The Berkeley Network − A Retrospective

Eric Schmidt

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

The Berkeley Network connects a number of UNIX® machines on the Berkeley campus. It provides facilities for file transfer, sending and reading mail, and remote printing. Operating in a batch mode, network requests are transferred one by one through an inter-connected network until they reach their final destination.

This document describes the history and goals of this project, the design decisions faced, discusses issues in portable software development in networks, and discusses the future of this project.

Introduction

This document is intended for readers with an interest in networking who are familiar with two other documents about the network, ‘‘An Introduction to the Berkeley Network’’, and the ‘‘Network System Manual’’, by this author. It is not necessary to read this document to set up and maintain the network, although systems persons will benefit if they are familiar with the concepts presented here.

The sections are presented as follows:

The most important section is the last, which details a set of principles the author has learned during this project.

Principals

This project was a collaboration of many individuals. Dr. Robert S. Fabry participated in the initial design and has exerted the strongest influence on polishing the final product. Bill Joy and Ed Gould provided valuable technical expertise every step of the way, primarily about developing systems programs. The support staff of the EECS Department and the Computer Center (Bob Kridle, Jeff Schreibman, Vance Vaughan, Robyn Allsman, and Ricki Blau) were involved in setting up and administering this multi-domain project. The lowest-level concepts the author used came from experience at XEROX PARC, primarily from discussions with David Boggs, one of the ETHERNET† designers.

History

The network project can be divided into two distinct phases. The first, from January 1978 to May 1978 (4½ months) involved designing and implementing the network facilities now available. The second, from October 1978 to March 1979 (5 months), saw the addition of many more machines to the network with emphasis on portability, security, and minor design changes. The network has been in almost continuous service to users since May 1, 1978.

First Phase

An initial design was worked out with Dr. Fabry in January 1978. A suitable connection was made between the Computer Center ‘‘A’’ and ‘‘Q’’ machines (then called ‘‘D’’).

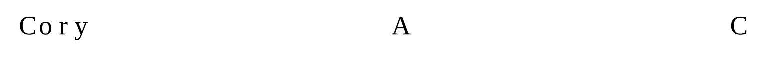
Development proceeded on two fronts − a set of daemons were written to transfer files, using UNIX pipes for debugging. The lowest-level protocols were designed and implemented of the terminal-type connection between A and Q. These were debugged using simple programs to send and receive packets, and a pair of programs to transfer a file from one to the other. This was the first experience with a distributed software development − the Q machine is a DEC PDP-11/34 with no lineprinter, a non-standard tape drive, and terminal access only by telephone, so most development had to proceed on the A machine.

During this phase the goals of the project increased in scope. The implementor and only user had to use the network to transfer network source and worked out simple ways to automate this (the ‘‘.netrc’’ file is an example).

When it appeared usable by more than the implementor, the connection was changed to be between A and the Computer Center ‘‘C’’ machine:

Up until now, the network had required an account on both machines. It became clear this was a handicap since the A machine had too many accounts and the password file was immense. Certain ‘‘free’’ commands were allowed, without an account.

The Cory machine was soon added to the network:



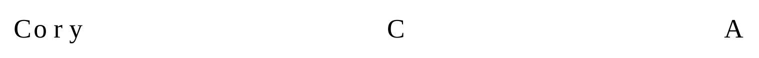
† ‘‘Ethernet: Distributed Packet Switching for Local Computer Networks’’, by Robert Metcalfe and David Boggs, CACM, July 1976.

This produced major changes in the design − initially we had assumed network users would have accounts on all machines. This was unreasonably strict and a solution (kludge) was worked out where a request was examined and forwarded through the queue(s) on the intermediate machine. The network became table-driven to accomplish the routing, and distributed software development became more difficult because of the increased number of machines. The implementor quickly discovered only one solution: Always have a designated source machine for all changes. To this day, software changes are made only on this ‘‘source’’ machine, the others are guaranteed to have copies. This makes remote updating (copying new versions around the network) possible.

Documentation was written and sent off to about fifteen faculty, staff, and graduate student users. A few bugs were fixed and the system frozen from the end of May to October. End of Phase 1.

Second Phase

While the implementor was away at a summer job, the connection between the A and C machines was switched to reduce the loading on the A machine:

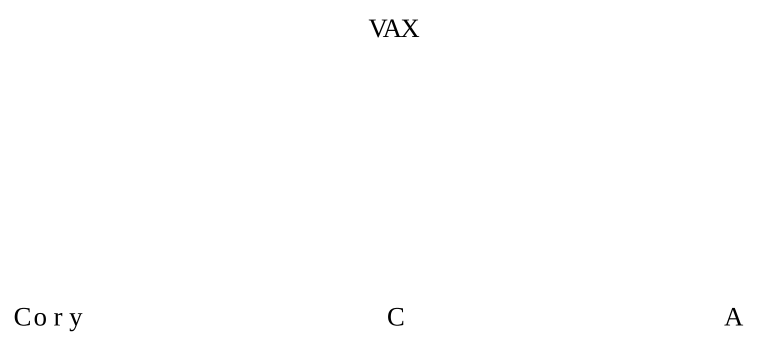


Unknown to the implementor, the network source was modified in divergent and incompatible ways, and the commands were moved to a different place. These changes invalidated certain assumptions about full pathnames and some commands such as inter-machine mail stopped working.

The Computer Center had also made absolutely incompatible changes to some system calls. This began a path of software divergence that became very painful and is still not completely solved.

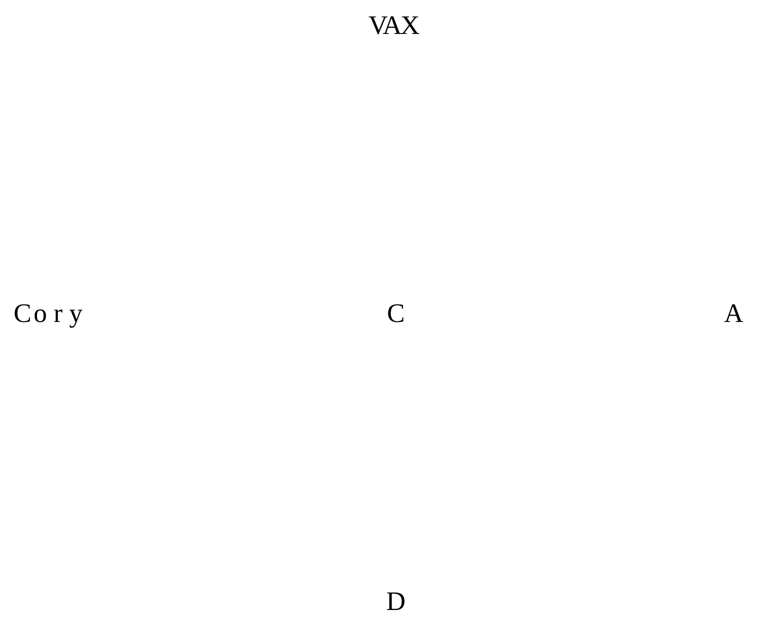
Fortunately, the Computer Center placed Robyn Allsman in charge of maintaining the network on their machines − to lighten that routine part of the load from the implementor.

The Computer Center acquired a ‘‘D’’ machine, and the EECS Division a DEC VAX 11/780, running an experimental Version 7 UNIX system. The implementor decided to use the (at this point) unused VAX to to do software development and incorporate the Version 7 changes into the network code. By this time, the protocols were stable which made it possible to run a Version 7 network on the VAX connected to the old Version 6 code on the existing network, to facilitate debugging. Because of improved terminal availability and better machine response, many new ways were used to debug the network − using pipes, using TTY lines wired together on the VAX and over the usual machine-to-machine link. A file was added (‘‘initfile’’) which allowed quick reconfiguration of the daemons when system parameters were changed. A temporary connection between the VAX and C machines was arranged.

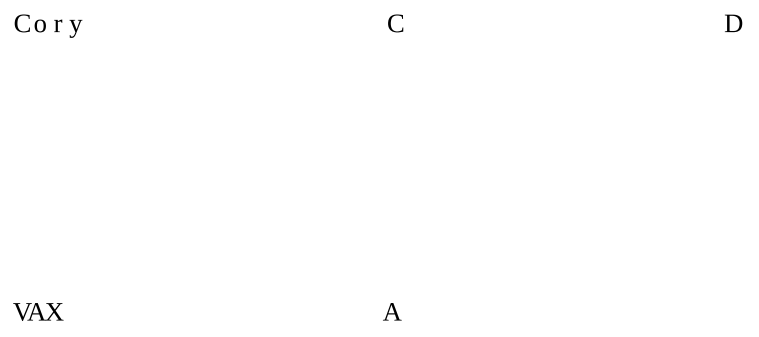


The network code had to be able to run on three different types of machines − the VAX running UNIX Version 7, the Cory machine running Version 6, and the anomalous Computer Center machines. There was no conversion package available at the time,† and the old network code had not used any system header files, so after a great deal of experimentation, conditional compilation was used as much as possible and a procedural interface was used to elide system differences.

The new UNIX command *make* (I) was used with a ''makefile'' to organize this regime. The old network code was used to bootstrap the D machine onto a network running the new network code exclusively.



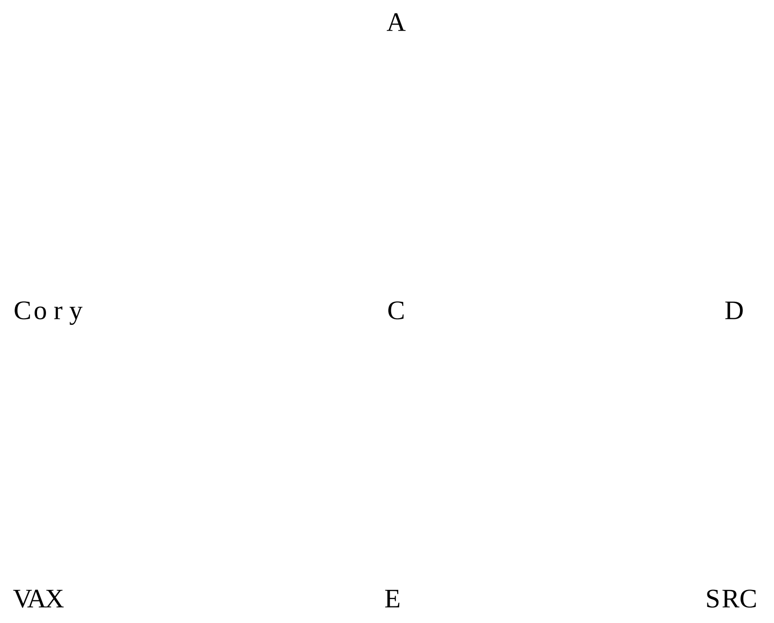
Shortly thereafter, over the Christmas break, the VAX and Cory connections went down for security reasons (discussed in the ''Security'' section). After seven weeks, they reentered the network in a new configuration.



Shortly after that, they were down again because of a lightning strike for another week but have been operational since then. During the last time period the network was made less Berkeley-specific and a copy

† The ''retrofit'' library, by Bill Joy, is now available.

was run on the Rand Corporation UNIX machines. Documentation was rewritten and prepared for release. The network queues were converted to send shortest-job first. Extensive monitoring of system load, network performance, and network use was added. The format of the *netq* command was changed to summarize more information on output. The E machine, and then later the Survey Research Center (SRC) machine, were added.



Tuning was still important − serious overloading problems caused by sluggish response stopped the network between Cory and C for a week. Network parameters were tuned to help solve this problem. The complexity of software development and maintenance became too great for unstructured changes. Versions on all the machines except the VAX were frozen for a month at a time. The protocols were almost immutable. People were delegated responsibility to observe and straighten out, if necessary, problems with the net queues.

As this is written, the software is stable and the user-documentation is finished and being sold, and there is hope of adding more UNIX machines to the network.

Overall System Description

The Berkeley network operates in a batch/ request mode, and is similar in concept to a line printer queue. ‘‘Requests’’ are queued up at the source, where they are sent in shortest job first order through an interconnected network of machines to their destination. At each intermediate node, they are queued as if they were originated locally.

The network consists of a set of user-executed commands, a queue of requests to be sent, and a continuously-running program called a *daemon* which transmits requests in the queue and listens for any request being sent to it. There are many network commands − one to send mail, one to read mail, one to copy files, etc. They all use the *net* command to access the network. The *net* command takes a command, assorted parameters, with any input data, and puts a request in the queue. These requests are composed of a header, the command to be executed, and any data for input to the command. The header contains network information such as the destination machine, login name, and password. This request is stored in the queue as a normal ASCII file, owned by the invoker. This way UNIX commands can be used to examine the file.

The daemon examines the queue to see if there is anything to send. If so, it begins sending to a remote daemon, using a protocol to establish this dialog, involving retransmissions and timeouts. The remote daemon accepts the requests, parses the header information, and takes any data for the command and puts it in a file. The main loop of the daemon then returns to a waiting state.

The command execution is done by ‘forking’ a series of processes. One of these is the user’s login shell, which is given the command to execute. Another is a process which waits for the command to be executed, then examines the output of the command. The output is typically sent back to the user, via a *net* command, executed by the daemon.

In the reverse transmission, the command is called ‘‘mwrite’’, and it is routed and handled exactly as in the forward mode, except no password is required. The ‘‘mwrite’’ command is executed on the original machine with input data which is a copy of the output of the remote command. The user is either ‘‘written’’ or ‘‘mailed’’ to, depending on certain options. If ‘‘written’’, the user’s screen is filled with the output† of the command executed remotely, just as if he had executed it locally. Otherwise, it is in his mailbox, as mail from the remote account he used. The user’s terminal must be write-able (see the *mesg* (I) option), the originating user must still be logged in, and he must not have logged off and on again.

The output from the command is preceded by a line of information listing the command, the time it was sent, and the time elapsed.

Our design then tries to simulate local UNIX commands as much as possible. With defaults set correctly† the user in principle need only precede the command he is executing with the command *net.*

Copying Remote Files

The most frequent use of the network is file-transfer using the *netcp* command. The *netcp* command is based on the *cp* (I) command. Its two arguments are a source and destination file, optionally with remote machine names prepended:

netcp *from−file to−file*

where the names are local or remote. Since

```
Cory:/usr/pascal/sh
```

is a file on the Cory machine,

```
% netcp  junk  Cory:/usr/pascal/sh
```

will transfer the file ‘‘junk’’ to the named file on the Cory machine‡.

The way the transfer is accomplished depends on the type of file copy:

1. Copy local file to remote file −
   On the remote machine a *cat* (I) command is executed on the remote file with the local file as standard input.
2. Copy remote file to local file −
   A *cat* command is executed on the remote machine from the remote file to standard output. This standard output is sent back to the local machine into a *response file,* instead of being written or mailed to the user.
3. Copy remote file to another remote file −
   If both are on the same machine, a *cp* command is sent. Otherwise, a *netcp* command is sent to the remote machine where the *from−file* exists, to copy that file to the *to−file*’s machine.

This last case is experimental. Unfortunately, the system is structured only to carry one login name and password to a remote machine. Since the last option involves three machines, the second remote machine is handled imperfectly at best.

† Standard output and standard error files.

† With a ‘‘.netrc’’ file, see below.

‡ For more examples, see the ‘‘An Introduction to the Berkeley Network’’ document.

Sending Mail

The *mail* (I) command on the network machines has been modified to examine the names of the recipients of a particular message. If their names have a remote prefix, *mail* executes an internal command ‘‘sendmail’’, which in turn executes a *net* command. This net command sends a mail command to the remote machine, with the message as input. Since the recipient would like to know which machine the message came from, a simple program ‘‘mmail’’ is executed to insert a pseudo-header indicating the real source of the mail. The net command logs in as user ‘‘network’’, so remote mail does not require an account on the destination machine. This facility has proven invaluable.

Reading Mail

The *netmail* command uses the *net* command to send a command to read mail for a specified user on a remote machine. Since the existing mail programs on different machines vary in their options, it was decided the only thing that would work on both UNIX Version 6 and 7 systems was to copy the mail from the remote to the local machine. If the user subsequently logs in on the remote machine, his mail will be there, as if it were unread. An internal program ‘‘prmail’’ is used to copy the user’s mail back to the local machine. It knows the location of the mailboxes and the user’s name.

The mail programs at Berkeley are being modified to search a database to see whether a user would like to receive all his mail on another machine and automatically forward it. This will diminish the need for the *netmail* command.

Printing on remote lineprinters

The *netlpr* command takes a list of arguments as files to be printed on a remote lineprinter. Unfortunately, there can only be one standard input file for the remote command, so each file is sent as a *net* command executing the command *lpr.*

Other System Components

The ‘‘.netrc’’ file.

A user must specify defaults for frequently repeated options on a per-machine basis. This is done in a file ‘‘.netrc’’ in the user’s login directory, and is fully described in the ‘‘An Introduction to the Berkeley Network’’.

The *netq* command.

To see the network queue, the user must type the *netq* command. It lists the queue for each directly-connected machine, in the order requests will be sent. Each request is listed, one per line, giving the local and remote machines, the destination machine, the time sent, and the command to be executed. Commands which are internal to the network are called ‘‘responses’’ in the *netq* listing.

The *netrm* command.

Requests may be removed from the send queue using the *netrm* command. Since the originator of the file owns the queue file, a simple user-id check suffices to validate permission. Unfortunately, this notion breaks down for queue files of requests on intermediate machines. On such a machine an appropriate user does not exist, and the files are owned by ‘‘root’’. There is an option to *netrm* to remove all the user’s queue files, to make *netrm* easier to use.

The *netlog* command and other information.

A number of log files are kept by the network. Users may execute a *netlog* command which prints the last few entries of a log of commands sent and received. Also listed is the user, the time of the entry, and the return code of the command.

A more unreadable logfile is ‘/usr/net/plogfile?’. This log file has all the information of *netlog,* in a more cryptic form, along with trace information about net errors. The beginning and ending of sending and receiving are noted. This way the exact state of the network can be determined.

Hourly and daily statistics in a file '/usr/net/netstat?'. The number of bytes transmitted, the number of commands, and a breakdown of their type, and system load is recorded. This information is recorded in both hourly and daily form to track the performance of the network under different system loads.

Every hour, a *netq* command is executed and the number of queue entries is recorded in a file '/usr/net/netqstats'. This gives an estimate of the queue length.

Finally, the login names of each local user are recorded in a file '/usr/net/usernames'. Periodically, these names are sorted and duplicates removed. This gives a complete listing of network users, useful for sending network-specific mail and for general interest.

Protocol Explanation

The network uses three distinct levels of protocol. The highest level of protocol (the ''command'' protocol) refers to the organization of the information sent to the remote machine. An intermediate level splits such a stream into distinct numbered packets with a small header in each packet. The lowest level refers to the appearance of these packets on the hardware connection. At the present, this is a modified 6-bit ASCII code. Each of these layers is distinct, and presents the interface through procedure calls.

The Command Protocol

Each machine sends a request using a precise command protocol involving a header, the command, and any associated data.

```
length     header     command   ... data ...
```

All but the length field is formatted by the *net* command before the file is queued for transmission. The length is used to detect abnormally short, and poorly-formed, requests. The header includes all the information to route and verify the request. It includes

a) the origin and destination machines,

b) the login names on both machines,

c) a version stamp for this command protocol,

d) the time sent,

e) information about the originating terminal, and

f) the pseudo-command.

The pseudo-command is read by *netq,* and instead of printing the actual command being executed, prints something more appropriate. All the commands which use *net* (e.g. *netcp)* use the pseudo-command to list themselves rather than the command they are executing on the remote machine.

In order to be able to print the data on a normal UNIX terminal for debugging, the fields within the header are variable-length ASCII, separated by colons (':'). This forces the daemon to parse the header information and requires that literal colons (e.g. in the command being sent) be properly escaped within the protocol, but I felt the alternatives of using byte counts or fixed-length fields were too difficult to debug. The shortest header is approximately 85 bytes. Fortunately, this means the shortest command will fit into a single packet.†

The Packet Protocol

The above information is conveyed to each machine as a stream. This is done using subroutine calls to read and write data of arbitrary length over the link. The write procedure breaks the information into a set of numbered packets, with a length and exclusive-or check-sum in a header:

```
length     seqnum     type  chksum    ... data ...
```

† (less than 100 bytes, see below).

The length, type and checksum are one byte each, and the sequence number is two bytes. Since the packets are variable length the checksum is in the header rather than at the end of the packet to avoid the extra computation required to access it.

Each packet is transmitted over a link to a listener. Normally an acknowledgement packet is sent back. If there is an error, nothing is sent back, and the sending end will retransmit after a certain number of seconds.

There are no windowing or piggyback acknowledgements for two reasons: 1) this scheme is very simple to implement and 2) the error rate if each packet were not acknowledged would be high because of the hardware involved. If the future, I hope that both hardware and kernel device drivers will allow this improvement.

The so-called ‘‘rendezvous’’ protocol, whereby two daemons agree to communicate, is a simple ‘‘contention’’ scheme. When one daemon wants to transmit, it sends a special packet ‘‘reset’’ to the possible receiver, then transmits his first packet. Normally a daemon able to receive listens for a ‘‘reset’’. If it receives one, it enters a section of code designed to receive a header command, and data, and ultimately will execute it. If not, after a prescribed time interval is checks to see if there are any requests to send. Should both send at once, the characters may be garbled, or both may receive resets at the same time. In each case they both will retransmit. Each has a randomizing factor to break any ties which might develop.

In retrospect, this protocol is very primitive. Now that the network is in production use, the extra acknowledgements and separate ‘‘reset’’ are too expensive. A redesign would modify the protocol to transmit more than one packet before acknowledging it (ACKs), use negative ACKs to indicate immediately that an error has occurred, and eliminate the separate ‘‘reset’’ entirely.

The ‘‘rendezvous’’ protocol consumes a fair amount of time when both daemons choose to send packets. The alternative of constantly sending status packets when the daemons would be idle was never seriously considered because it was felt that the daemon should have as light a system load as possible; it seems now the daemons are busy most of the time and thus the initial connection tradeoffs should have been studied more closely.

The Low-Level Protocol

The network transmits over TTY lines through terminal interfaces and system drivers which behave as if the characters coming from another machine are from a terminal. This mode was chosen because it is absolutely the simplest, cheapest interconnection scheme possible. Unfortunately, the UNIX terminal drivers cannot accept 8-bit bytes unless they are in *raw* mode. This was judged to be a high system load, so the TTY lines operate in *cooked*, the reverse of *raw*, mode. In this mode certain bit combinations, e.g. ASCII newline and escape, do not transmit through the terminal driver to the user’s program but rather are interpreted as control information.

After much experimentation, the following transmission method was chosen. Each 3 byte triple is viewed as 24 bits, and broken into 4 6-bit groups. Each 6-bit number is in the range 0-63, and is added to a constant representing the lowest acceptable character code (a blank) in the ASCII sequence. This is sent as an ASCII character to a receiver who gathers 4 bytes, subtracts the increment, and shifts the 4 6-bit groups into 3 bytes. This represents a 3 to 4 expansion of all characters over the link, or a 33% loss of bandwidth.

In retrospect, this expansion has a considerable cost. The most scarce resource in the network is the actual hardware speed of the links. The alternative of using raw mode was never seriously considered.

The implementor’s hope is that better hardware will make better middle- and lower-level protocols easier. Until then, the difficulties of using TTY lines efficiently make further protocol improvements unlikely to yield big increases in speed.

A Note About Features this Protocol Lacks

In UNIX a process may only read or write one I/O device at a time. A daemon approach requires a single process reading and writing on a link to another machine. This process must decide who will receive this packet. I judged (correctly) that this decision was hard to schedule using UNIX pipes and signals, and only allow one kind of communication between daemons. This also makes it almost impossible to forward packets through intermediate machines. Thus intermediate machines copy whole requests before sending them again.

If the design specification required a simple packet-oriented driver within the system, the UNIX kernel could decide which of several special files this was destined to, and allow much more intermixing of traffic than before. I did not realize the importance of this and, in retrospect, would have chosen the other of the two approaches.

Portability

The acquisition of VAX/UNIX (Version 7) and the divergence of the Computer Center and Cory Hall Version 6 systems made the portability of the network source code important. Until then, the source code on all machines was identical. Fortunately, the UNIX implementors encountered these same problems and developed a number of facilities the network now uses.

Since many system calls use machine and version dependent data fields, so-called ‘‘include’’ files are available to hide the system differences and help standardize the system interface. The conditional compilation feature of the C language was used to select which kinds of code to generate, when the ‘‘include’’ files were not sufficient. Roughly, the following command included at the beginning of each C module:

```
# include <whoami.h>
```

would define which system, by name, the code was run on. For example, the above defines ‘‘VAX’’ on the VAX machine, and then lines such as

```
# ifdef VAX
 .
 .
# endif
```

control the code generated. In the network, sequences such as this in turn define other sequences, such as

```
# ifdef CORY
# define FUID
# define OLDTTY
# define PASSWDF
# endif
```

defines the unusual features of the Cory machine: the combined user-id and group-id returned by the *getuid()* system call, that it uses the old 1-character terminal names, and that it has a split password file for security reasons. Each of these symbols, e.g. ‘‘FUID’’, is tested in order to compile the correct code for that feature.

To help in isolating the changes, attempts were made to create a procedural interface to hide machines differences. These procedures are all in one file. Only one or two cases exist of conditional sections of code not in ‘‘mach.c’’ or ‘‘mach.h’’, its header file.

One problem these features pose is testing changes − the conditional sections hide errors in inapplicable code. A regimen was adopted: Testing was first done on the VAX (Version 7), then, after it was stable for a few days, moved to Cory, where typically there was some Version 6-dependent error, and after that was fixed and stable, it was moved to the Computer Center machines. This notion of a ‘‘testing’’ machine is very important − the VAX always has an up-to-date copy of the network source, even though other machines may lag in improvements.

There is now a ‘‘retrofit library’’ that simulates many of the features ‘‘mach.c’’ provides. It was not stable enough when the network was converted to Version 7, otherwise I would have used it.

At this point, when the entire source for the software for the network is transferred between machines only the first five lines of the ‘‘makefile’’ need be changed.

Security

Over Christmas vacation of 1978, a 15-year old high school student repeatedly broke into the Computer Center and Cory machines. He was able to use the network to gain access to privileged files on the VAX, and the fear of protection ‘‘holes’’ caused the staff to take the network down for seven weeks.

There were two security problems posed by the network:

1. The threat to the ‘‘root’’ account on another system.
2. The threat to a user’s remote accounts.

1. Threat to ‘‘root’’

Originally the network would allow a user logged in as ‘‘root’’ on one machine to execute any command as ‘‘root’’ on another network machine. As far as we know, this feature was never used to break into a system. However, the network has been changed to prevent a user from logging in as ‘‘root’’ on another machine, regardless of the password. This check is performed on the sending machine. Since ‘‘root’’ could conceivably circumvent this check by altering the command, the receiving end of a command checks the user-id of all commands being executed. If it is zero (i.e. ‘‘root’’) only a set of five commands is allowed, all needed by the network internally, and believed ‘‘safe’’.

We believe this makes the network ‘‘safer’’ than many local machine features such as the use of dial-up lines.

2. Threat to user’s remote accounts.

If a user places remote passwords in his ‘‘.netrc’’ file, an illicit superuser could get the password to all the user’s remote accounts. Even if the user does not care, system managers dislike this because the illicit superuser could now use a legal account on another system to break into it.

We have no good solutions to this. Users are now warned of this danger in the documentation, and a ‘‘.netrc’’ file with passwords must be readable only by the owner of the file.

Various solutions have been proposed:

a) Disallow passwords in ‘‘.netrc’’ files.
   Unfortunately, heavy network users would have to repeatedly type their password.

b) Encrypt the ‘‘.netrc’’ file.
   A program would have to exist to decrypt the file. A superuser could get access to whatever key technique that program used, if it were on the local machine. A public-key encryption scheme would make this option possible. We decided it was too much work to implement this.

c) Once-a-session passwords.
   In this scheme, a user would register his password when he logged in, then use the network without needing to type in a password. When he logged off, the password would be removed. We discarded this because we could not guarantee the password would disappear unless we wrote a daemon, which itself could be compromised. The best solution along this line uses the ‘‘alias’’ feature of the C shell. Each net command is aliased with itself and the **−l**, **−p** options. When the user logs in, he sets a shell variable to his password. When he logs off, his shell dies and the passwords are forgotten.

I believe the current alternatives are sufficient for a conscientious user to protect himself and still have an easy-to-use network interface.

Future Plans

1. Hardware

The net has suffered with low speed hardware. Short-term plans include speeding up the current terminal interface hardware from 1200 Baud to 9600 Baud and writing a driver for the device to bypass the internal UNIX character queues. This driver will improve the reliability of transmission and decrease the character interrupt overhead. The speedup from 1200 Baud to 9600 Baud may overload the systems due to the number of hardware interrupts it causes.

In the longer term, the EECS Department is considering acquiring direct-memory-access (DMA) devices such as the Logical Network Interface (LNI) or the Digital Corp. DMC-11 links for high-speed transmission. These devices are capable of over 1-megabit speeds, and would increase the speed of the current network by factors of hundreds.

2. Adding More Machines to the Network

The INGRES Research group and various other research units within the EECS department have expressed interest in being added to the network.

3. Remote Use of the Typesetter Facilities

The Computer Center A machine has a Graphics Systems phototypesetter and the VAX Research machine has a Versatec 36'' dot-matrix plotter with a *troff* simulator. Software now being debugged will allow remote use of the typesetter by running the *troff* program locally and sending the typesetter device codes to the remote machine.

This will distribute the typesetting load and help overloading on the A and VAX machines. It will also allow the use of *troff* macro packages only available on some machines.

4. Remote Mail Forwarding

The UNIX mail programs will be modified to forward mail to another account on another machine, allowing a user with accounts on many machines to read it all on one designated machine.

Summary Points

The author would like to stress these points about his experience:

1. Success in building networking software depends on having ready access to the correct hardware. The minimum is two terminals connected to two usable machines with two magnetic tape drives or some other existing means of software transfer.
2. Design in portability and security. More careful attention to machine dependence and security in the first phase would have saved much later re-programming.
3. Develop good local debugging techniques. The ''self-loop'' trick for network debugging depends on the accuracy of that simulation. UNIX pipes, for example, were not sufficient to simulate TTY lines because TTY lines are 7-bits with a restricted ASCII range.
4. Encourage users to use the system. Their feed back is important. However, it is necessary to have an unused network link for new protocol development, etc.
5. There is a fine line between support of an existing network and research. In the best of all possible worlds, support of research-developed software would be the responsibility of the systems staff for the machines it runs on. This is seldom the case.
6. The concept of layers of networks was very helpful in this project. There appear to be these levels:

user interface

queues and daemons

command protocol

packet protocol

transmission protocol

teletype lines

These levels are quite distinct. If a new, better network not involving queues is built, the transfer of files could still be by *netcp*. If state-of-the-art link hardware is used, perhaps all of the levels below the command protocol could be discarded.

7. The chief virtue of the current system is its extreme flexibility and low start-up costs. No modifications to the UNIX kernel are required and all local features are conditionally specified in a header file.

8. Networks are hard to build because

   a) They involve mutually-cooperating copies of software on (usually) differing computers.

   b) Many options are not practical because of compatibility considerations − new networks need drivers in unchangeable systems, and new protocols have to accept the old protocols until the old protocols are extinct.