useable he is waked up and told it is no longer the case. Iclean always returns 0, but may also have waked a user prog. Wf_reset also returns 0.

The effect of the call on kw_reset (at line 117 of ncpd/1main.c) is thus only to set the tables of network files in the kernel. It does not directly cause any rests to be sent to the network. It does directly cause all the user programs with open network files to be notified that the connection has been closed.

The "reset all" (rst_all, line 296 of ncpd/send pro.c) is triggered by the daemon receiving an imp-to-host message that says the net is resetting. This happens as one of the first inputs from the IMP after it is activated. It also happens if the IMP goes down and come back up.

Rst_all begins a cycle that would, if uninterferred with, sequentially send resets to every other network host. The purpose is as much to let them know this site is up as it is to find which other sites are up. For the moment just presume there is a "list" of hosts. Rst_all will cause each of them in turn to be sent a "reset" command. When one of these is either acknowledged or timed out, rst_all sends the next one. While all this is going on it is still possible to open a connection to a site that has not been sent a reset. When that site comes up in the initialization cycle it will not be sent another reset.

The initial call on rst_all comes from procedure ir_reset (line 526 of ir_proc.c) in response to a reset message from the imp. This message was passed from the kernel to the daemon via to_ncp and decoded in the module kr_dcode.c. On the initial entry a bit map of known hosts is initialized. Then for each host on that list, the routine chk_host is called. If the host has not been in use, a reset is sent to that host and rst_all is exited. If the host has already been in use, rst_all keeps moving down its list either until finished or until it actually ends up sending a reset. The number of the host being "initially reset" is kept in a global "init_host".

The reset commnad sent to another host will ultimately be responded to by either a "rfnm" (request for next message) or an incomplete transmission. When either of these is received back from the imp, a check is made to see if it applies to the host being initialized. If so, another call is made on rst_all to trigger initialization of the next one on the list.

The set of hosts to be initalized in this way has historically been arrived at in several ways. At the very first all hosts were reset before anything else took place. That was extremely time consuming, so the scheme outlined above was developed. It operated on all possible host numbers from 1 to 256. Later a global value rst_max was used to limit the number of these initializing rests. More recently, a system list of "known hosts", kept in /usr/net/hnames, was used for this purpose. The code as it now stands has both methods, depending on a conditional compile based on whether RSTKNOWN is

defined or not. Most sites will probably want to use the "only known hosts" scheme, particularly when there aren't very many of them compared to the possible 256. Also, when the new IMP leader format is used and there are many more than 256 hosts possible, the scheme of initializing only a subset of the possible hosts will be preferred.

6.3 Opening a Network File

6.3.1 Declarations

The user program which is going to perform an open on a network file should do the following include to get the required struct and type definitions:

```
#include        net_netopen.h
```

This File contains a structure declaration of the following form:

```
struct openparams{
        char  o_op;
        char  o_type;
        int   o_id;
        int   o_lskt;
        int   o_fskt[2];
        char  o_frnhost;
        char  o_bsize;
        int   o_nomall;
        int   o_timeo;
        int   o_relid;
        }
```

In order to use this, the procedure which is going to do the open of a network file should have an instance of this struct declared to reserve space and have a name for referencing it; for example:

```
struct   openparams   alpha;
```

The name alpha is thus given to an area of storage large enough to contain the structure, and fields within this structure will be referenced by the names alpha.o_op, alpha.o_type, etc. Now let's look at these elements in turn and give a little more information about them than the comments field in net_netopen.h offers to the novice.

6.3.2 Elements of Openparams

char o_op This field is never used by the user. However the space in needed because it will be filled in as this set of parameters is handed around inside the network system. It will be filled in with a kernel to daemon op_code which is of little or no concern to the user process. If it makes you feel better, do alpha.o_op=o;.

char o_type

This is a type field which the user program will need to fill in. Each bit that is currently used by the system has a define associated with it in net_netopen.h. More detail on the types later.

int o_id This field is not filled in by the user procedure, but the space for it must be there because it is used by the system internals. It will be used to communicate the identity of the file being dealt with between the NCP kernel and the NCP daemon.

int o_lskt The local socket number. This can be either an absolute socket number or an offset from the base of a set of sockets assigned to your file by virtue of a previous open. (See comments associated with o_relid.) Typically the user process knows what local socket he wishes to use based on certain conventions or assignment of sockets. At any given site there will be certain sockets set aside for specific functions and most of these will correspond network wide for the corresponding functions at all network sites. Other conventions could be established by network fiat. Notice that Unix is only using 16 bits to specify the local socket although the corresponding field in Host/Host protocol is 32 bits. Network Unix implementors simply decided that larger socket numbers would not be needed.

int o_fskt[2]
or
long o_fskt Thirty-two bits are needed to specify the foreign socket. Usually the low-order word is all that is needed. This can also be either relative or absolute. On a listen this field has no meaning. On an init it must be given. If the system you are using has o_fskt declared as a long, you needn't worry about the index. If o_fskt is 0, the default is to connect to socket 1.

char o_frnhost This is the number of the host to which you are connecting. If you know the host number you can provide it, otherwise set it to zero. If o_frnhost is zero, the name string which is the first argument to open (see below) will be used to look up an appropriate number. If o_frnhost is not zero it will be used regardless of the first argument to open. If the connection is a general listen this field should be left zero and the first argument to open should be /dev/net/anyhost.

char o_bsize The byte size of the connection. In Unix this must be a multiple of eight bytes, e.g., 8, 16, 24, 32, ... BUT, until further changes are made in the network Unix system it should be 8. For the moment, all final connections resulting from an ICP will be 8-byte connections and the value returned from a read or a network file is the number of 8-bit bytes accepted.

int o_nomall This is the nominal allocation for a normal network connection. It indicates how much data the local host should be willing to accept. Later on, when pipeline protocol is accepted, this field will have a double meaning. See the discussion of new data declarations for

openparams in the pipeline program specification.

int o_timeo — The number of seconds to wait before timing out on a connection attempt. A value of 90 is often used. A value of zero means you are willing to wait indefinitely.

[int o_pplcnt] — [This will be added to the openparams when the pipeline protocol is implemented. Then it will indicate the pipe count. In the meantime, do not use it!]

int o_relid — Is needed if the o_relative type bit is set. It is to be set to the file id (returned from some previous open) of a netfile. Whenever a network file is opened, a set of eight sockets (4 read and 4 write) is reserved for your use. Type relative means that the number specified in o_lskt is an offset (between 0 and 7) from the first socket you were assigned when doing the original open. Many network protocols use relative sockets when more than a duplex pair of connections is needed to perform their assigned tasks. In FTP for example, control messages pass back and forth over the initial socket pair, but an additional (relative) socket is used to open the data connection.

6.3.3 The Bits in o_type

The type of connection you need determines how you will set these bits. Unfortunately, some understanding of the host-to-host protocol is required in order to decide what to select. Unless you are completely familiar with that protocol you'll want to refer to it to fully appreciate your options. The definition of the flags is by-in-large in the order you'll want to make these decisions.

o_direct — You set this bit only if you want a direct connect. A connection can be initiated as direct or ICP (initial connection protocol). In the case of direct connect, the socket speciied is used for hte life of the connection. In the case of ICP, the specified socket is merely used as a short-lived contact during which the two hosts exchange socket numbers to be used for the send and receive sides of a more long-lasting connection. The initial "contact" connection is closed as soon as the required information is exchanged in order that the socket can be available for other contacts. In Unix, if you specify ICP (by failing to set the o_direct bit in o_type) the open will return only when the "target" connection is open.

o_server — This bit is only used if o_direct is not set, that is, if you are doing an ICP connection. You set it if you want to listen for an ICP connection from another site. Leave it unset if you are initiating an ICP connection.

o_init — This bit is only used if o_direct is set. If you set it, you are requesting to initiate a connection. This implies that a host-host RFC (Request for Connection) is going to be sent. If this bit is not set, you will be doing a listen, i.e., preparing the system to accept a connection from

another host.

o_specific If this bit is set you are requesting a connection only with a specific host (according to o_frnhost or the first arg of the open). The bit is only meaningful if o_init is not set (i.e., you are listening). Listening for a connection from a non-spcific host makes sense; initiating a connection to a non-specified host does not make sense.

o_duplex If this bit is set you are requesting two associated connections, one for reading from and one for writing to a remote host. If the bit is not set you are requesting a simplex connection either to read from or to write to a remote host. Which one you want to do is determined from the gender [evenness or oddness] of what you specified as o_lskt. Even sockets are for reads; odd sockets are for writes. If you specify duplex, be sure you set o_lskt and o_frnskt to even values. The connections will be set up from o_lskt to o_frnskt+1 and from o_frnskt to o_lskt+1.

o_relative This bit is to be set if the number for o_lskt is less than 7 and is intended to indicate one of the sockets in the group of eight assigned to an already existing file id which was established by a previous open. The association is made based on the parameter o_relid.

[o_ppln]
[o_norml] [These two will be used when the system is expanded to use pipeline connections. If o_init is set, one or the other but not both of these should be set. If o_init is not set (on listens) having either or both of these set indicates willingness to accept pipeline or normal or both kinds of connections. Regardless of the state of o_init, having neither of these bits set is equivalent to setting just o_norml.]

6.3.4 Examples of Setting o_type

Usually the entire type field will be zero when you declare it, but if there is any question, set it to zero yourself. Then,

alpha.o_type = o_direct | o_init | o_specific

indicates you want to *initiate* a *direct* connection to a *specific* host. For this type of connection you'll have to all provide values for alpha.o_lskt, alpha.o_frnskt, alpha.o_bsize, alpha.o_nomall and alpha.o_timeo. The other flag bits in o_type will remain their default values.

alpha.o_type = o_specific | o_server

indicates you want to listen (sever listens) for connections from a specific host and when contacted, perform the ICP ritual.

6.3.5 The Open Call Itself

After you have made all the above preparations, (declaring the struct, reserving the space, naming the reserved space, setting all the appropriate flag bits in o_type, and setting all the other required elements of the openparams struct, you can then do the open call:

```
fid = open (<hostname>, &alpha);
```

The specified host name and the openparams at address alpha will be used to open your connection. The <hostname> must be a defined file on your system, usually "/dev/net/<host acronym>". Examples are /dev/net/ill-nts, /dev/net/usc-isic, and so forth. On general listens the host names specified must be /dev/net/anyhost.

The address &alpha must be the start of an openparams struct, which you have just so carefully filled in with appropriate information. This sytem call will not return until either the conection has opened or the system has discovered that the connection cannot be opened. If the returned fid is less than zero the connection failed for some reason. The Unix routine perror will tell you why.

6.4 The Socket Machine

The "Socket Machine" is implemented as a collection of procedures in the NCP daemon. The machine is operated in the C language by procedure invocations of the form:

```
(*skt_oper[command][old state])(socket pointer);
```

The array skt_oper contains names of procedures which will perform the required action and set the new state. Each of these procedures requires a single argument which is a pointer to the socket on which the operation is to be performed. The state transition array (for the reader also looking at the source code) is declared on the last page of file ncpd/skt_oper.c. This array is used for requests on sockets that already exist. For requests on sockets that do not already exist, the array so_unm is used in the same way. It is declared on the last page of ncpd/so_unm.c. Table 2 which follows presents an overview of the state transitions and actions performed. It is recommended that the reader follow the operation of this machine through the source code, using this table merely as a reference.

The socket machine is driven in two ways, but always in response to some input delivered to the NCP daemon from the NCP kernel. The first way is to act on a socket request. Space for one socket struct (skt_req) is declared for this purpose. Routines which use the socket machine in this way first fill in the appropriate fields of skt_req and then call upon the procedure get_skt. This procedure attempts to fulfill the request by finding a socket struct whose local socket field matches that of the request. If successful it calls the appropriate procedure through skt_oper. If not successful it calls one of the procedures through so_unm.

The second way the socket machine is driven is by specifying the command state directly rather than filling the struct skt_req. As an example of this, in ncpd/kr_dcode.c the following line of code is found:

```
(*skt_oper[si_close][s_p->s_state])(s_p);
```

The socket pointer s_p points to a socket struct which has the correct current state. The string si_close is defined to be the value of a close command. This code would be executed when the command sent

Table 2

The Socket Machine State Transition Matrix								
rrent	Inputs which will change state/action; (new state)							
te (No.)	Kill	Timeout	host dead	s/sn/glsn	init	rfc	cls	CLOSE
l (1)				create socket; KSETUP; (3)	create socket; KSETUP; (9)	queue request; (2)		
eign RFC ued (2)		send cls; (7)	Free socket; KRESET; (1)	attach socket to file; KSETUP; send rfc; file level open; (8)	attach socket to file; KSETUP; send rfc; file level open; (8)			
en (3)	KCLEAN; (5)	KCLEAN; (5)	KCLEAN; KRESET; (5)			file level open; (8)		KCLEAN; free socket; (1)
)SE to wait (4)								send cls; free socket; (1)
)SE wait (5)								free socket; (1)
ind CLOSE t (6)		(5)	KRESET; (5)				(5)	(7)
wait (7)		free socket; (1)	free socket; KRESET; (1)					
n (8)	send cls; KCLEAN; (6)		KCLEAN; KRESET; (5)				KCLEAN; (5)	send cls; KCLEAN; (7)
C Wait (9)	send cls; KCLEAN; (6)	send cls; KCLEAN; (6)	KCLEAN; KRESET; (5)			file level open; (8)	send cls; KCLEAN; (5)	send cls; KCLEAN; (7)

up from the kernel indicated that a close had been done on the file associated with this socket.

In the state transition table, an action specified in capital letters indicates a request which will be sent directly to the kernel from the daemon in order to keep them synchronized with respect to sockets and files. The procedures are all in source file ncpd/kwrite.c. The ones which correspond to the table are kw_sumod (KSETUP), kw_clean (KCLEAN), and kw_reset (KRESET). In addition, some of the other actions specified will also result in a command being sent to the kernel. For example, the file level open will start a series of events with the "File Machine" and when that series is completed the socket will be in the open state.

Also in the table, "cls" means a host-to-host protocol cls command, while "close" means a local close of the file.

6.5 The File Machine

The "File Machine" can be thought of as a finite state machine similar to the socket machine. However, the implementation is not clearly delineated by driving the routines through a two-dimensional array of procedures. For the file machine, state information is kept in the element f_state of a file structure. In general this state information is used to control switch statements in the code which in turn will perform the appropriate actions and set the next state. Most of the procedures dealing with the file machine are in ncpd/files.c, although some references to the state of a file can be found in ncpd/krdcode.c.

The file machine state transitions are summarized in Table 3. As in the previous table, capital letters generally indicate a procedure that will send a command to the kernel. Because of the dispersed nature of the file machine it is somewhat more difficult to follow in the code, but this is recommended as the only way to get the complete story. The earlier section on opening a network file will get the reader started. The four events which cause transitions from the "null" state are detected through a series of tests starting in ncpd/krdcode.c/kr_open. Remember also that the daemon may send commands to the kernel as side-effects of other actions.

Table 3

The File Machine State Transitions			
Current State (No.)	State Changing Event	Response to the Event	Next State
Null (1)	UICP Open	get UICP socket (KSETUP); send rfc [RTS] to foreign server (KSEND)	(2)
	SICP Open	get SICP socket (KSETUP); listen	(4)
	Non-ICP Open (listen)	obtain a socket; listen; KSETUP	(7)
	Non-ICP (init)	obtain a socket; init connection (KSETUP; KSEND)	(7)
UICP OPEN WAIT (2)	foregin rfc received	open (KMOD); send allocate for socket (KSEND)	(3)
	free ICP socket	detach ICP socket; (KFRLSE); release file (KFRLSE); open err (KRDY)	(1)
UICP Socket Number Wait (3)	received socket number from server	init duplex connection to s, s+1 (KSETUP; KSEND); init freeing of UICP socket	(7)
	free ICP socket	detach ICP socket; release file (KFRLSE); open err (KRDY)	(1)
SICP OPEN Wait (4)	foreign rfc received	send matching rfc [STR] (KSEND); open (KMOD)	(5)
	free ICP socket	detach ICP socket; release file (KFRLSE); open err (KRDY)	(1)
SICP Allocate Wait (5)	allocate received	send socket number (KSEND)	(6)
	free ICP socket	detach ICP socket; release file (KFRLSE); open err (KRDY)	(1)
SICP RFNM Wait (6)	RFNM received	init duplex connection to u+2, u+3 (KSETUP; KSEND); init freeing of ICP socket	(7)
	free ICP socket	detach ICP socket; release file (KFRLSE); open err (KRDY)	(1)
DATA Open Wait (7)	partial open	open (KMOD); if rcv socket send ALL (KSEND)	(7)
	free data socket for simplex file	release file (KFRLSE); open err (KRDY)	(1)
	Full open	open (KMOD); if rcv socket send ALL; KRDY (null)	(9)
	free data socket for duplex file	detach socket from file; init freeing of other socket	(8)
Gone Wait (8)	free data or ICP socket from file having no other sockets attached	release file (KFRLSE)	(1)
	free data or ICP socket from file having other sockets attached		(8)
Open (9)	free data socket for duplex file	detach socket from file; init freeing of other socket	(8)
	free data socket for simplex file	release file (KFRLSE)	(1)

List of References

1. Unix Time-Sharing System
   Ken Thompson and Dennis Ritchie
   Communications of the ACM
   July 1974, Vol 17, Number 7

2. Specifications for the Interconnection
   of a
   Host to an IMP
   Report no. 1822 Bolt Beranek and Newman Inc.
   Chapter 3, System Operation

3. Host/Host Protocol for the ARPA Network
   Alex Mckenzie, BBN
   NIC Document 8246

4. Official Initial Connection Protocol
   Document #2
   J. Postel, UCLA-NMC
   NIC Document 7101

5. Ants Mark I User's Guide
   Karl Kelley
   Center for Advanced Computation 2/1/74

6. Ants Mark Two System
   Karl Kelley
   Center for Advanced Computation 1/10/74

BOUND BY
HM

UNIVERSITY OF ILLINOIS-URBANA
510.84IL63C C001
CAC DOCUMENT$URBANA
243-246 1978

3 0112 007264077