Center for Advanced Computation

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

CAC Document Number 227
CCTC-WAD Document Number 7509

Networking Research in Front Ending and Intelligent Terminals

Experimental Network Front End Experiment Plan

May 16, 1977

CAC Document Number 227
CCTC-WAD Document Number 7509

Networking Research in Front Ending
and Intelligent Terminals

EXPERIMENTAL NETWORK FRONT END EXPERIMENT PLAN

by

Geneva G. Belford
Daniel E. Putnam

Prepared for the
Command and Control Technical Center
WWMCCS ADP Directorate
Defense Communications Agency
Washington, D.C. 20305

under contract
DCA100-76-C-0088

Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

May 16, 1977

Approved for Release: [illegible]
Peter A. Alsberg, Principal Investigator

Table of Contents

Summary

Background

Under contract DCA100-76-C-0088, the Center for Advanced Computation (CAC) of the University of Illinois at Urbana-Champaign is investigating the capabilities of network front ends. As a part of this contract an experimental network front end (ENFE) is being developed to interface a WWMCCS H6000 to the ARPA Network and to conduct experiments with a host-to-front-end protocol. The experimental network front end is being developed on a DEC PDP-11/70. The CAC has previously enhanced the UNIX operating system for PDP-11's in order to allow a PDP-11 to act as a mini-host on the ARPA network. Networking software developed by the CAC is being further enhanced to support the proposed ARPA Network Host-to-Front-End Protocol (HFP).

The experimental network front end will be evaluated in a series of tests and experiments. This document describes our current plans for the ENFE tests and experiments. Details are tentative and subject to change as the experimentation progresses.

Goals of Experimentation

The goals described in this plan fall into three categories:

1. performance testing,
2. fine-tuning of the system, and
3. investigating benefits to be gained from design changes.

Our effort during this contract year will emphasize the first category, performance testing. Major tuning and design changes within the current architecture are deferred to next year (Phase B).

Tests will be made to determine whether the system works as it is supposed to and is able to provide the front-ending facilities needed in the short term by the WWMCCS community. In particular, throughput and terminal support tests will be high priority.

Extensive timing measurements will be taken throughout the system. A major effort will be put into identifying those portions of the system that are relatively time consuming. In the course of testing and timing the system, we expect to do some fine-tuning and to identify design features that need improvement. These improvements are of secondary priority.

Tools for Experimentation

In order to carry out an experimental program, a number of tools are needed in addition to the ENFE itself. These tools include software (and possibly hardware) to carry out the necessary measurements, software and hardware to exercise the system (e.g., generate artificial traffic through the front end) and analytical tools for studying and interpreting the data. In this document we describe an optimal set of tools for thoroughly understanding the front end. The minimal set that we expect to implement under the current contract includes:

1. timing facilities for generating interrupts as specified by the experimenter and for timestamping messages;
2. artificial traffic generators (involving both software and hardware) at the three front-end interfaces (the interfaces to the host, to the network, and to the terminals); and
3. software to collect data as the experiments are run.

Other tools described in this document - such as a simulation program and queueing theory analysis - are essential to a follow-on study of how the front-end design may be improved. Time does not permit application of these tools under the current contract.

Specific Tests

In the last section of this report, we present a set of scenarios for specific experiments. These are necessarily tentative, particularly as to details. Here we briefly list the basic measurements which we plan to make. The listing is roughly in order of decreasing priority.

1. The throughput of the ENFE will be determined for the three major host-to-network paths:
   a. through the host-host service module (basic network access);
   b. through the program access service module and thence through user telnet; and
   c. through the server virtual terminal service module.

   Statistics on the message transit times along these paths will also be collected.
2. The terminal-handling capacity of the ENFE will be determined. The two paths through the ENFE to be exercised are those
   a. to the local host and
   b. through user telnet to the network.
3. Timing measurements will be taken between key points in the ENFE. These will determine
   a. how long the individual modules take to handle a message and
   b. how long the inter-process communication mechanism takes to relay a message from one module to another.
4. Data will be collected relevant to space allotment and memory management problems. Such data will include queue and table sizes, failure frequency of the message relay mechanism, etc.

5. Speed tests will be made of the hardware bit-streamer (to carry out the 8-to-9-bit conversion). Comparison tests with software to make the conversion will also be made.

The basic tests and measurements described above can all be carried out with software and hardware that is (or will be) available at the CAC.

Goals of Experimentation

Introduction

The ultimate goal of this experimental project is to develop a front end which works as well as is possible within the broad constraints imposed by building it on top of the Unix system in a PDP 11/70. To this end, a thorough program of testing and evaluation must be implemented. This program must be interactive with a program of fine-tuning and design modification. That is, a comprehensive program of experimentation will involve experimenting with design changes, as well as carrying out tests on the initial design. Because of time constraints, however, no more than trivial tuning of the system can be carried out under the current contract. Experimentation will be limited to making basic tests. In the first subsection below, we briefly describe the short-term goals of the basic testing program. In the second subsection, we describe some problems of fine-tuning which will be addressed this year insofar as time permits. For completeness, the last two subsections describe a set of long-term goals which we should plan to meet in the future.

Testing of the ENFE

General performance. The primary goal of the experimentation is to thoroughly test the experimental network front end - basically, to determine whether the system works as it is supposed to and is able to provide the front-ending facilities needed in the short term by the WWMCCS community. To this end, the throughput and response of the front end will be measured for its various functions and under varying system loads. An attempt will be made to identify points in the system where bottlenecks might occur under actual running conditions. The system will also be examined for potential deadlocks.

Terminal-handling capacity. Since it is anticipated that an important function of the network front end will be to provide an interface to user terminals, a study will be made to determine what level of service can be provided to how many terminals. What is desirable, in fact, is a curve giving response as a function of the number of terminals connected to the front end.

Extensive timing measurements. In order to make the evaluation as comprehensive as possible, timing measurements will be taken throughout the system. Modules which seem anomalously slow will be examined to see whether the slowness is a result of the design or of poor coding. A major effort will be put into identifying those portions of the system that are relatively time-consuming, since those are the places where modifications might have a sizable impact on system performance.

Fine-tuning the System

Space allotments. A number of minor design features are not being firmly decided upon _a priori_ but will be determined through experimentation. Chief among these features are the space allotments for various buffers, queues, tables, etc. For example, the inter-process communication (IPC) mechanism requires that each process have an associated IPC queue where it looks for "events" and for notifications of messages. Presumably, some maximum length must be assigned to these queues. What is this maximum? Should it be different for different processes? Unless the front end contains unlimited memory space, these questions will have to be answered.

Similarly, the transfer of messages requires that the messages be put into memory segments. How much front-end memory should be allotted to these segments? Should a particular block of memory be specifically allocated to segments? If so, one must choose the size of this block carefully. If not - i.e. if segments may be placed anywhere in free

core - there is a strong likelihood that the memory will become fragmented. This will make it increasingly difficult to find long segments.

Fragmentation, if it occurs, can cause other problems. If there is a shortage of memory, service modules may be swapped into and out of the system. For example, as the system presently runs, the NCP Daemon can be swapped out. Since the NCP Daemon is fairly long (10K words), memory fragmentation could keep it from being swapped back in. There is even a potential for deadlocks here. Segments containing messages addressed to an out-of-core module could prevent that module from being swapped in to read and release those segments.

There are other strategy questions with respect to the handling of segments. For example, a maximum length for the segment descriptor table must be specified. Also, the space allocation strategies are different for small and large segments. The dividing line between "small" and "large" needs to be set.

8-to-9 bit conversion mechanism. The particular hardware configuration being used in this experimental project has caused a bit-conversion problem. The Honeywell H6000 uses 36-bit words, while the DEC PDP-11 uses 16-bit words. Thus, communication between the host and the front end requires translation between 8-bit and 9-bit bytes. We are purchasing a hardware device (a "bit streamer") to effect this translation in the front end. Preliminary study indicates that the bit streamer is more than twice as fast as a software translator would be. However, against this one must balance the cost of the bit streamer. The cost would be particularly important if it appears that more than one bit streamer will be needed to handle the load. Furthermore, it may be that the higher speed of the hardware translator has a negligible impact on overall front-end performance. Since a software translator is

easy to code, the whole question of whether the 8-bit to 9-bit conversion should be done by software or by hardware is subject to study by experimentation.

Message size on link. As was noted in the H6000 specifications [CAC Document Number 220], the maximum message size for the host-to-front-end link should probably be adjusted after experimentation. The size for this experimental project has initially been set at 3600 bits. It may be that a larger or smaller value may improve throughput. This parameter is, of course, not an integral part of the front-end design but an installation parameter. It would be useful, however, if we could determine an optimum or near-optimum value.

As experimentation and testing progresses, other opportunities for fine-tuning the system are likely to appear. The design aspects discussed above are not meant to form an exclusive list, but are merely those which at this time appear to us worth studying.

Investigation of Potentially Useful Design Changes

There are a number of design features which we would not wish to alter in ther short term, but which should be studied with an eye to changing them in the future. We are not talking about studying the effects of major design changes - i.e., changes in front-end architecture - but only of relatively minor changes which promise to improve performance within the existing architecture.

Inter-process Communication (IPC) mechanism. Two features which obviously come under this heading are the IPC and the associated non-blocking I/O mechanism. Since the existing Unix mechanisms for inter-process communication were totally unsatisfactory for a front end, we were obliged to design new mechanisms as a first step in designing the ENFE. Severe time constraints were placed on the project. More thought might yield significantly improved ways for the front-end processes

to relay messages to each other. The IPC and non-blocking I/O mechanisms will be examined especially carefully to see (1) whether they perform adequately and (2) if changing some aspects of them might improve throughput.

Moving modules into the kernel. An important aspect of the design to be investigated is the ultimate level of the various software modules in the front end. As a matter of convenience, most modules are being initially implemented as user-level programs. It is not clear, however, that they should remain so. Moving some of them into the kernel may well improve the performance of the front end. In particular, the heavily used channel protocol module (CPM) appears to be an especially good candidate for system-level implementation.

Flow control. Flow control is another area in which relatively minor design changes may improve throughput. The flow control mechanism between the host and the front end is specified in the host-to-front-end protocol. Changing this specification in the short term is not feasible. However, if experimentation identifies problems with this specification (e.g., if the mechanism appears to cause a bottleneck), then it will clearly be appropriate to attempt to improve on the HFP flow-control scheme.

Flow control between front-end modules is mainly effected by the IPC mechanism. The CPM can stop or start data transmission from the services by sending them appropriate IPC events. The CPM also controls flow in the other direction by keeping track of and limiting the number of unacknowledged messages it has sent to the various services. On the other side of the services (e.g., between the host-host service and the NCP) the non-blocking I/O mechanism automatically provides a primitive sort of flow control. In short, the front end's internal flow control is carried out by the simplest, most readily available means. The

impact on throughput is yet to be determined. Experiments should pinpoint any inadequacies in internal flow control and identify any paths requiring a more sophisticated mechanism.

Memory management. Some details of memory management, as noted above, will be studied during the fine-tuning of the system. A broader investigation is desirable. It may be that the front end as designed will have enough free memory space so that no conflicts will arise in normal use. However, the potential for serious trouble exists. There is no limit on the amount of message space that a service may request, nor is there any central control to limit the amount of space assigned to the individual services. A greedy service could lock up the system - or at least seriously reduce throughput - by grabbing too much memory.

In short, no matter how much memory the front end has, there is always the possibility of running out. Strategies for handling this possibility are not yet worked out. It may be that, if space is short, certain messages (perhaps some of those addressed to a module that has gotten behind on pickups) should be swapped out to disk. If swapping turns out to be desirable, a swapping strategy would have to be worked out with care to minimize the effect on throughput. Furthermore, an efficient swapping mechanism would have to be designed and coded. At the present time, disk I/O is blocking, so that frequent swapping could seriously impact the performance of the front end.

Evaluation of the ENFE Architecture

The program outlined above should give us a good feel for the potential of the Unix/11-70 front end. We will have thoroughly tested the capability of the system as designed and also as modified by some fine-tuning. We will have investigated whether minor design changes

within the current architecture are likely to improve performance significantly. If time constraints permit us to carry out this program completely, a natural final step is to put the results into perspective by evaluating the basic architecture of the ENFE. It should be possible to draw conclusions about the overall limitations of the architecture and to identify functions for which major design changes are advisable. This phase of the experimentation will provide valuable input into future research on alternative front-end architectures.

Introduction

In order to carry out an experimental program, a number of tools are needed in addition to the ENFE itself. These tools include software (and possibly hardware) to carry out the necessary measurements, software and hardware to exercise the system (e.g., generate artificial traffic) and analytical tools for studying and interpreting the data. This section contains a description of the measurement and traffic-generating tools that are being built to study the various aspects of the experimental system. In addition, we describe the ways in which the ENFE, together with measurement and message-generating software, will be configured for experimentation. Because of time constraints, it will be impossible to build or to use sophisticated analysis tools under the current contract. Because of the importance of such tools, however, we have included a discussion of them in appendix 1.

Measurement Tools

Clocks. Since throughput and response time are central issues in the operation of a front end, timing will play a key role in measuring the performance of the ENFE. Clock readings may be obtained in the PDP-11 in two ways. The first method uses interrupts generated by the cycles in the AC power supply. Unix handles such line interrupts in a routine called "clock" where, among other things, a timing variable is incremented. This variable can be accessed anywhere in the system and is effectively a clock which ticks sixty times each second. Actually, small variations in the line frequency and delays in handling interrupts may cause slight inaccuracy in these readings. The need to measure very small time intervals with high accuracy makes the use of this clock mechanism unacceptable for our experimentation.

The second method for obtaining clock readings provides for more accuracy and flexibility through the use of a programmable clock. The CAC owns such a clock, which will be used in the experimentation. The programmable clock is a hardware device containing a crystal-controlled clock whose operation can be set by the system to run in several different modes at several different rates. In the Repeat-Interrupt mode, the clock decrements a 16-bit counter at rates of 60 Hz, 100 kHz, or 10 kHz. When the counter reaches zero, an interrupt is generated and the counter is reset to a specified value. It then resumes counting down to zero. By varying the clock rate and the initial value of the counter, the clock can be set to interrupt at a wide range of frequencies. Furthermore, by resetting the counter to random values, time intervals obeying any prescribed statistical distribution can be generated.

The counter can also be read directly by the system. This feature will allow us to timestamp messages, collect data on the precise times of event occurrences, etc.

Unix histogram utility. Given a timing mechanism of either sort described above, the routine which handles the clock (or line) interrupts can collect statistics on which software module was in execution as the interrupt occurred. At each interrupt, the value of the program counter before the interrupt can be examined and a histogram can be built up which describes the relative amounts of computation performed in each section of a given program. There is already a Unix system utility called "prof" which enables a user to produce just such a histogram of his own program. To use this facility a user need only invoke the "-p" option on compilation and then interpret the results with the prof program.

It must be remembered that observations made at regular intervals are hardly independent. It is possible to conceive of a loop

which takes exactly one clock tick to execute. Observations of such a loop would lead to the erroneous impression that one small part of the loop took all the execution time. This effect is probably not important in practice. Since several hundred thousand instructions can be executed each second, there is plenty of room for random effects such as I/O interrupts to intervene between clock ticks spaced at, say, one sixtieth of a second. Still, prudence demands that we be concerned with this problem. We therefore plan to generate the monitoring interrupts at random time intervals. Use of the programmable clock provides us with a fine enough time scale so that inserting a random number of clock ticks between each successive interrupt is feasible.

With minor changes the present Unix monitoring system can be adapted to produce histograms useful in studying the operation of the ENFE. For each of the primary user-level modules of the ENFE, two histograms can be produced. The first monitors the computation performed in the sections of the module itself, and the second monitors computation performed in the kernel at the behest of the given user-level module. By combining the kernel histograms of each of the user-level modules, a picture of the overall activity in the kernel can be obtained as well.

Building the monitoring facility we need requires only a few alterations to the existing system. The routine that handles the programmable clock interrupts must be able to initiate histogram data collection when kernel processes are interrupted as well as when user processes are interrupted. Also, the system primitive "profil" that initiates this data collection must be altered to handle kernel mode as well as user mode. Next, the library routines employed by the compiler when the "-p" option is used must be altered so that they perform the same operations for kernel processes as for user processes. Finally, the prof program must be changed to handle the dual histogram data produced for each user module.

Thus, only minor changes to already existing features of the system will allow us to obtain a picture of the nature and level of the overhead incurred in the operation of the ENFE. Other information can be gathered as well; for example, data for histograms of queue lengths could be easily collected along with the data described above. Such histograms would show the distribution of queue lengths and perhaps indicate system bottlenecks.

Timestamping messages. As a message moves through the front end, it will be timestamped by recording in it the current clock reading as the message passes certain milestones. For example, such milestones might include the queueing of the message for service by the CPM or the completion of handling of the message by the CPM. The record of timestamps, readily stored in message TEXT, will provide a very complete picture of the message's progress through the front end.

The best way to make these measurements is to build a timestamping facility directly into the IPC or non-blocking I/O mechanisms, since it is there that messages are both enqueued and dequeued for passage to the service modules. It may also be appropriate to insert timestamping facilities at certain points within the service modules. As a message leaves the front end, the timestamp data can be read and processed or stored on disk for later processing and interpretation.

Monitoring memory usage. An important task in the experimental program will be to identify the effects of available memory on ENFE performance. When Unix is employed as a timesharing system on the PDP-11/50 at the CAC, a surprising degradation of response is sometimes observed with as few as six or eight users. The amount of computation directly required by these users does not by itself explain this degradation. One possible explanation is that Unix is spending large amounts of time swapping processes into and out of core. Processes in Unix can

spawn other processes, and each process requires at least 1K words of core. The average core usage is about 8K bytes per process. Just a few primary processes can therefore bring about a condition in which a high level of swapping occurs.

This problem has been anticipated in the design of the ENFE. The new IPC and non-blocking I/O mechanisms cut down on the creation of new processes. However, memory will be monitored in the experimental program in order to determine if further steps should be taken. First, it is easy to collect data on failures to locate a free memory segment. The code in Unix which attempts to locate free segments will be augmented to collect data on its success or failure. Second, the amount of core available to Unix can be restricted to determine the effects on all the other experimental measurements. If small reductions of core space drastically degrade performance, then it may be worthwhile to consider further design changes.

Dynamic modification of the amount of contiguous core memory available in an executing system is difficult to do properly. It is relatively straightforward to reboot the Unix system so that it will have a specified amount of memory. After the system has been taken down and is coming back up, it enters a loop in which it organizes available core. It is easy to arrange for this process to terminate prematurely. To study the effects of restricting memory, system performance will first be measured under an experimental load with full core space made available. In subsequent steps, the system will be taken down and brought back up with smaller amounts of memory and then subjected to the same experimental load.

Software and Hardware to Exercise the System

Generating and measuring artificial traffic.

In order to be able to interpret experimental measurements unambiguously, experiments

should be carried out in a controlled manner. Therefore, software modules to generate artificial message traffic will be built. Primitive message generators are needed for debugging purposes and will be available early in the project. The message generators for experimentation will have the capability of generating messages of prescribed lengths and at time intervals obeying a prescribed statistical distribution. In this way, the experimenter will be able to systematically vary the load on the front end and study the resulting performance.

Data collection facilities will also be needed. These facilities will probably vary with the experiment or set of experiments. A discussion of dynamic approaches to collecting statistics may be found in appendix 2. It has not yet been decided how sophisticated the data collectors need to be for the current project. Data collection facilities will be combined with the message generators into single modules. Thus, single modules will both send and receive messages, collecting statistics on the messages as they are received. Such modules will look much like simple host processes.

Artificial message traffic will be introduced at the three points in the ENFE where real traffic would enter. These points are:

1. the host interface,
2. the IMP interface, and
3. the terminal interfaces.

From these points of entry, the messages will flow along any of several paths through the front end. (See figure 1.) The particular path to be followed is specified as part of the message. The message generator will allow the various paths to be exercised according to a job mix prescribed by the experimenter. After traversing such a path, messages will be intercepted and the desired information will be extracted.

ENFE SOFTWARE ARCHITECTURE

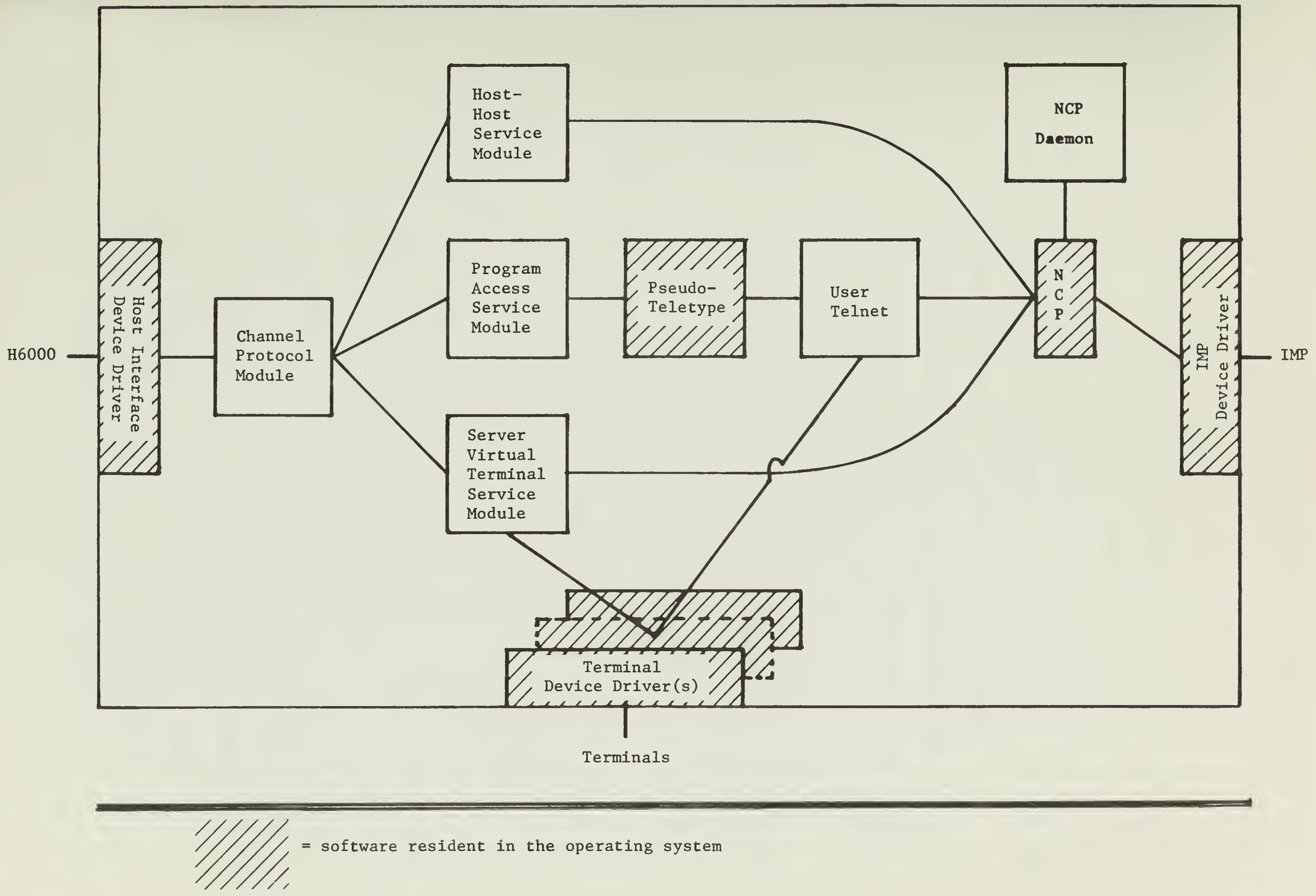
H6000
Host Interface Device Driver
Channel Protocol Module
Host-Host Service Module
Program Access Service Module
Server Virtual Terminal Service Module
Pseudo-Teletype
User Telnet
NCP Daemon
N C P
IMP Device Driver
IMP
Terminal Device Driver(s)
Terminals
= software resident in the operating system

Figure 1

The message generating modules could be connected to the system in various ways. That is, there are a number of ways in which artificial traffic can be introduced into and made to flow through the ENFE. The configurations differ considerably in ease of implementation and in the credibility of the conclusions which can be drawn from experiments using them. In the remainder of this subsection, we discuss mainly the configurations that we plan to implement. It may turn out to be necessary to make some changes in these as the experimentation progresses.

Configuration to exercise the internal ENFE software. We will first study the performance of the ENFE as an information relay between host and network. Therefore, at either end of the ENFE we will introduce independent streams of input traffic. This will be accomplished by placing message generators inside the front end to simulate the input of message traffic. This scheme provides the maximum control over inter-arrival times. In many ways this is the easiest approach to implement since everything is internal to the ENFE.

Since the CPM expects to receive messages from kernel-level device drivers, there is some difficulty in designing the software to generate simulated messages from the host. This software must both fit in with the system architecture and perform the desired function. It is possible to avoid this difficulty by supplementing the front end with an altered copy of the CPM that simulates the CPM of the local host. The message generator can then look much like a host process. This pseudo host process then relays messages to the pseudo host CPM, which in turn, following the host-to-front-end protocol, communicates with the front-end CPM. This strategy is illustrated in figure 2.

Since the CPM is a user-level program, the whole process of simulating messages from the host can be done at the user level. At the same time, connecting a pseudo host CPM directly to the front-end CPM will allow limited experimentation with the host-to-front-end protocol to be undertaken early in the project. Furthermore, the use of a copy of the already existing CPM may ease the effort of implementation. A disadvantage is that this approach introduces more extraneous software than may be necessary into the front end and therefore causes more interference with its normal operation.

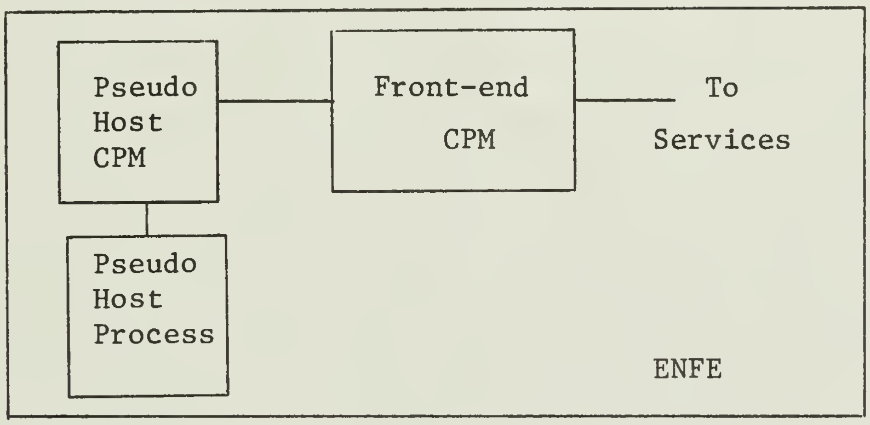

Figure 2

A Configuration for Introducing Pseudo Host Messages

On the other side of the front end, some other approach needs to be taken. Messages arriving from across the network must be simulated. We plan to do this by inserting a message generator (a "pseudo foreitn process") in the ENFE to send messages out through the IMP interface to be echoed back by the IMP. No special adaptations of the IMP need to be made. The headers of the messages need only point to the ENFE as the destination, and the IMP will relay these messages back to the ENFE as part of its normal function. This will be easy to implement, since the IMP device driver, the IMP interface and the IMP itself will stand in their usual relationship.

In summary, the basic configuration to be used in the front-end experimentation is pictured in figure 3.

More complex configurations. A major objection to the configuration described above is that the software inserted into the front end to generate artificial messages will interfere with the normal functioning of the front end. To fully remove this effect the ENFE would have to be tested in an environment in which the artificial traffic was generated by external machines simulating local and foreign hosts. On the host side this might be accomplished as follows. In addition to the PDP-11/70, which is being used for the ENFE, the CAC has two other PDP-11's (an 11/20 and an 11/50) which possess IMP interfaces. The data transfer mechanism of the host interface in the front end is similar to that of an ARPANET IMP interface. This could allow the IMP interface on one of the CAC PDP-11's to be connected to the interface on the host side of the front end. However, considerable software development in the additional PDP-11 would be needed in order for it to play the role of a front-ended host. At this point, it is not clear how many hardware/software problems could arise in attempting to set up such a configuration. In any case, we do not believe that an elaborate setup of this kind will be feasible during the current contract year.

If realistic network bandwidth and delays are desirable, messages could be transmitted between the ENFE and an actual foreign host on the ARPANET. The ultimate test will use the PDP-11/70 at Reston as a front end actually connected to the H6000. Realistic end-to-end testing with this configuration will be carried out by sending messages between the pseudo foreign process in the 11/70 at Urbana and a test process in the H6000 at Reston. This will be done after the latter system is fully integrated and checked out.

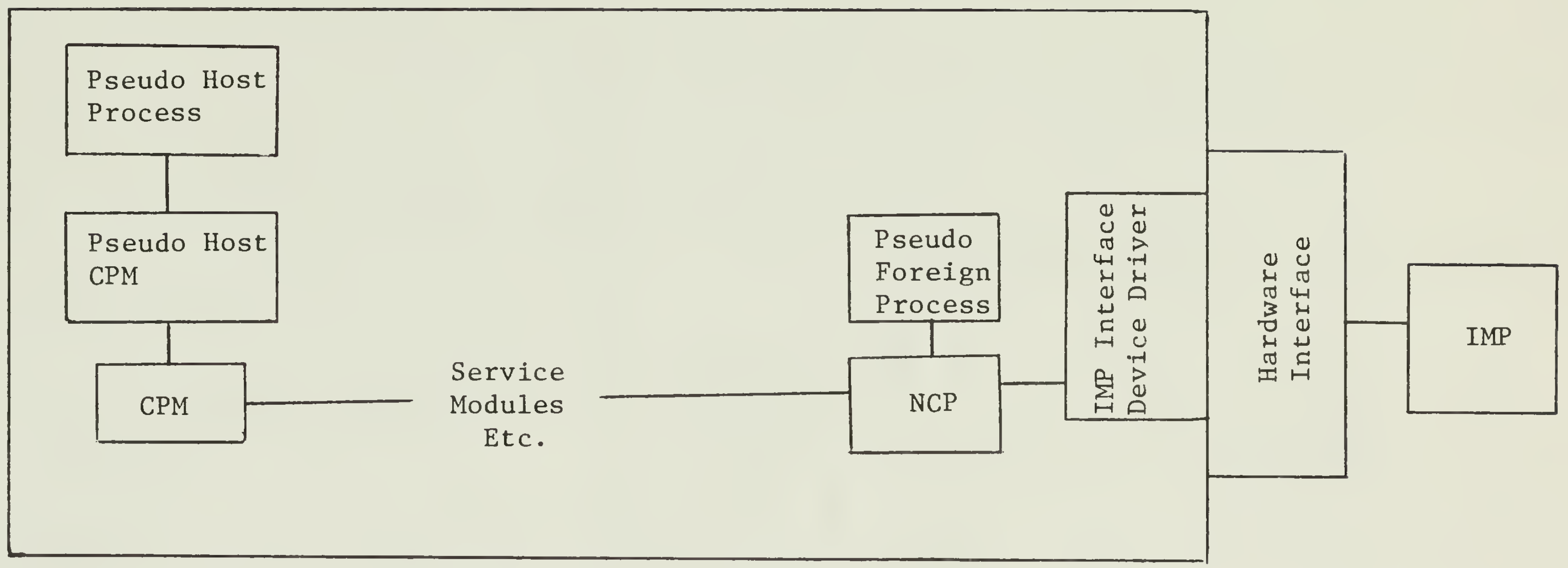

Figure 3

Configuration for Exercising the ENFE Software

Tests of terminal-handling capacity. Generating artificial traffic at the terminal interface presents problems not encountered for the host and IMP interfaces. In particular:

1. There are many terminals while there is only one host and one IMP.
2. Events that cause interrupts occur much more frequently. (In many cases, an interrupt will occur after every character input.) This makes simulation of terminal traffic by software within the ENFE difficult.
3. "Looping" the hardware interfaces (i.e., connecting output channels to input channels so that traffic can be generated within the ENFE, sent out through the interface and then returned to the ENFE) will cause automatic interrupts as desired. However, this approach produces very unrealistic loading of the interfaces, since terminal traffic is extremely asymmetric.
4. The behavior of humans sitting at terminals is erratic. Realistic simulation of terminal input is therefore probably impractical.

The way that we plan to study terminal-handling capacity is to use a separate machine to generate traffic on a terminal input line. For this purpose, we propose to use the intelligent terminal built by the CAC. The intelligent terminal has the capacity to output characters at a rate equivalent to that expected from several hundred normal user terminals. The terminal handler within the ENFE will have to be modified to fan out this bulk input, to make it look more like messages arriving from multiple terminal connections. From this point on, the messages can be sent along various paths through the ENFE. Alternatively, the traffic can at some point be turned around, with an "explosion" factor

to take care of the usual asymmetry in terminal traffic, and sent back out through the terminal handler to the intelligent terminal.

A significant software effort will be required to implement this plan. Since all details are not yet decided upon, it is difficult to predict precisely what will be needed. Minimum requirements will include:

1. modifications to the terminal handler,
2. a traffic generator in the intelligent terminal, and
3. data collectors within the ENFE (and possibly within the intelligent terminal).

Description of Experiments

Introduction

It is advisable to begin this section with a word of warning. Experimentation is an interactive process. Results of early experimentation may make other proposed experiments unnecessary and/or suggest new experiments that are not included in this plan. Basic tests, timing measurements, etc., will certainly be carried out. Suggested experimental details should be considered tentative. Details of more elaborate experiments will be worked out after results of the simpler experiments have been obtained.

The set of experiments outlined below involves making timing and other measurements of the internal ENFE software. These can be begun reasonably soon, as successive ENFE software modules are coded and debugged. More elaborate experiments, designed to test the actual performance of the ENFE as an interface between an H6000 host and the ARPANET, are dependent upon the timely receipt of hardware (e.g., the ABSI) and the timely completion of numerous interlocking software design and coding tasks. Time may not permit the undertaking of such ambitious experiments during the current contract year.

Performance Tests of the ENFE Internal Software

Single-message timing measurements. The experimental configuration shown in figure 3 will be used. The system will be preset to states allowing TRANSMIT Commands and associated data to be freely transmitted. That is, the pseudo host CPM, the front-end CPM and the service modules will see the logical channels that are used as ESTABLISHED. Other front-end software, such as the NCP, will be primed to handle and relay data as requested by the service modules. This initial setting up of channels and connections will be handled by sending appropriate BEGIN Commands from the pseudo host process in the ENFE.

Messages will be generated and transmitted through the front end. Single-message transit times in an otherwise empty system will be measured. Timestamps (stored within the message data field) will be added to the message at various points in the system, as specified below. Software modules will accumulate statistics on message transit times through the front end. There are three main paths through the ENFE. (See figure 1.) The paths are:

1. the path through the host-host service module,
2. the path through the program access service module and the user Telnet module, and
3. the path through the server virtual terminal service module.

Data may travel in both directions along all of these paths - i.e., from the host interface towards the network and from the network interface towards the host. Concentrating attention on the directed paths from the host towards the network, we list below the recommended points where the messages are to be timestamped. (Messages moving in the other direction will be timestamped at analogous points; i.e., when they arrive at and are relayed by the various modules along the path.)

1. Along the path through the host-host service (HHS) module (basic network access), the message will be timestamped immediately after
   a. the message is generated,
   b. the pseudo host CPM places a notification of the message in the front-end CPM's IPC queue,
   c. the message is picked up by the CPM,
   d. the CPM initiates transmission to the HHS module via IPC,
   e. notification of the message is placed in the HHS IPC queue,

f. the message is picked up by the HHS module,

g. the HHS module completes the WRITE of the data to the file associated with this logical channel, and

h. the data is intercepted by the statistics collection module.

2. Along the path through the program-access-service (PAS) module, and thence through the user telnet module to the network, the message will be timestamped immediately after

   a. the message is generated,

   b. the pseudo host CPM places a notification of the message in the front-end CPM's IPC queue,

   c. the message is picked up by the CPM,

   d. the CPM initiates transmission to the PAS module via IPC,

   e. notification of the message is placed in the PAS IPC queue,

   f. the message is picked up by the PAS module,

   g. the PAS module completes the WRITE of the data to the file associated with this logical channel,

   h. the user telnet module completes the WRITE of the data for processing by the NCP, and

   i. the data is intercepted by the statistics collection module.

3. HFP Messages arriving over the link to the host and designated for the Server Virtual Terminal Service (and thence to the network) follow a path completely analogous to that described for the host-host service. Messages will therefore be timestamped at points analogous to those listed under 1, above.

It is likely that not all of these timestamping points will be necessary for single-message timing in an empty system. For example, a negligible amount of time may occur between points e and f, above. However, if we build in the timestamping apparatus at these points, we will then also be able to monitor the time spent in IPC queues when the system is being exercised more realistically with multiple messages.

The reader might also notice that some of the timestamping points defined above are somewhat vague. A certain amount of flexibility was purposely left in for the convenience of the implementor. Furthermore, the exact point of the timestamping cannot always be clearly specified without reference to the actual code.

In carrying out the experiments, it will probably be convenient to exercise paths 2 and 3 together. Messages will be generated to carry Telnet data addressed to the local host but to be transmitted via the network. These will then flow along path 2 (user Telnet) to the IMP, where they will be echoed back. Upon their return, they will automatically follow the server Telnet path (path 3) through the ENFE. Artificial traffic simulating data from the host Telnet service back to the terminal will follow this loop through the ENFE in the reverse direction.

<u>Multiple-message timing measurements</u>. The three paths described above will be exercised simultaneously and under varying loads. The message generator at the host interface will be enhanced to generate messages at random times and addressed to random service modules. This should mimic the arrival of HFP Messages from various host processes. The mean arrival rate, number of channels, and job mix (what fraction of the Messages are addressed to which service module) will be specifiable by the experimenter.

The message generator at the IMP interface must be similarly enhanced to generate messages randomly. It is probably reasonable to build in some correlation between the number of messages generated for the different paths at each end of the system. For example, if the host is supposedly generating heavy host-host traffic, one would expect fairly heavy traffic in both directions along the host-host path.

The messages will be timestamped as they progress through the system. The timestamping software will be identical to that used for timing single messages.

The statistical data collection modules will require a small enhancement. They will have to sort the data with respect to which of the three different paths the data traveled on.

The examination of timing statistics for varying arrival rates and job mixes will provide us with a large amount of information on front-end performance and data-handling capacity. The data may also identify bottlenecks, points where the software seems anomalously slow, points where data traveling on one path may impact on other paths, etc.

<u>Other measurements under multi-message loading</u>. Either in parallel with the timing measurements or as a subsequent task, other types of data will be collected. Most of these will be concerned with the memory management and space allotment problems discussed earlier in this document. The data collected should include the following.

1. Average length of each of the IPC queues. This can be measured in either of two ways (or both, for a consistency check):
   a. Part of the data which a message carries along with it might be its original position (e.g., no. 5) in each of the queues it is placed in.

   b. The software monitor might be set to collect data on queue lengths at the time of scheduled interrupts.

2. Failure rate for various features of the message transmission mechanism. Counts should be kept of the number of times (in a known time interval and for a known total number of messages) that each of the following happenings occur:

   a. No free map register is available when requested.

   b. No memory segment of the requested length is available.

   c. The sending of an IPC event has failed because of a lack of room in the IPC queue.

   d. A request for a segment has failed because the segment descriptor table is full. (For more data on the requirements of this table, one might want to monitor its length, as described above in 1b for queue lengths.)

<u>Tests of the ENFE as a Terminal Handler</u>

We believe that the best way to study the terminal-handling capacity of the ENFE is to attach an auxiliary computer to the terminal interface and use that computer to generate artificial terminal traffic. The configuration that we propose to use for this project has been described briefly above (p.23). As we indicated there, details have not yet been worked out. A considerable software effort is anticipated. It will probably not be possible to carry out sophisticated experiments. The basic goal is to determine how many terminals the ENFE can effectively handle. To this end, the amount of bulk input from the auxiliary computer will be increased until it appears that the ENFE can no longer handle it. By making reasonable assumptions, we expect to be able to translate this upper limit on bulk input into an upper limit on number of terminals.

Such an upper limit is only a gross measure of terminal-handling capacity. There will probably be no obvious cutoff point where the ENFE "can no longer handle" the traffic. What usually happens is that system response degrades at an ever increasing rate as terminal traffic increases. We would really like to determine the entire curve describing response vs. terminal load. The timing facility within the ENFE will be reasonably adequate for this purpose only if most of the time is consumed inside of the ENFE. If the terminal interface turns out to be a bottleneck, measurement of response will require addition of timing facilities to the auxiliary computer that is sending (and receiving) the simulated terminal traffic. Furthermore, we envision the bulk input as a character stream. Turning it into a sequence of messages - perhaps with timestamps - for timing purposes will increase the software development effort nontrivially.

Some basic system measurements, such as the time to handle a keystroke, can be made early in the experimentation. Simple data of this kind should be adequate to allow us to make a preliminary judgment as to the capability of the ENFE to handle large numbers of terminals.

Experiments with More Complex Configurations

At the present time, it is not clear how much additional information is to be obtained by attaching the front end to a host, sending messages over the network, etc. However, such thorough, realistic testing of the entire system will be carried out if time permits.

References

1. Denning, P.J. and Eisenstein, B.A. "Statistical Methods in Performance Evaluation", ACM Workshop on System Performance Evaluation, April 1971.

2. Kemeny, J.G. and Snell, J.L. Finite Markov Chains, Van Nostrand, Princeton, NJ, 1960.

3. Kleinrock, L. Queueing Systems. Vol. 1: Theory, Wiley-Interscience, 1975.

4. Kleinrock, L. Queueing Systems. Vol. 2: Computer Applications, Wiley-Interscience, 1976.

5. Scherr, A.L. An Analysis of Time-shared Computer Systems, MIT Press, 1967.

Appendix 1

Analyzing and Understanding the Experiments

Introduction. It is not enough to run numerous front-end experiments and gather mounds of raw data. The data must be organized and interpreted if we are to learn anything from the experiments. In this section we discuss several analytical tools which may be useful in understanding the experimental results. First, we discuss the event-driven simulation system that is available for our use. Simulating the front end - or key portions of it - may allow us to identify design problems which are obscured by the complexity of the behavior of the real front end. Timing experiments run on the front end would provide parameters for the simulations, as well as rough checks on simulation validity. Simulations could also interact with experimentation by calling attention to gaps in the data needed to understand system behavior.

Second, we discuss two closely related analytical tools - Markov chains and queueing theory - which may be useful in understanding certain very limited aspects of the system, such as the flow-control mechanism between two modules. Questions of desirable buffer or queue capacity could also be addressed. In fact, such questions are probably best investigated theoretically and the results checked experimentally. It would be wasteful and time-consuming to fine-tune the front-end system by exploring experimentally all possible system variations, when analytical guidance is available.

System simulation. Simulation is an important tool in computer system analysis. By realistically mimicking system behavior under various conditions, bottlenecks can be identified and overall performance can be studied. In experimentation, cause and effect can be obscured by the complexity of the system under observation. A simulation study can help to interpret ambiguous experimental results. Simulation can also be useful in fine-tuning a system. It is often much simpler to test a proposed system change by making the change in a system simulator than by actually changing the implementation.

A general, event-driven, simulation system is available at the CAC for system analysis. It was written in C language for Unix by a member of our research group. The system is very flexible, allowing for both deterministic, scheduled events (e.g., the arrival of a previously dispatched message) and stochastic events (e.g., the generation of a message by a process).

The major components of the system are an event generator, a general queue manipulation routine, and a set of statistical routines. The event generator is the heart of the simulator. Given a list of event names, the associated event distributions (i.e., probability distributions for occurrences) and initial instances, this routine generates the corresponding sequences of events. Examples of events are the arrival and pickup of messages. The events are generated randomly with the prescribed probability distributions. The queue manipulation routine handles the addition and deletion of events from queues, properly maintaining the queue linkages. The statistical routines gather data on events, as requested by the user. Routines are available to compute statistics up through second order. In addition, there is a provision for the user to write his own data analysis routines.

Even though this simulation system is available to us, it would require a considerable amount of additional work to set up a realistic simulation of the entire experimental network front end. Simulations also tend to be expensive, having to run for long periods of time to generate statistically valid results. For these reasons, it is unlikely that full-scale simulations of the front end will be carried out as an adjunct to experimentation. Simulations of selected features of the front end (such as the HFP flow control mechanism) may, however, help to identify places where small changes in the front-end design could lead to performance improvement.

Markov chains. The main disadvantage of simulations - namely, the long run times required to generate valid statistics - can be avoided by an alternative approach. This approach is to model certain aspects of system behavior as stochastic Markov processes [2]. The idea is to first define a set of states describing the subsystem to be modeled. The system moves from state to state as time passes. A transition matrix T can be defined with elements T_{ij} giving the probability of a transition from state j to state i in time Δt.

The matrix T can then be used to obtain steady-state population densities, or probabilities that the system is in a given state. Suppose that p is a column vector whose components p_j are the probabilities that the system is in state j. Then the steady-state probabilities must satisfy

$$Tp = p.$$

This homogeneous system of linear, algebraic equations may be solved for p by one of the numerical techniques available for large, sparse matrices.

The problem is that for a system of any complexity the number of states, and hence the matrix T, is enormous. We used a Markov model for a preliminary study of a flow-control scheme and found several hundred states necessary. To specify a state for flow-control purposes requires specifying the number of full (or empty) buffers at each end and the locations (necessarily discretized) of all messages and acknowledgements in the system. Nevertheless, we found that our Markov model readily yielded results on throughput and buffer utilization. (For example, if states 1 and 2 are the only ones for which all send buffers are full, then $p_1 + p_2$ gives the fraction of the time that all send buffers are expected to be full.)

A computer program for analyzing Markov models of flow-control schemes was written during our preliminary study. It is available for further study of flow control and associated buffer management. We propose to make use of this program, with some modifications, to gain a better understanding of the flow-control mechanisms within the front end and between host and front end. Like simulation, Markov analysis could be very helpful in fine-tuning these aspects of the system. Unfortunately, the proliferation of states with system complexity makes it unlikely that this tool will be useful on a broader scale.

Queueing theory analysis. Queueing theory is the study of waiting lines for a service, or set of services. Jobs are assumed to enter the system and to join a queue for service. After moving to the head of the queue, they are processed and either leave the system or join the queue for another service. Some simple queueing systems may be analyzed as Markov processes. For example, in a single-queue, single-service system, the system states are defined by a single variable - the number of jobs in the queue. The transition matrix then tends to be very simple and solvable by analytic means.

Queueing theory is a natural way to determine buffer needs at the various points in the front end. Consider, for example, the CPM buffers. Assume for simplicity that there is no flow control. Suppose that we can measure or estimate the rate of arrival of messages to be processed and the time it takes the CPM to process each message. Then queueing theory will immediately yield the probabilities p_n that there are n messages waiting to be processed. We would then choose the number N of buffers to be large enough so that

$$\sum_{n>N} p_n$$

is very small (so that the buffers seldom overflow). Flow control will, of course, prevent buffer overflow. But the queueing theory analysis is still useful, since if message arrivals must be slowed because of a shortage of buffers, system throughput will decrease.

It is likely that the more complex aspects of the front end will not be amenable to queueing theory analysis. However, it is worth noting that queueing theory has been successfully used to develop a better understanding of many features of computer system behavior [3,4]. Furthermore, Scherr's classic response analysis [5] of time-sharing systems used only elementary queueing theory and arrived at results closely agreeing with experiment. It may be that the best way to determine the maximum number of terminals that can be handled by the front end is to combine a theoretical response study with experimental determination of key parameters. That is, attaching one or two terminals to the front end can provide the basic data on the behavior of the front end as a terminal handler. Then, instead of adding more and more terminals (or artifically exercising the system as discussed above), an extrapolation can be carried out using theoretical results to estimate

system degradation with increasing number of terminals. This approach could save large amounts of experimentation time and expense.

As with other theoretical tools, queueing theory might best be used interactively with experiments to fine-tune the system design. Suppose we wish to improve the terminal-handling capability of the front end. We might begin by studying (as proposed in the preceding paragraph) the capability of the existing design. We can then experimentally check the validity of the theoretical extrapolation by actually increasing the number of terminals. If the extrapolation appears to be valid, we can then use it to determine efficiently which of a set of design changes is most likely to improve front-end performance. A thorough check can then be made on just this one design.

Appendix 2

Dynamic Statistical Data Collection Techniques

Statistics, such as mean buffer utilization and mean time for a message to traverse a given path, will need to be computed in the course of experimentation. The obvious way to do this is to collect the raw data, tabulate it, and process it - computing the desired statistics - at specified intervals or even after an experiment has been run. The problem with this approach is that it is extremely inefficient. Large amounts of storage are required for the tabulated data. We cannot afford to use more than a minimal amount of front-end storage for experimental data. Using more will have a deleterious effect on our efforts to study the storage needs of a front end. The data could, of course, be read out to disk. But frequent disk reads would also have an impact on measured performance.

An alternative approach to collecting statistics is to construct running averages (or other statistics) as the observations are made. This approach eliminates the need for storing old observational data. Denning and Eisenstein [1] discuss such schemes in the context of computer performance evaluation. In detail, the simplest scheme goes as follows.

Let x_1, x_2, ... be a sequence of observations of some quantity of interest. (The x_i's might, for example, be observed values of the number of events in the IPC queue of the host-host service module.) We wish to compute estimates of the mean value $\overline{x}$ of the sequence. A set of such estimates can be computed recursively from the equations

$$(1) \qquad \begin{aligned} \hat{x}_o &= 0 \\ \hat{x}_k &= \hat{x}_{k-1} + \frac{1}{k}(x_k - \hat{x}_{k-1}). \end{aligned}$$

Equations (1) are readily seen to be algebracially equivalent to computing the mean value in the usual way from the stored sequence. That is,

$$\hat{x}_k = \frac{1}{k}(x_1 + x_2 + \ldots + x_k).$$

The difference is that to compute $\hat{x}_k$ from (1) requires only the new observation x_k and the preceding estimate $\hat{x}_{k-1}$. Furthermore, a procedure to compute estimates from (1) is trivial to write. This simplicity also makes the overhead of computing estimates of $\bar{x}$ from (1) minimal.

The statistical data collection procedure described above has one disadvantage - it _closes down_. That is, successive observations are added in with steadily decreasing weights, so that they have less and less effect on the computed mean. This doesn't matter if we are examining a system in its steady state, so that the observed mean should remain constant in time. But if the system is just starting up, or if it takes a long time to reach a steady state, we must concern ourselves with the close-down problem.

One way to circumvent close-down is to reinitialize the data collection process occasionally. For example, at regular intervals the mean computed over the preceding time interval might be read out and the recursive computation of (1) restarted with $\hat{x}_o = 0$. This restart process necessarily introduces a statistical bias. However, the process is asymptotically unbiased, in the sense that the effect of this bias damps out as the number of observations increases. Even so, reinitialization must be used with care and not carried out too frequently.

Denning and Eisenstein discuss several so-called <u>responsive estimators</u>, or unbiased statistical estimators which shed the effects of early observations and do not close down. The simplest of these is the <u>moving-window estimator</u>, in which running averages are computed over the preceding T observations. The parameter T is chosen to be large enough to get good statistics, but small enough so that the average is sensitive to expected changes. Unfortunately, the method requires storage of the last T observations, and is therefore probably not a practical alternative.

A type of responsive estimator known as an <u>exponential estimator</u> is, however, worth consideration. The idea is the following. The estimation scheme defined by (1) can be generalized to

$$\hat{x}_1 = x_1,$$
$$\hat{x}_k = \hat{x}_{k-1} + a_k(x_k-\hat{x}_{k-1}),\ k=2,3,\ldots .$$

This estimator is valid for many choices of constants $a_1, a_2, \ldots$. An exponential estimator is one for which $a_k = a$, where $0<a<1$, for all k. The exponential dropoff in the weighting of old observations is readily seen by writing out $\hat{x}_k$:

$$\hat{x}_k = ax_k + (1-a)ax_{k-1} + (1-a)^2ax_{k-2} + \ldots$$
$$= x_1(1-a)^{k-1} + a \sum_{i=o}^{k-2} (1-a)^i x_{k-i} \ .$$

Thus, by using an exponential estimator, we can avoid the close-down problem without introducing the bias (and computational overhead) of periodic reinitialization.

Choosing a is clearly a nontrivial problem. Too small an a will prevent the estimate from responding well to changes. Too large an a will cause the estimate to change rapidly and erratically in response

to minor fluctuations. Denning and Eisenstein discuss the tradeoffs involved in choosing a in some detail.

There exist, of course, other more complicated estimators of the mean. For the purposes of front-end experimentation, however, the statistical software should be kept as simple as possible. We therefore propose to use the equations (1), probably with a provision for reinitialization to avoid problems from transient behavior when the system is started up. If a truly responsive estimator seems to be needed for collecting certain data, we can, at very little cost, go over to an exponential estimator.

It is not enough just to determine the average values of important system measurements. We also need to compute variances. To study buffer utilization, for example, we need to determine not only that process P has on the average 10 full message buffers, but we need to determine how large a variation from 10 we may expect. This is critical for the determination of the number of buffers required to avoid frequent buffer overflow.

As is well known [1], an unbiased estimator for the variance of a sequence of observations, given that the mean is also estimated, is

$$\hat{\sigma}_k^2 = \frac{1}{k-1} \sum_{i=1}^{k} (x_i^2 - \hat{x}_k^2)$$

As with the mean, it is not necessary to keep a record of all the sample values. Instead we keep the running sum of the squares of the sample values, $s_k = \Sigma_1^k x_i^2$:

$$(2) \qquad \begin{aligned} s_o &= 0 \\ s_k &= s_{k-1} + x_k^2 \end{aligned}$$

At any time we may compute $\hat{\sigma}_k^2$ (for $k \geq 2$) by:

$$(3) \qquad \hat{\sigma}_k^2 = \frac{1}{k-1}(s_k - k\hat{x}_k^2).$$

In summary, we propose to compute measurement statistics dynamically. After every observation, equation (1) will be used to calculate the mean value of the sequence of measurements up to then. Equation (2) will be used to compute s_k. Sample variances can be computed at the end of the experiment using (3). Occasional reinitialization may be used to avoid problems from the close-down effect and from start-up transients. Although, as discussed above, more sophisticated schemes are available, we believe that this simple, low-overhead approach will be best for the front-end experimentation.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CAC Document Number 227 CCTC-WAD Document Number 7509	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (*and Subtitle*) Network Research in Front Ending and Intelligent Terminals - Experimental Network Front End Experiment Plan		5. TYPE OF REPORT & PERIOD COVERED Research
		6. PERFORMING ORG. REPORT NUMBER CAC #227
7. AUTHOR(*s*) Geneva G. Belford Daniel E. Putnam		8. CONTRACT OR GRANT NUMBER(*s*) DCA100-76-C-0088
9. PERFORMING ORGANIZATION NAME AND ADDRESS Center for Advanced Computation University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Command and Control Technical Center WWMCCS ADP Directorate, 11440 Isaac Newton Sq., N. Reston, Virginia 22090		12. REPORT DATE May 16, 1977
		13. NUMBER OF PAGES 44
14. MONITORING AGENCY NAME & ADDRESS(*if different from Controlling Office*)		15. SECURITY CLASS. (*of this report*) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (*of this Report*)

Copies may be requested from the address given in (11) above.

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)

No restriction on distribution.

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (*Continue on reverse side if necessary and identify by block number*)

Network front end
Computer system measurement

20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

The CAC is engaged in an investigation of the benefits to be gained by employing a network front end. A DEC PDP-11/70 is being used as front end for connecting a Honeywell 6000 host to the ARPANET. This document presents a plan for testing of and experimentation with the experimental front end currently under construction.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)