THE BBN MULTIPROCESSOR

S.M. Ornstein, W.B. Barker, R.D. Bressler,
W.R. Crowther, F.E. Heart, M.F. Kraley,
A. Michel, M.J. Thrope

Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

This paper appeared in the Computer Nets Supplement to the Proceedings of the Seventh Hawaii International Conference on System Sciences, January 1974, and is reproduced with the permission of the publisher, Western Periodicals Company, California.

THE BBN MULTIPROCESSOR*

S.M. Ornstein, W.B. Barker, R.D. Bressler, W.R. Crowther
F.E. Heart, M.F. Kraley, A. Michel, M.J. Thrope
Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

Abstract

The BBN multiprocessor has gone from conception to prototype over the past year. It is highly modular at several logical and physical levels and will soon be a new IMP in the ARPA Network. It is very flexible both in the range of bandwidths it can handle and the number and type of interfaces it can accommodate.

1. INTRODUCTION

Last year we presented a paper which described the multi-processor we were then setting out to build as a new IMP for the ARPANET [1,2]. Much has been accomplished in this past year and we report here on progress made as well as on some important features of the system that have evolved. Familiarity with the earlier paper is assumed in what follows.

The architecture, as previously described, is highly modular and allows for IMPs of greater or lesser processing power than the present 516/316-based IMPs, as well as for many more and more varied phone line and Host interfaces. The hardware consists of busses joined together by special bus couplers of our design. There are processor busses each of which contains two processors, each in turn with its own "private" 4K memory to store frequently run code. The more processor busses, the greater the system processing power. There are memory busses to house the segments of multiported "common" memory — the more memory busses, the more memory ports. Finally, there are I/O busses which house device and line controllers as well as a special (priority ordered) task disburser which replaces the traditional priority interrupt system. The latter allows equality among the processors so that if some fail the rest can continue to run all system tasks, albeit at reduced capacity.

2. DESIGN ISSUES

In this section we describe features we have designed into the system, some of the more interesting of which relate to reliability issues.

2.1 ADDRESSING & LOCKING

The Lockheed SUE, with a 15-bit word address, can address up to 32K words. A 1.5-megabit line running over a 1/2 sec. round trip satellite channel holds 750,000 bits or about 50,000 words, copies of which must be held in the IMP for possible retransmission. Address expansion is thus inescapable and to allow for several such lines and be reasonably unbound by address space, we have allowed for half a million words. The bus coupler serves as the vehicle for address expansion. 8K of a processor's address space are used for direct references to its private memory. (Although we expect to use only 4K, 8K has been set aside to allow for growth.) Another 8K is used principally for addressing system I/O (on the up to four I/O busses). We assign 8 addresses to each I/O device for pointers and status and control registers; 960 devices can be accommodated in all.

16K of each processor's address space is mapped through the couplers to common memory. At the processor end of each coupler are four program-settable map registers for each possible processor on the bus. (We

*This work was supported by the Advanced Research Projects Agency under Contracts DAHC15-69-C-0179 and F08606-73-C-0027.

expect to use only two processors per bus but up to four are allowed for.) These map registers expand a 15-bit address to a 19-bit system address on the memory busses. By use of the maps, each processor can thus access, at any one time, four 4K pages in system address space. Read accesses through a particular one of these windows are turned by the coupler into read-clear operations, thereby providing the indivisible test-and-modify operation required for program interlocking in a multi-processor. (The processor itself presently lacks such an instruction.)

2.2 ACCESS ENABLING

The coupler paths that connect processor busses into memory and I/O busses have program settable enabling switches at their far (memory and I/O) ends, thus permitting processors to be cut in and out of the system. To allow processors to access one another and to permit reloading as discussed below, we have provided reverse paths in the processor to I/O couplers which also have enabling switches. Normally the forward paths to memory and I/O are turned on and the backward paths are shut off. Since these paths represent a hazard whereby a "sick" processor or device could damage healthy processors, we have arranged that only by storing a password at the proper address can a switch be changed. This greatly reduces the probability that a berserk processor painting memory will affect the path. A processor can neither enable nor disable its own access paths but one processor, deciding that another is sick and should be eliminated from the system, can amputate the bus of the offending processor. It can be similarly reinstated later.

The logic upon which amputation decisions are based is not yet fully understood and will be worked out as experience grows. We expect to require all processors to execute periodic healthiness-proving tasks. A regular system task, performed by any free processor, verifies that all processors have passed their tests and amputates any unhealthy one(s). Protective embellishments easily suggest themselves and we expect to do what seems necessary.

2.3 DISCOVERY

The operational program implements the IMP algorithm with whatever hardware is working at a particular site at a given time. The program discovers the hardware configuration as follows: Memory is found by trying to access it; a failure interrupt results if Memory is not there. Processors are found by accessing a register whose response indicates if the processor is absent, running or halted. I/O Devices are found by reading the 1st word of every possible device in I/O space — a failure interrupt means no Device, a response returns a unique 16-bit device type. Any parameters needed to run the devices are available as status words in the 8-word block. It is somewhat harder to find where the bus boundaries are, but they too can be found by searching for the bus coupler disable switches. In the event that there is some property we cannot otherwise discover, we have set aside 3 registers (associated with the clock device) to hold this information. For example, the IMP number (used for network routing) is contained in 8 bits of these registers.

The Discovery logic is not an initialization phase; rather the program periodically runs through the Discovery logic and reconfigures whenever a change occurs. It thus automatically adapts the IMP algorithm not only to the wide variety of possible configurations but also to those which contain broken components.

2.4 PARITY

At present the memories we are using do not store parity; however, we have built into our system design (and into the hardware) mechanisms to incorporate parity. These mechanisms have been tested with prototype parity memory and we have recently ordered parity memories for our production machines. We use a novel parity computation based not only upon the contents of a word but also on its address. The scheme also detects both "all ones" and "all zeros" failures. For writes to common memory, parity is computed at the processor end and fed, via the coupler, to the memory where it is stored with the word. Reads from memory fetch this stored parity, which is compared to a recomputed parity at the processor end of the coupler, thus checking both the memory and coupler paths in both directions. For units on the I/O bus, in order to check the coupler paths, a special card computes and transmits parity for all words being read from the I/O bus by the processors and checks parity on all words arriving from processor busses.

2.5 RELOADING

At present we use paper tape to load the system. The operator starts a processor which, from tape, loads its own private memory, its map registers and thereby any or all of common memory. It also loads, using backward coupling, the private memories on all other processor busses in the system. After the memory has been loaded, a startup procedure is executed which finally turns on the other processors.

Since all crucial switches, parameters, registers and control flip flops have been made addressable by reads and writes, load-

ing the system and starting it up can be done by externally force feeding it with the right set of addresses and data. Although we presently use paper tape in conjunction with a bootstrap ROM executed by a processor for this purpose, we are planning to construct a means whereby the system can be force fed directly from the network. The mechanism for this is a device on the I/O bus which monitors phone lines from adjacent IMPs looking for a special format which signals arrival of reload information. The card then performs the reload by executing store type bus cycles using the reload data.

This sort of operation, which looks forward to elimination of paper tape, switches, and other operator dependent functions, is appropriate to the IMP job. If a running system fails, as viewed from the net, the first step is to send it a regular "for IMP" message which causes a standard system restart to be attempted. If that seems not to work, the next step is to send another regular message trying to activate the reload-from-the-net code in hopes that it is still intact. Only if that fails would one attempt to force a full restart from scratch, in which case the special card described above is called into play. The first data sent halts the processors in order to stop any interfering activity. Then the reload-from-the-net code is refreshed and finally a processor restarted running that code which then completes reload via the normal packet mechanism.

2.6 MECHANICAL MODULARITY

We have settled on a modular mechanical structure well matched to the modular logical structure of the system. This structure is important in that it allows easy construction of systems of varied size and permits repair of parts of a system while the rest of it continues to operate. The basic unit is a cooling module which houses either 1) a 16-slot bus complete with its own power supply, 2) a 24-slot bus without power, or 3) a power supply for such a 24-slot bus. These units, each with its own set of fans, sit on rails in a vertical tier in a rack, five of them filling a standard height rack. (The 14-processor system requires three racks.) Figure 1 shows how the cooling modules stack. Air flow is from back to front so that racks placed beside one another do not directly heat each other. A tilted pan at the bottom of each module separates the air flow between stacked modules, thus eliminating chimney effects. Cards plug in from the front and all device and coupler cables also connect on that side. An entire unit can be removed to the rear for repair or replacement of the bus, fans, etc. — all without disturbing operation of the remainder of the system.

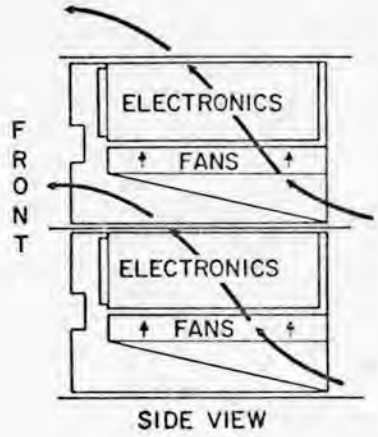


Figure 1
Mechanical Structure

3. THE TEST PROGRAM

The primary design objective of the test program is to exercise all of the hardware as intensively and extensively as possible, detecting all failures and reporting them precisely and comprehensibly. Extensive testing implies a wide variety of test modules; intensive testing implies permitting the entire computational power of the system to be focused on individual components at times. These objectives led to the selection of a system based on processes, analogous to a time-shared system's jobs. Processes are not tied to processors; a given process will switch rapidly from one processor to another. Nor is a process in general tied to a specific copy of code; like time-shared jobs, processes share a single copy of sections of pure procedure.

There are four types of processes: the "system" processes, including the clock, timeout, and type-out processes; the device-specific processes, which are tied to particular I/O devices, two processes per device; the "GART" (Get A Random Test) processes, which select a test at random from a table of tests to be performed; and a dummy process, whose sole purpose is to assure that there is always a runnable process.

Each GART test is designed to test a particular element or feature of the system. These range from standard processor and memory tests (the latter are also useful for checking bus couplers) to exercising the various bus coupler switches and maps. The I/O devices are kept busy by circulating various data through them.

4. WHERE WE STAND

Although the system uses Lockheed SUE processors, busses, memories, etc., we have so far designed and built nine BBN card types for the system: three coupler cards for each of the three bus types, a full-duplex memory channel card, a Host interface card

(which operates at speeds up to 1.5 megabit), transmit and receive modem cards, the pseudo-interrupt card and a clock card. These designs are virtually all finalized and many are in production (printed circuit or similar) form.

We are presently finishing the design of two other cards: the first of these is the parity checking card for the I/O bus described above under the discussion of parity. The second is a checksum/block-transfer card which flows a block of memory through itself computing a checksum as it goes. This is used to checksum critical code from time to time [3], to compute checksums for network end-to-end checking of messages, and other useful checking purposes. A transfer mode can be enabled so that it can also be used to move blocks of information about in memory (checksumming as it goes if desired). In addition we are presently embarking on modifications to the modem transmit and receive cards which will allow them to deal with 1.5 megabit lines and design of the special interface which monitors incoming inter-IMP lines watching for reload information as described above.

At present we are running several systems. Two small systems are being used for testing and debugging of the IMP program. These are sometimes run as separate single bus IMP systems which are connected together with our prototype 516 IMP into a three-node network. At other times the two busses are combined into a single system using a bus coupler. In this case one bus is used as a dual processor bus and the other as a combined memory and I/O bus. This system then works with the 516 IMP to form a 2-node net.

The growing prototype 14-processor system presently consists of three dual processor busses, two memory busses and one I/O bus. We have grown up to this system gradually but it now operates with sufficient reliability under stress (shaking of cables, margining power supplies, shuffling of cards, etc.) that we are presently in the process of building toward the full prototype (i.e., adding the 2nd I/O bus and the remaining four processor busses). By mid-1974 we hope to have two production copies of this large prototype working in the network. During 1974 we plan also to design satellite modem interface cards and to produce and deliver three moderate sized systems with satellite capability [4].

The basic IMP system program is up and running in multi-processor form, that is, with processors picking tasks up via the pseudo-interrupt system and using locks to prevent interfering accesses to resources. So far it has been run only with a two-processor system, but it will shortly be put on the larger prototype. The inner parts of the system, store and forward, Host, task, etc., seem solid. The work that remains is in implementing the system maintenance, monitoring, and debugging functions (i.e., system DDT, periodic status reports, etc.). This coding is about half done and needs finishing as well as debugging. The network error recovery code is ready for debugging. The special reliability code which keeps the system up when parts of the hardware fail is being designed.

Much work must be done in the present network to accommodate the advent of the new line of machines. For example, the whole reloading mechanism must be changed since one's neighbor may now be very different from one's self. The network must therefore be able to pass core load images packet-by-packet to an immediate neighbor of the machine needing reloading.

Our small IMP is built on a single logical bus (consisting of two separate physical busses connected by an extender) which combines memory, processor and I/O. This system embodies none of the special reliability stemming from multiple hardware copies but is the least expensive version available. Small reliable systems are another matter and require, in general, doubling the system to provide complete redundancy of parts to allow for any single failure. Such systems may prove to be one of the more significant outgrowths of this development effort.

REFERENCES

1. Heart, F.E. et al, *The Interface Message Processor for the ARPA Computer Network,* Proceedings AFIPS 1970 SJCC.
2. Heart, F.E. et al, *A New Minicomputer/ Multiprocessor for the ARPA Network,* Proceedings AFIPS 1973 NCC.
3. Crowther, W.R. et al, *Reliability Issues in the ARPA Network,* ACM Data Communications Symposium, Nov. 1973.
4. Butterfield, S.C. et al, *The Satellite IMP for the ARPA Network,* Seventh Hawaii Int. Conf. on System Sciences, Jan. 1974.

THE EVOLUTION OF A HIGH PERFORMANCE MODULAR PACKET-SWITCH*

Severo M. Ornstein and David C. Walden
Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

ABSTRACT

The authors discuss the design of a series of packet-switching systems in terms of evolving design considerations, and resultant hardware and software structures, from the first such system to a recent, entirely new, packet-switch.

INTRODUCTION

Since 1968 we have been involved in the task of developing and refining a series of packet-switches. Our work has been directed primarily toward providing packet-switching systems to be used in the ARPA Network[1], but our systems have found some "outside" use in the past and we foresee even more widespread outside use in the future, both in commercial and in non-ARPA government applications.

The first packet-switch in our series, described in Reference 2, was based on the Honeywell 516 computer. When the decision to use the 516 was originally made in 1968, the range of machine choices was much smaller than it is today. The second packet-switch in our series was developed in 1971, as described in Reference 3; based on the Honeywell 316 (the successor to the 516), this second machine was developed primarily to provide a less expensive version of the original switch. The choice of the 316 as successor to the 516 permitted this second version to be developed with minimal reprogramming, although hardware "specials" in the system had to be reengineered.

Over the past six years, we have also had responsibility for delivering new systems to the ARPA Network and for maintaining and operating that network. This has given us the opportunity to continually study and refine the packet-switches, a process described, for instance, in References 4,5. Our experience has produced many insights into just what characteristics make a machine more or less suitable for use in a packet-switch. On the basis of this experience, a new machine called the Pluribus[6,7], has been designed and built with the specific requirements of a packet-switching node in mind from the outset.

In the present paper we first discuss a number of packet-switching network considerations and their influence on the design of the packet-switching computer itself. We then describe the hardware and software structures both of the 516/316 packet-switch and of the Pluribus-based switch, and show how each addresses these considerations.

NETWORK CONSIDERATIONS

Processor

The speed of the packet-switch's processor is an important determinant of the throughput rates possible in the network (the effect of processor speed on delay is of much less concern since processing delays are typically less than a millisecond). The store-and-forward processing bandwidth of the processor can be computed by counting instructions in the inner program loop. The source-to-destination processing bandwidth can be calculated in a similar fashion. These rates should be high enough so that the entire bandwidth of the network lines can be used, i.e., so that the node is not a bottleneck. Experience indicates that the instruction cycle time is the main factor in this bandwidth calculation; complex or specialized instruction sets would not be particularly valuable because simple instructions make up most inner loops--at least this is true in the systems we have built.

A different aspect of the packet-switch's processor which can also affect throughput is its responsiveness. Because circuits are synchronous devices, they require service with very tight time constraints. If the switch does not notice that input has completed on a given circuit, and does not prepare for a new input within a given time, the next input arriving on that circuit will be lost. Similarly, on output the switch must be responsive in order to keep the circuits fully loaded. This requirement suggests that some form of interrupt system or high-speed polling device is necessary to keep response latency low, and that the overhead of an operating system and task scheduler and dispatcher may be prohibitive. Finally, we note that the amount of time required by the switch to process input and output is most critical in determining the minimum packet size, since it is with packets of this size that the highest packet arrival and departure rates (and thus processing requirements) can be observed. Of course, data buffering in the device interfaces can partially alleviate these problems.

Memory

The speed of memory may be a major determinant of processor speed, thus affecting the switch bandwidth. An equally important consideration is memory speed for I/O transfers, since the switch's overall bandwidth results from a division of total memory bandwidth based on some processing time for a given amount of I/O time. First, there is

- - - - - - - - - -

*This work was supported under Advanced Research Projects Agency (ARPA) Contracts Number DAHC-15-69-C-0179 and F08606-73-C-0027.

the question of whether the I/O transfers act in a cycle-stealing fashion to slow the processor or whether memory is effectively multi-ported to allow concurrent use. Second, there is the issue of contention for memory among the various synchronous I/O devices. In a worst-case scenario, it is possible for all the I/O devices to request a memory transfer at the same instant, which keeps memory continuously busy for some time interval. A key design parameter is the ratio of this time to the available data buffering time of the least tolerant I/O device. This ratio should be less than one, and may therefore determine how much I/O can be connected to the node.

The size of the memory, naturally, is another key parameter. In our experience, with a terrestrial network the switch program and associated data structures take up the majority of the switch's storage; this may change with the introduction of high-speed satellite circuits. The remainder of the switch's memory is devoted to buffering of two kinds: packet buffering between adjacent nodes, and message buffering between source and destination nodes. These requirements can be calculated quite simply in each case as the product of the maximum data rate to be supported times the round trip time (for a returning acknowledgment). In large networks it may be necessary to rely on sophisticated compression techniques to ensure that tables for the routing algorithm, the source-to-destination transmission procedures, and so on, do not require excessive storage.

I/O

The speed of the I/O system has been touched upon above in relation to processor and memory bandwidth. Other factors worth noting are the internal constraints imposed by the I/O system itself--its delay and bandwidth. A different dimension, and one that we have found to be inadequately designed by most manufacturers, is the flexibility and extensibility of the I/O system. Most manufacturers supply only a limited range of I/O options (some of which may be too slow or too expensive to use). Further, only a limited number of each type can be connected. A packet-switch requires high performance from the I/O system, both in the number of connections and in their data rates.

General Architecture

There are other factors to consider in evaluating or designing a computer for the packet-switch function apart from performance in terms of bandwidth and delay. As we mentioned, extensibility in I/O is very important and was comparatively rare until recently; it is more common to find memory systems which can be expanded. Processor systems which can be expanded are not at all common, and yet processor bandwidth may be the limiting factor in some switch configurations. Without a modular approach allowing processing, memory, and I/O growth, the cost of the switch can be driven quite high by large step functions in component cost.

A final aspect of switch computer architecture is its reliability, particularly for large systems with many lines and Hosts. A failure of such a system has a large impact on network performance. Computer manufacturers tend to cut corners in order to compete on price and delivery schedules. The penalty for this practice is usually paid in the coin of lowered reliability. These issues of performance, cost, and reliability become critically important in large networks serving thousands of Hosts and terminals on a 24-hour-a-day basis.

THE 516/316 IMP

As stated above, we have previously built a pair of packet-switches based on first the Honeywell 516 and then the Honeywell 316 technology. We call our packet-switches Interface Message Processors (or IMPs), and the 516 and 316 based IMPs are described below.

Hardware

The 516 and 316 IMPs can be considered to be the same machine. The 316 derivative is less expensive, smaller, effectively 30% slower, and less tightly engineered than the original ruggedized 516s. Architecturally, however, the machines are very similar. They share the following characteristics: 1) simple processor with 16-bit word length, 512-word pages, single accumulator, approximately 1 microsecond cycle time, one index register, indefinite indirect addressing, power-fail, and auto-restart; 2) 16K words of memory which can be conveniently addressed; 3) a multiplexed 16-channel direct memory access unit for Input/Output; 4) a 16-level priority interrupt system; and 5) a Teletype for maintenance and local debugging.

To the basic machine are added a number of special hardware features which suit it to the IMP job, specifically:

Full-duplex, synchronous interfaces are provided for connecting the IMP to the communication lines. These interfaces are clocked by the modems at the bit level. They are treated by the program at the packet level (that is, they are direct access devices which send and receive blocks of words from storage, and use interrupts to notify the program of the completion of each packet transfer). The jobs of generating inter-packet (idle) "sync" characters, providing packet framing characters, fetching successive words of a packet from memory via a memory channel, serializing and deserializing all characters to and from the modem, formulating and checking a 24-bit cyclic redundancy check, detecting overflow and format errors, and signaling the completion interrupt are all handled by the interface.

Full-duplex, asynchronous interfaces connect the IMP to one or more Host computers. Connecting an IMP to a wide variety of different Hosts requires a hardware interface, some part of which must be custom tailored to each Host. A standard portion of the interface is built into the IMP, identical for all Hosts (it is a direct

memory channel device like the modem interface), while a special portion must be uniquely designed for each different Host type. A bit serial interface is used partly because it is less expensive and partly to accommodate conveniently the variety of word lengths in the different Host computers. The interface operates asynchronously, each data bit being passed across the interface via a Ready For Next Bit/There's Your Bit handshake procedure. This technique permits the bit rate to adjust to that of either member of the pair and allows for pauses when words must be stored into or retrieved from memory.

A relative time clock provides an interrupt every 25.6 microseconds for performing all time-dependent tasks such as timing out packets for retransmission. Programmed counters are run off this clock to perform still lower frequency periodic jobs.

A watchdog timer is provided which is normally held off (i.e., restarted) by the program. The timer is restarted every time a correct response to a (periodic) "trouble report" is returned from the net indicating that most of the basic system is operating properly. If the timer ever runs out, a failure of some sort is indicated and a reload-and-restart program is activated.

A program-generatable interrupt, known as the task interrupt, permits program-generated tasks to be queued and handled by the same priority-ordered, interrupt-driven program structure as hardware (I/O) generated tasks.

A program-readable hardware-held register identifies the number and configuration of the machine.

It is illuminating to consider the impact of some detailed differences between the 516 and 316 machines. First, although the 516 basic cycle is .96 microsecond and the 316 cycle is 1.6 microseconds, the real traffic-handling capabilities of the IMP are not in this simple ratio since other factors affect it. In the 516, I/O channel pointers are kept in memory, while in the 316 they are in hardware. The result is that each I/O word transferred in the 516 takes about 4 microseconds while in the 316 it takes 3.2 microseconds. Thus for the same amount of I/O traffic, the 516 has less real time left for program execution, but it does instruction faster. In fact, for the IMP task, although one must consider the memory access time used for I/O transfers, the processing bandwidth tends to predominate. The relative importance of processing time to I/O time, however, will vary for a number of reasons. For instance, packet processing is less costly per bit for longer packets than short, whereas I/O transfer time per bit does not change. Second, in the 516 the watchdog timer is a hardware timer; in the 316 the timer is a software counter run off the relative time clock interrupt routine. Neither timer can really verify that all features of the program are operating properly. The 316 timer is vulnerable to a break in a tiny piece of the clock code but, by using the relative time clock, it eliminates the need for a separate hardware timer. We make up for this apparent deficiency by checking in other places in the code to see if the clock code is running, and if it is not, a reload is initiated. Several other key data and control structures are tested in a similar fashion.

There are several other features which differ between the two machines but which have not turned out to be significant: 1) the 516 has a 512-word block of protected memory, originally built in for protecting recovery and startup programs, while the 316 does not have this feature; 2) the halt instruction in the 516 can be inhibited (turned into a NOP) so that interrupts will always be able to force control to interrupt serving routines while the 316 simply halts in all cases; and 3) the 516 has the option of halting or interrupting on power fail while the 316 is only able to interrupt.

Software

The operational 516/316 IMP program is composed of a number of functionally distinct pieces; each piece occupies no more than a few pages of core (a core page is 512 16-bit words). These programs communicate primarily through common registers that reside in page zero of the machine and are directly addressable from all pages of memory.

The programs' main data structures are tables and queues. The buffer storage space is partitioned into about 50 fixed-length buffers, each of which is used for storing a single packet. An unused buffer is chained onto a free buffer list and is removed from this list when it is needed to store an incoming packet. A packet, once stored in a buffer, is (almost) never moved. After a packet has been passed onto another IMP or a Host, its buffer is returned to the free list. The buffer space is partitioned in such a way that each process (store-and-forward, Host traffic, etc.) is always guaranteed some buffers. For the sake of program speed and simplicity, no attempt is made to retrieve the space wasted by partially filled buffers.

In handling store-and-forward traffic, all processing is on a per-packet basis. Further, although traffic to and from Hosts is composed of messages, the IMP rapidly converts to dealing with packets; the Host transmits a message as a single unit but the IMP takes it one buffer at a time. As each buffer is filled, the program selects another buffer for input until the entire message has been provided for. These successive buffers are, in general, scattered throughout the memory. An equivalent inverse process occurs on output to the Host at the destination IMP. No attempt is ever made to collect the packets of a message into a contiguous portion of the memory.

Buffers currently in use are either dedicated to an incoming or an outgoing packet, chained on a queue awaiting processing by the program, being processed, or awaiting acknowledgment or other response. Occasionally, a buffer may be simultaneously found on two queues.

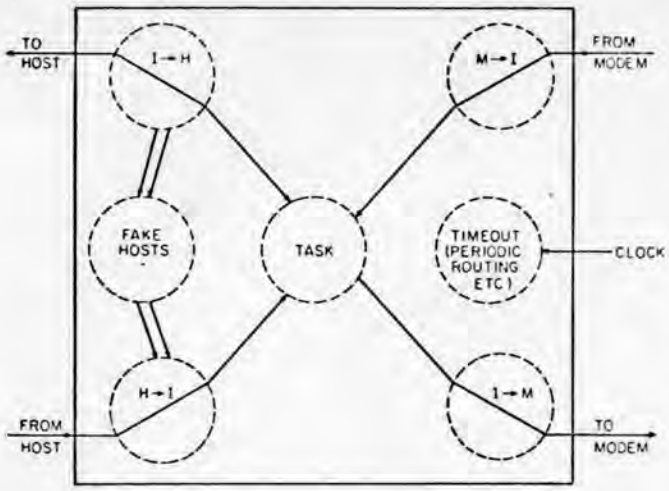


Figure 1

Figure 1 shows in schematic form the program modules most directly concerned with packet processing.

The Host to IMP (H-I) routine handles messages being transmitted from the IMP's own Hosts. The routine uses addressing information provided by the Host at the front of a message to construct a "header" that is prefixed to each packet of the message. It also segments the message into packets and passes each of the packets to the task routine and sets the task interrupt. The routine then acquires a free buffer and sets up a new input. The routine is serially reentrant and services all Hosts connected to the IMP.

The task routine uses header information to direct packets to their proper destination. The task routine is driven by the task interrupt, a program-settable interrupt, which is set by each program which puts a packet on the task queue. Packets for a local Host are passed to the IMP to Host routine. Packets for other destinations are placed on a modem output queue as specified by the routine table.

The IMP to modem (I-M) routine transmits successive packets from the modem output queues, and sends acknowledgments for packets received by the modem to IMP routine.

The modem to IMP (M-I) routine receives packets from the communication circuits and passes them to the task routine.

The IMP to Host (I-H) routine sets up successive outputs of the packets of a message found on the Host output queue. It also constructs end-to-end destination-to-source acknowledgments for messages passed to a Host and gives them to the task routine.

There are several other significant routines in addition to those just mentioned. The timeout routine is started every 25.6 milliseconds on a clock interrupt. The routine has two sections: the fast timeout and the slow timeout. The timeout routines do such things as waking of Host or modem routines which have asked to be awakened, garbage collection of tables, transmission and updating of the routing tables, marking lines alive or dead, and so forth.

The two timeout routines run off a common interrupt, although they are constructed to allow the faster routine to interrupt the slower routine should the slower routine not complete execution before the next timeout period.

As just mentioned, one of the tasks of the timeout routines is garbage collection. Every table, most queues, and many states of the program are timed out. Thus if an entry should remain in a table for an abnormally long time, or if a program (say IMP to Host) should remain in a particular state for an abnormally long time, this entry or state is garbage-collected and the table or program is returned to its initial or nominal state. In this way, abnormal conditions--which, despite all debugging and precautions, occasionally happen in all big systems (especially when independent computers such as a Host and IMP are communicating)--are not allowed to hang the system up indefinitely.

The way in which a table is frequently scanned from the timeout routines is interesting. For instance, suppose it is necessary to look at every entry in a 64-entry table every now and then. The program could be constructed to just wait for the proper interval and then look at every entry in the table on one pass through the timeout routine. However, this would cause an enormous transient in the average timing of the IMP program as a whole. So instead, the program is constructed to look at one entry each time through the timeout routine. This takes a little more total time but is much less disturbing to the program as a whole.

The program has an initialization section and a sizeable background loop. The background loop includes handling of the IMP's Teletype, a debugging program which resides in each IMP, the statistics programs, the trace program, and several other functions. The Network Control Center[8] and the Network Measurements Center[9] frequently find it useful to communicate with one of these background programs. So that these programs may send and receive messages from the network, they are treated as fake Hosts. The Host to IMP and IMP to Host routines both think they can handle eight Hosts although no more than four real Hosts are possible on an IMP. The other four Hosts are these background programs, which simulate the operation of the Host/IMP data channel hardware so that the Host/IMP routines are unaware that they are communicating within anything other than a real Host. This trick saved a large amount of code and provides a valuable isolation of program functions.

The size of the initialization code and the associated tables deserves mention. This was originally quite small. However, as the network has grown and the IMP's capabilities have been expanded, the amount of memory dedicated to initialization has steadily grown. This is mainly because the IMPs are no longer identical. An IMP may be required to handle a Very Distant Host, or TIP hardware, or five

lines and two Hosts, or four Hosts and three lines, or a very high speed line, or, in the near future, a satellite link. As the physical permutations of the IMP have continued to increase, we have clung to the idea that the program should be identical in all IMPs, allowing an IMP to reload its program from a neighboring IMP and providing other considerable advantages. However, maintaining only one version of the program means that the program must rebuild itself during initialization to the proper program to handle the particular physical configuration of the IMP. Furthermore, it must be able to turn itself back into its nominal form when it is reloaded into a neighbor. All of this takes tables and code. Unfortunately, we did not foresee the proliferation of IMP configurations which has taken place; therefore, we cannot conveniently compute the program differences from a simple configuration key. Instead, we must explicitly table the configuration irregularities.

It is characteristic of the IMP program that many of the main routines are entered both as subroutine calls from other programs and as interrupt calls from the hardware. The programs are arranged in a priority order; control passes upward in the priority chain whenever a hardware interrupt occurs or the current program decides that the time has come to run a higher priority program, and control passes downward only when higher priority programs are finished. No program may execute itself or a lower priority program; however, a program may freely execute a higher priority program. This rule is similar to the usual rules concerning priority interrupt routines.

In one important case, however, control must pass from a higher priority program to a lower priority program--namely, from the several input routines to the task routine. For this special case, we modified the computer hardware to include a low-priority hardware (task) interrupt that can be set by the program. When this interrupt is honored (that is, when all other interrupts have been serviced) the task routine is executed. Thus, control is directed where needed without violating the priority rules.

Some routines must occasionally wait for long intervals of time. Stopping the whole system would be intolerable; therefore, should the need arise, such a routine is dismissed, and the timeout routine will later transfer control back to the waiting routine. The control structure and the partition of responsibility among various programs achieve the following timing goals: 1) no program stops or delays the system while waiting for an event; 2) the program gracefully adjusts to the situation in which the machine becomes compute-bound; 3) the modem-to-IMP routine can deliver its current packet to the task routine before the next packet arrives and can always prepare for successive packet inputs on each line--this timing is critical because a slight delay here might require retransmission of the entire packet; 4) the program will almost always deliver packets waiting to be sent as fast as they can be accepted by the phone line; and 5) necessary periodic processes (in the timeout routine) are always permitted to run, and do not interfere with input-output processes.

THE PLURIBUS IMP

In its original form, the IMP was based as much as possible on the most suitable off-the-shelf hardware then available: special hardware design was kept to a minimum and consisted of interfaces and a few special features which were added to a standard machine. During the first few years of the network's existence, new and more flexible computer structures began to appear on the market, and the special requirements of packet-switching began to be better understood. The Pluribus architecture, developed specifically to suit the needs of a packet-switch,* is the outgrowth of these changes. There are two primary goals for the machine, representing areas in which the earlier IMPs were felt to be lacking. These were flexibility (i.e., the ability to expand or contract smoothly over wide ranges) and reliability. Originally there was a more primitive goal of higher throughput, but this was soon seen to be balanced by the need for a cheaper, smaller machine with less throughput. Bandwidth is thus seen as one domain in which greater flexibility is desired.

Let us consider the issue of flexibility in a little more depth. In most machines certain hardware "utilities" are shared among the various logical units. These include rack space, power, cooling, etc. Generally these utilities come in fairly large chunks, with correspondingly large steps in cost. Thus, one can typically add, say, interfaces up to a certain point; at which time a new rack, power supply, etc., must be added to permit further expansion. Even then, one may run out of logical channels or come up against other hard boundaries. The 516/316, for example, has a fixed memory channel arrangement which limits connection to a total of at most seven high-speed circuits and/or Host computers. The specific component which is totally inflexible in most systems is the processor; that is, there is typically no processor modularity or possible variation of processor capacity. The flexibility goal of the Pluribus was to smooth large step functions in cost by utilizing a highly modular design and to push really hard boundaries (such as absolute limits on memory addressing capabilities or processing capacity) well beyond requirements anticipated at least for the next few years.

The machine design was thus to allow for large numbers of I/O units, for wide ranges in processor power (up to at least an order of magnitude in traffic bandwidth handling capability improvement over the 316), and for larger possible memory (to permit longer and/or faster links, say, via satellite).

- - - - - - - - - - -

*The Pluribus architecture is also seen to be suitable for many applications other than that of a packet-switch, but these other applications will not be described in the present paper.

Now consider reliability. If a single IMP fails on an average of once a month (quite a high MTBF for conventional computers), then in a network of thirty such IMPs, one will fail on an average of once a day. In a network consisting of large numbers of computers (e.g., hundreds), issues of reliability take on paramount importance. In a multiply connected network the failure of a single IMP should be felt only locally. Economics, however, limits the number of paths between any given pair of IMPs. In the ARPA Network many IMP pairs are connected by only two disjoint paths, some by only one, and this tends to increase reliance on individual IMPs and to make their reliability a key issue.

If a single computer fails on an average of ten times a year, then a collection of ten computers, treated as a unit, will fail on an average of 100 times a year. However, suppose that rather than viewing the ten computers as a unit which is down if any one of its constituent computers is down, we view the ten computers as a unit which is up as long as any of its constituent computers is up. Further, suppose the mean time to repair a failed computer is small compared to the time between failures. In that case the probability that the constituent computers will all be down is very small, so it is unlikely that the unit as a whole will be down. The reliability of the Pluribus IMP takes advantage of such probabilities. Note that a key assumption is that individual failures are independent of one another. Although it is impossible to guarantee this independence (flood, total power failure, sabotage, etc.), nonetheless, in the Pluribus design, considerable attention has been given to maintaining as much isolation as practical so that one failure does not induce another.

With the goals of flexibility and reliability in mind and with the price and size of minicomputers dropping, it was decided that the Pluribus should be built along the lines of a minicomputer-multiprocessor, or more generally, a multi-resource (processors, memories, I/O channels, etc.) system.

In considering which minicomputer might be most easily adaptable to a multi-resource structure, the internal communication between the processor and its memory was of primary concern. Several years ago machines were introduced which combined memory and I/O busses into a single bus. As part of this step, registers within the devices (pointers, status and control registers, and the like) were made to look like memory cells so that they and the memory could be referenced in a homogeneous manner. This structure forms a very clean and attractive architecture in which any unit can bid to become master of the bus in order to communicate with any other desired unit. One of the important features of this structure is that it made memory access ing "public"; the interface to the memory had to become asynchronous, cleanly isolable electrically and mechanically, and well documented and stable. A characteristic of this architecture is that all references between users are time-multiplexed onto a single bus. Conflicts for bus usage therefore establish an ultimate upper bound on overall performance, and attempts to speed up the bus eventually run into serious problems in arbitration.

In 1972 a new computer was introduced--the Lockheed SUE--which follows the single bus philosophy but carries it an important step further by removing the bus arbitration logic to a module separate from the processor. This step permits one to consider configurations embodying multiple processors, as well as multiple memories and I/O, on a single bus. It also permits busses which do not include any processor at all. The processor used in the SUE computer is a compact, relatively inexpensive (approximately $600 in quantity), quite slow processor with a microcoded inner structure. Table I shows some of its characteristics. Its slowness and cheapness, of course, go together and since in a modular multi-processor, increased bandwidth is achieved merely by adding more processors, the weak/cheap processor has the advantage of allowing smaller steps to be taken along the cost/performance curve.

TABLE I - SUE COMPUTER CHARACTERISTICS

16-bit word
8 General Registers
3.7 microseconds add or load time
Microcoded
Two words/instruction typical
8-1/2" x 19" x 18" chassis
64K bytes addressable by a single instruction
200 ns minimum bus cycle time
850 ns memory cycle time
425 ns memory access time

Several components of the SUE computer were adopted for the Pluribus system, in particular the bus, the processor, and the bus arbitration logic.

Hardware Structure of the Pluribus IMP

Reliability was a main concern in planning the hardware architecture. The goal was to provide hardware which could be exploited by the program to survive the failure of any individual component.

The hardware consists of asynchronously and independently functioning communication busses, coupled together. From a physical point of view, the SUE chassis represents the basic construction unit; it incorporates a printed circuit back plane which forms the bus into which 24 cards may be plugged. From a logical point of view this chassis includes a bus which provides a common connection among all units plugged into the chassis. All specially designed cards as well as all Lockheed-provided modules plug into these bus chassis. With this hardware, the terms "bus"

and "chassis" are used somewhat interchangeably, but we will commonly call this standard building unit a "bus." Each bus requires one card which performs arbitration. A bus can be logically extended (via a bus extender unit) to a second bus if additional card space is required; in such a case, a single bus arbiter controls access to the entire extended bus.

One can build a small multiprocessor just by plugging several processors and memories (and I/O) into a single bus. For larger systems one quickly exceeds the bandwidth capability of a single bus and is forced to multi-bus architecture. Please refer to Figure 2 for the following discussions.

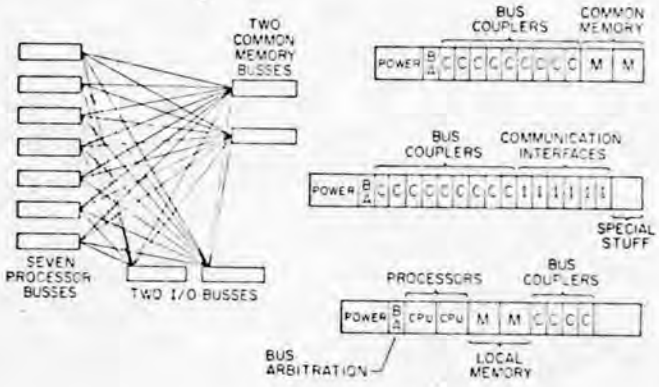


Figure 2

The functional units of the system (processors, memories, I/O controllers, and special devices) are distributed on these busses in such a way that units which must communicate frequently are placed on the same bus, whereas units which communicate less frequently will in general be on different busses. Units on the same bus can thus communicate at high speed without disturbing the remainder of the system. When a unit on one bus must communicate with a unit on another bus, some interference occurs while both busses are momentarily involved in the interaction. Each bus, together with its own power supply and cooling, is mounted in its own modular unit, permitting flexible variation in the size and structure of systems. There are processor busses each of which contains two processors, each in turn with its own local 4K memory which stores frequently run and recovery-related code. There are memory busses to house the segments of a large memory common to all the processors. Finally, there are I/O busses which house device controllers as well as certain central resources such as system clocks and special (priority-ordered) task disbursers which replace the traditional priority interrupt system.

The design is highly modular and permits systems of widely varying size and performance. In the interests of clarity, however, we will describe that design in terms of a particular system containing 14 processors--a system designed to have ten times the traffic-handling capability of the 316 IMP.

Resources. A central notion in a parallel system is the idea of a "resource", which we define to mean a part of the system needed by more than one of the parallel users and therefore a possible source of contention. The three basic hardware resources are the memories, the I/O, and the processors. It is useful to consider the memories, furthermore, as a collection of resources of quite different character: a program, queues and variables of a global nature, local variables, and large areas of buffer storage. The multiprocessor system is therefore in reality a multi-resource system, as mentioned above. The basic idea of the system is to provide multiple copies of the vital resources in the hope that the algorithm can run faster by using them in parallel and can survive failures of copies of any of the resources. The number of copies of the resource which are required to allow concurrent operation is determined by the speed of the resource and the frequency with which it is used. An additional advantage of multiple copies is reliability: if a system contains a few spare copies of all resources, it can continue to operate when one copy breaks.

It may seem peculiar to think of a processor as a resource, but in fact in a Pluribus IMP system the parallel parts of the algorithm compete with each other for a processor on which to run. Indeed, a novel feature of the Pluribus IMP design is the consistent treatment of all processors as equal units, both in the hardware and in the software. There is no specialization of processors for particular system functions, and no assignment of priority among the processors, such as designating one as master. Not only the IMP application job but also the multiprocessor control and reliability jobs are distributed among the processors so that all jobs are uniformly treated. The processors are viewed as a resource used to advance the algorithm; the identity of the processor performing a particular task is of no importance. Programs are written as for a single processor except that the algorithm includes interlocks necessary to insure multiprocessor sequentiality when required. The software thus consists of a single conventional program run by all processors. Each processor has its own local copy of about one quarter of this program and the remaining three quarters is in commonly accessible memory.

Processor busses. The bus used in the Pluribus can support up to four processors. As more are added, contention for the bus increases, and the performance increment per processor drops. The Pluribus IMP uses two processors per bus, which loses almost nothing in processor performance. When a processor makes access to shared memory via the switching arrangement, that access incurs delays due to contention and delays introduced by the intervening switch. In the IMP application, some parts of the program are run very frequently and other parts are run far less frequently. This allows a significant advantage to be gained by the use of private memory. A 4K local memory containing an individual copy of the frequently run code is associated with each processor on its bus. This allows faster access to this "hot" code; the local memories all contain the same code.

In the IMP application, the ratio of accesses to local versus shared memory is better than three to one.

Shared memory busses.* The shared memory contains program, message buffers, global variables, etc. Buffer requirements, of course, vary depending on the site. In a regular IMP, 40K words of common memory are necessary. Four memory units operating concurrently are required in order to hold processor contention to an acceptable level in the 14-processor system. Since the bus is considerably faster than the memories, two logical memory units may be placed on each shared memory bus with almost no interference. Two such memory busses are thus required.

I/O busses.* The I/O system consists of more standard busses. In the case of the 14-processor system, two such busses allow for the necessary bandwidth and provide redundancy for reliability purposes. Into these busses are plugged cards for each of the various types of I/O interfaces that are required, including interfaces for modems, terminals, Host computers, etc., as well as interfaces for standard peripherals. The I/O bus also houses a number of special units including (1) a clock which serves roughly the same functions as the clock in the 516/316 IMP; (2) a checksum/block transfer card which flows a block of memory through itself computing a checksum as it goes (used to checksum critical code, to compute end-to-end-checksums, etc.); (3) a special hardware task disbursing unit known as a Pseudo-Interrupt Device (PID) discussed further below; and (4) a "reload" card which monitors up to eight communication lines, watching for specially formatted reload messages from the network and processing them should any arrive.

Interconnection system. To adhere to the requirement that all processors must be equal and able to perform any system task, busses must be connected so that all processors can access all shared memory, so that I/O can be fed to and from shared memory, and so that any of the processors may control the operation and sense the status of any I/O unit.

A distributed inter-communication scheme was chosen in the interest of expandability, reliability, and design simplicity. The kernel of this scheme is called a Bus Coupler, and consists of two cards and an interconnecting cable. In making connections between processors and shared memory, one card plugs into a shared memory bus, the other into a processor bus. Similar connections are made for every processor bus to every shared memory bus. When the processor requests a cycle within the address range which the Bus Coupler recognizes, a request is sent down the cable to the memory end, which then starts contending for the shared memory bus. When selected, it requests the desired cycle of the shared memory. The memory returns the desired information to the Bus Coupler, which then provides it to the requesting processor, which, except for an additional delay, does not know that the memory was not on its own bus. The Bus Coupler also does address mapping. Since a processor can address only 64K bytes (16-bit address), and since we wished to permit multiprocessor configurations with up to 1024K bytes (20-bit address) of shared memory, a mechanism for address expansion is required. The Bus Coupler provides four independent 8K-byte windows into shared memory. The processor can load registers in the Bus Coupler which provide the high-order bits of the shared memory address for each of the four windows.

Given a Bus Coupler connecting each processor bus to each shared-memory bus, all processors can access all shared memory. I/O devices which do direct memory transfers must also access these shared memories. These I/O devices are plugged into as many I/O busses as are required to handle the bandwidth involved, and bus couplers then connect each I/O bus to each memory bus. Similarly, I/O devices also need to respond to processor requests for action or information; in this regard, the I/O devices act like memories and Bus Couplers are again used to connect each processor bus to each I/O bus. The path between processor busses and I/O busses is also used to allow processors to examine and control other processors for startup and trouble situations.

Software Structure of the Pluribus IMP

The problem of building a packet-switching store-and-forward communications processor lends itself especially well to parallel solution since packets of data can be treated independently of one another. Other functions of the IMP program such as general housekeeping, routine computations, reliability tasks, etc., can also be easily performed in parallel. The structure chosen works as follows: first, the program is divided into small pieces, called strips, each of which handles a particular aspect of the job. For example, one strip handles special routing messages from neighboring IMPs, another handles input from a local Host, and others handle further I/O and housekeeping functions. When a particular task needs to be performed, for instance upon receipt of a message over a communications circuit, the name (number) of the appropriate strip is put on a queue of tasks to be run. Each processor, when it is not running a strip, repeatedly checks this queue. When a strip number appears on the queue, the next available processor will take it off the queue and execute the corresponding strip. The program is broken into strips in such a way that a minimum of context saving is necessary.

Strips have different levels of importance. Data coming in over a high-speed communication circuit must be serviced more rapidly than data coming in over a Teletype-speed line. The number assigned to each strip reflects the priority of the task it performs. When a processor checks the task assignment queue, it takes the highest priority job then available. Since all processors access this queue frequently, the contention

- - - - - - - - - -

*The terms "memory bus" and "I/O bus" as used here and henceforth are not the same as conventional memory and I/O busses.

for it is very high. For efficiency, therefore, a special hardware device, the Pseudo Interrupt Device, was designed to serve as a task queue. A single instruction allows the highest priority task to be fetched and removed from the queue. Another instruction allows a new task to be put onto the queue. All contention is arbitrated by standard bus logic hardware.

The length of strips is governed by how long priority tasks can wait if all the processors are busy. The worst case arises when all processors have just begun the longest strip. In the IMP application, the most urgent tasks can afford to wait a maximum of 400 microseconds. Therefore, strips must not be longer than that. (Of course, a strip might be longer if it is run infrequently and if the urgent tasks do not have absolute time requirements. That is, one might build a statistically acceptable set of strip lengths.)

An inherent part of multiprocessor operation is the locking of critical resources[10]. This is the mechanism by which the algorithm enforces sequentiality when it is needed. Our system uses an uninterruptable load-and-clear operation (load an accumulator with the contents of a memory location and then clear the location) as its primitive locking facility (i.e., as the necessary multiprocessor lock equivalent to Dijkstra semaphores). To avoid deadlocks, we assign a priority ordering to our resources and arrange that the software not lock one resource when it has already locked another of lower or equal priority.

Using these facilities and techniques, the logical structure of the IMP program is similar to that described for the 516/316 machines. In addition there is a substantial new program segment devoted to maintaining the hardware and keeping the program running in the face of hardware failures.

Pluribus IMP Reliability*

Computer reliability is a common, serious, and difficult problem which has been approached in many ways. For critical applications (e.g., space exploration), large amounts of money are spent to overcome such apparently trivial weaknesses as problematical power supplies and connectors. Although a great deal of attention is given to tailoring computers to particular job environments, the commercial world of computer manufacturers has provided no adequate answer to the reliability problem.

The notions of fault-tolerant and fail-soft systems have been around for a number of years and because the reliability is such a crucial issue in a communications network, it was decided that some of these ideas should be exploited in the design of the Pluribus IMP.

- - - - - - - - - -

*A more complete account of the topics discussed in this section can be found in Reference 11.

The reliability goal of the Pluribus IMP is not that the system should never break, but rather that it should recuperate automatically within seconds or minutes from most troubles and that the probability of total failure should be dramatically reduced over traditional machines, say to once a year or less. The system should survive not only transient failures but also solid failures of any single component. It is not necessary to operate correctly most of the time so long as outages are infrequent, kept brief, and fixed without human intervention. Outages of a few seconds are tolerated easily, and outages of many seconds, while causing the particular node to become temporarily unusable, will not in general jeopardize operation of the network as a whole.

Achieving this sort of reliability requires hardware that will survive any single failure, even a solid one, in such a way as to leave a potentially runnable machine intact (potentially in that it may need resetting, reloading, etc.). Second, it requires all of the facilities necessary to survive any and all transients stemming from the failure and to adapt to running in the new hardware configuration. To provide adequate hardware, extra copies of every vital hardware resource are included. Sufficient isolation is provided between the copies so that any single component failure will impair only one copy.

Appropriate hardware. It is not sufficient merely to provide duplicate copies of a particular resource; it is necessary to assure that the copies are not dependent on any common resource. Thus, for example, in addition to providing multiple memories, there are multiple busses on which the memories are distributed. Furthermore, each bus is not only logically independent, but also physically modular. The chassis, with its own power supply and cooling, is built into an integral unit which may be powered down, disconnected, and removed from the rack for servicing or replacement while the rest of the machine continues to run.

All central system resources, such as the real time clock and the PID, are duplicated on at least two separate I/O busses. All connections between bus pairs are provided by separate bus couplers so that a coupler failure can disable at most the two busses it is connecting; all other interconnections between busses are unaffected.

When a particular communications circuit is deemed critical, it is connected to two identical interface units (on separate I/O busses), either of which may be selected for use by the program. When the extra reliability is not worth the extra cost, the line is only singly connected.

In order for the system to adapt to different hardware configurations, facilities have been provided which make it convenient for the software to search for and locate those resources which are present and to determine the type and parameters of those which are found.

To allow for active failures, all bus couplers have a program-controllable switch that inhibits transactions via that coupler. Thus, a "malicious" bus may be effectively "amputated" by turning off all couplers from that bus. These switches are protected from capricious use by requiring a password. Naturally an amputated processor has no access to these switches.

Software survival. With the above features, the Pluribus hardware can experience any single component failure and still present a runnable system. One must assume that as a consequence of a failure, the program may have been destroyed, the processors halted, and the hardware put in some hung state needing to be reset. Three broad strategies have guided the means used to restore the algorithm to operation after a failure: keep it simple, worry about redundancy, and use watchdog timers throughout.

SIMPLICITY: First, all processors are identical and equal; they are viewed only as resources used to advance the algorithm. Each is able to do any system task; none is singled out (except momentarily) for a particular function. A consequence of this is that the full power of the machine can be brought to bear on the part of the algorithm which is busiest at a given time. A further consequence is that should any processor fail, the rest will continue to perform the necessary tasks, albeit at reduced capacity.

A second system characteristic which arose from a desire to keep things simple is passivity. The terms active and passive describe communication between subsystems in which the receiver is expected to put aside what it is doing and respond. The quicker the required response, the more active the interaction. In general, the more passive the communication, the simpler the receiver can be, because it can wait until a convenient time to process the communication. Neither the hardware interfaces nor other processors tell a processor what to do; rather, tasks to be done are posted in the PID and processors ask the PID what should be done next.

There are some costs to such a passive system: first, the resulting slower responsiveness has necessitated additional buffering in some of the interfaces; second, the program must regularly break from tasks being executed to check the PID for more important tasks. The alternatives, however, are far worse. In a more active system, for example one which uses classical priority interrupts, it is difficult to decide which processor to switch to the new task. The possibilities for deadlocks are frightening, and the general mechanism to resolve them cumbersome.

As a third example of simplicity, the entire system is broken into reliability subsystems which are parts of the overall system that verify one another in an orderly fashion. The subsystems are cleanly bounded with well-defined interfaces. They are self-contained in that each includes a self-test mechanism and reset capability. They are isolated in that all communication between subsystems takes place passively via data structures. Complete interlocking is provided at the boundary of every subsystem so that the subsystems can operate asynchronously with respect to one another.

The monitoring of one subsystem by another is performed using timer modules, as discussed below. These timer modules guarantee that the self-test mechanism of each subsystem operates, and this in turn guarantees that the entire subsystem is operating properly.

REDUNDANCY: Redundancy is simultaneously a blessing and a curse. It occurs in the hardware and the software, and in both control and data paths. We deliberately introduce redundancy to provide reliability and promote efficiency, and it frequently occurs because it is a natural way to build things. On the other hand the mere existence of redundancy implies a possible disagreement between the versions of the information. Such inconsistencies usually lead to erroneous behavior, and often persist for long periods.

There are several methods of dealing with redundancy. The first and best is to eliminate it, and always refer to a single copy of the information. When we choose not to eliminate it, we check the redundancy and explicitly detect and correct any inconsistencies. It does not really matter how this is done as the system is recovering from a failure anyway. What is important is to resolve the inconsistency and keep the algorithm moving. Sometimes it is too difficult to test for inconsistency; then timers are used as discussed below.

TIMERS: There is a uniform structure for implementing a monitoring function between reliability subsystems based on watchdog timers. Consider a subsystem which is being monitored. Such a subsystem is designed to cycle with a characteristic time constant, and a complete self-consistency check is included within every cycle. Regular passage through this cycle is therefore sufficient indication of correct operation of the subsystem. If excessive time goes by without passage through the cycle, it implies that the subsystem has had a failure from which it has not been able to recover by itself. The mechanism for monitoring the cycle is a timer which is reset by every passage through the cycle. There are both hardware and software timers ranging from five microseconds to two minutes in duration in the IMP system. Another subsystem monitors this timer and takes corrective action if the timer ever runs out. To avoid the necessity for subsystems to be aware of one another's internal structure, each subsystem includes a reset mechanism which may be externally activated. Thus, corrective action consists merely of invoking this reset. The reset algorithm is assumed to work although a particular incarnation in code may fail because it gets damaged. In such a case another subsystem (the code checksummer) will shortly repair the damage.

The entire system consists of a chain of subsystems in which each subsystem monitors the next member of the chain. Lower subsystems provide and certify some important

environmental feature used by higher level systems. For example, a low level code tester checksums all code (including itself), insures that all subsystems are receiving a share of the processor's attention, and guarantees that locks do not hang up. It thus guarantees the most basic features for all higher levels. These will, in turn, provide further environmental features, such as a list of working memory areas, I/O devices, etc., to still higher levels.

Before they can work together to run the main system, a common environment must be established for all processors. The process of reaching an agreement about this environment is called "forming a consensus", and the group of agreeing processors is known as the Consensus. An example of a task requiring consensus is the identification of usable common memory and the assignment of functions (code, variables, buffers, etc.) to particular pages.

The Consensus maintains and counts down a timer for every processor in the system in order to detect uncooperative or dead processors. This monitoring mechanism includes reloading the failing processor's local memory and restarting it. Reliance on the Consensus is vulnerable to simultaneous transient failure of all processors. For many cases (as for example when all of the processors halt), a simple reset consisting of a one-second timer on the bus and a 60 Hz interrupt routine suffices.

For more catastrophic failures the machine can be reset, reloaded, and restarted directly from the Network Control Center, which depends on the continual presence of human operators for successful operation. It is correspondingly powerful, resourceful, and erratic in its behavior.

SUMMARY

We have described our several-year effort to build packet-switches. The original pair of switches was to: 1) be an off-the-shelf product with many copies in the field, 2) be physically small, 3) be as cheap as possible, 4) be suitable for long periods of unattended operation, 5) preferably offer standard ruggedized options, 6) permit numerous independent buffered block transfers with completion interrupts, 7) connect to several 50-kilobit lines, 8) connect to diverse types of Hosts, 9) provide throughput to handle reasonable peak loads with a safety factor of two or three, 10) be manufactured by a firm able to assist in the design and production of the specialized interface hardware, provide maintenance, etc., 11) be delivered in 14 months. The Honeywell DDP-516 was chosen as the most suitable machine after careful review of all computers in its general class available at that time (1968). A 316 version was built later when this less expensive variant appeared on the market. A minimum of redesign permitted substantial cost savings.

In 1970, design of a completely new packet-switch was begun. This effort was founded on a desire for: 1) greater flexibility in configuration, 2) higher bandwidth capabilities, 3) significant savings on small configurations, 4) the ability to handle much larger numbers of interfaces if required, 5) the ability to attach more memory easily if required, and 6) substantial improvement in reliability.

The new machine architecture is based upon interconnected Lockheed SUE communication busses and is known as the Pluribus. After construction of a 13-processor, two-memory-bus, two-I/O-bus prototype, one of two production copies is presently (fall 1974) being installed in the ARPA Network.

ACKNOWLEDGMENTS

Since 1969 our efforts in developing packet switches have been supported by the Information Processing Techniques Office of the Advanced Research Projects Agency. At Bolt Beranek and Newman Inc. this work has been a team effort of which the present authors are only a small part, and we gratefully acknowledge the contributions of all the other team members whose work is reported here without explicit assignment of credit. R. Brooks, B. Erwin, N. Graham, and J. Moore helped with the preparation of the manuscript.

REFERENCES

1. L.G. Roberts and B.D. Wessler, "Computer Network Development to Achieve Resource Sharing," AFIPS Conference Proceedings, Vol. 36, June 1970, pp. 543-549.

2. F.E. Heart, R.E. Kahn, S.M. Ornstein, W.R. Crowther, and D.C. Walden, "The Interface Message Processor for the ARPA Computer Network," AFIPS Conference Proceedings, Vol. 36, June 1970, pp. 551-567; also in Advances in Computer Communications, W.W. Chu ed.), Artech House Inc., 1974, pp 300-316.

3. S.M. Ornstein, F.E. Heart, W.R. Crowther, S.B. Russell, H.K. Rising, and A. Michel, "The Terminal IMP for the ARPA Computer Network," AFIPS Conference Proceedings, Vol. 40, June 1972, pp. 243-254.

4. R.E. Kahn and W.R. Crowther, "Flow Control in a Resource-Sharing Computer Network," Proceedings of the Second ACM/IEEE Symposium on Problems in the Optimization of Data Communications Systems, Palo Alto, California, October 1971, pp. 108-116.

5. J.M. McQuillan, W.R. Crowther, B.P. Cosell, D.C. Walden, and F.E. Heart, "Improvements in the Design and Performance of the ARPA Network," AFIPS Conference Proceedings, Vol. 41, December 1972, pp. 741-754.

6. F.E. Heart, S.M. Ornstein, W.R. Crowther, and W.B. Barker, "A New Minicomputer/Multiprocessor for the ARPA Network," AFIPS Conference Proceedings, Vol. 42, June 1973, pp. 529-537.

7. S.M. Ornstein, W.B. Barker, R.D. Bressler, W.R. Crowther, F.E. Heart, M.F. Kraley, A. Michel, and M.J. Thrope, "The BBN Multiprocessor," Proceedings of the Seventh Annual Hawaii International Conference on System Sciences, Honolulu, Hawaii, January 1974, Computer Nets Supplement, pp. 92-95.

8. A.A. McKenzie, B.P. Cosell, J.M. McQuillan, and M.J. Thrope, "The Network Control Center for the ARPA Network," Proceedings of the First International Conference on Computer Communication, Washington, D.C., October 1972, pp. 185-191.

9. L. Kleinrock and W. Naylor, "On Measured Behavior of the ARPA Network," AFIPS Conference Proceedings, Vol. 43, May 1974, pp. 767-780.

10. E.W. Dijkstra, "Cooperating Sequential Processes," in Programming Languages, ed. F. Genuys, Academic Press, London and New York 1968, pp. 43-112.

11. S.M. Ornstein, W.R. Crowther, M.F. Kraley, R.D. Bressler, A. Michel, and F.E. Heart, "Pluribus--A Reliable Multiprocessor," submitted to the AFIPS 1975 National Computer Conference.

PLURIBUS — A RELIABLE MULTIPROCESSOR

by S. M. Ornstein, W. R. Crowther, M. F. Kraley,
R. D. Bressler, A. Michel, and F. E. Heart

Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

November 1974

PLURIBUS — A RELIABLE MULTIPROCESSOR*

*This work was supported by the Advanced Research Projects Agency (ARPA) under contracts DAHC15-69-C-0179 and F08606-73-C-0027.

by S. M. Ornstein, W. R. Crowther, M. F. Kraley, R. D. Bressler, A. Michel, and F. E. Heart
Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

INTRODUCTION

As computer technology has evolved, system architects have continually sought new ways to exploit the decreasing costs of system components. One approach has been to pull together collections of units into multiprocessor systems.[1] Usually the objectives have been to gain increased operating power through parallelism and/or to gain increased system reliability through redundancy.

In 1972, our group at Bolt Beranek and Newman started to design a new machine for use as a switching node (IMP) in the ARPA Network.[2,3] The machine was to be capable of high bandwidth, in order to handle the 1.5-megabaud data circuits which were then planned for the network. It was to have a high fanout to Host computers connected at a node. It was to come in all sizes (of processing power, memory, I/O) so that one could configure an

individual IMP to meet the requirements of its particular location in the network, and change that configuration easily should the requirements change. Most of all, it was to be reliable.

The family of machines we have produced which meets these goals has been named the Pluribus line. The machines are highly modular at several levels and have a minicomputer/multiprocessor architecture. Although the largest configuration we have put together so far contains only 13 processors, we believe there are no inherent problems with considerably larger systems. The structure and details of some of the hardware are described in earlier papers.[4,5] Familiarity with these papers will be helpful in understanding the present paper, which focuses on the issue of reliability. We believe that reliability will become an increasingly common concern as multiprocessors become more commonplace, and we believe that we have gained some interesting insights into the solution of this problem.

THE MULTIPROCESSOR ARCHITECTURE

A novel feature of our design is the consistent treatment of all processors as equal units, both in the hardware and in the software. There is no specialization of processors for particular system functions, and no assignment of priority among the processors, such as designating one as master. We chose to distribute among the processors not only the application job (the IMP job) but also the multiprocessor control and reliability jobs, treating all jobs uniformly. We view the processors as a resource used to

advance our algorithm; the identity of the processor performing a particular task is of no importance. Programs are written as for a single processor except that the algorithm includes interlocks necessary to insure multiprocessor sequentiality when required. The software of our machine consists of a single conventional program run by all processors. Each processor has its own local copy of about one quarter of this program and the remaining three quarters is in commonly accessible memory.

Hardware Structure

Reliability was a main concern in planning the hardware architecture. Although we tried to build the individual pieces solidly, our main goal was to provide hardware which could be exploited by the program to survive the failure of any individual component.

The hardware consists of busses joined together by special bus couplers which allow units on one bus to access those on another. Each bus, together with its own power supply and cooling, is mounted in its own modular unit, permitting flexible variation in the size and structure of systems. There are processor busses each of which contains two processors, each in turn with its own local 4K memory which stores frequently run and recovery-related code. There are memory busses to house the segments of a large memory common tc all the processors. Finally, there are I/O busses which house device controllers as well as

certain central resources such as system clocks and special (priority-ordered) task disbursers which replace the traditional priority interrupt system. About half of the machine consists of standard parts from the Lockheed SUE line; the remainder is of special design.

As emphasized in our initial paper,[4] we were fortunate to have a very specific job in mind as we designed the system. This enabled us to place specific bounds on the problems we sought to solve. For example, the proposed initial setting within a communications network means that outside entities (neighboring communications processors, Hosts, users, etc.) may help to notice that things are going wrong. It also means that recovery assistance is potentially available from the Network Control Center (NCC) through the network.[6,7] The system is designed generally to avoid reliance upon external help, but upon occasion such help is useful and therefore we have provided methods for permitting the system to be forcibly reloaded and restarted via the network.

Software Structure

The problem of building a packet-switching store-and-forward communications processor (the IMP) lends itself especially well to parallel solution since packets of data can be treated independently of one another. Other functions, such as routing computations, can also be performed in parallel.

The program is first divided into small pieces, called *strips*, each of which handles a particular aspect of the job. When a task needs to be performed, the name (number) of the appropriate strip is put on a queue of tasks to be run. Each processor, when it is not running a strip, repeatedly checks this queue. When a strip number appears on the queue, the next available processor will take it off the queue and execute the corresponding strip. We try to break the program into strips in such a way that a minimum of context saving is necessary.

The number assigned to each strip reflects the priority of the task it performs. When a processor checks the task queue, it takes the highest priority waiting job. Since all processors access this queue frequently, contention for it is very high. We therefore built a hardware device called the Pseudo Interrupt Device (PID) which serves as a task queue. A single instruction allows the highest priority task to be fetched and removed from the queue. Another instruction allows a new task to be put onto the queue. All contention is arbitrated by standard bus logic hardware.

The length of strips is governed by how long priority tasks can wait if all the processors are busy. The worst case arises when all processors have just begun the longest strip. In the IMP application, the most urgent tasks can afford to wait a maximum of 400 microseconds. Therefore, strips must not generally be

longer than that.

An inherent part of multiprocessor operation is the locking of critical resources to enforce sequentiality when necessary.[8] A load-and-clear operation provides our primitive locking facility. To avoid deadlocks, we priority-order our resources and arrange that the software not lock one resource when it has already locked another of lower or equal priority.

Status

During the early spring of 1974 a prototype 13-processor system was constructed. As this paper is being written (in the fall of 1974) two production copies have been constructed and are running. Each contains 13 processors, two memory busses, and two I/O busses. These machines have been connected intermittently into the ARPA Network for testing purposes and operational installation in the network is anticipated shortly. A single processor has been running on the network for an extended period in order to validate performance during routine operation. Three Satellite IMP configurations[9] are presently under construction as well as a non-IMP configuration designed to provide highly reliable pre-processing and forwarding of seismic data to processing and storage centers.

RELIABILITY GOALS

Since the term "reliable system" can have many different meanings, it is important to establish clearly just what we are and what we are

not trying to achieve. We are not trying to build a non-failing device (as in [10]); instead, we are trying to build a system which will recuperate automatically within seconds, or at most minutes, following a failure. Furthermore, we want the system to survive not only transient failures but also solid failures of any single component. In many cases (such as the IMP job) it is not necessary to operate continuously and perfectly: it suffices to operate correctly most of the time so long as outages are infrequent, kept brief, and fixed without human intervention.

How one copes with infrequent brief outages depends on what one is trying to do. For tasks not tightly coupled to real-time requirements (e.g., for many large numerical computations), a simple device is to choose checkpoints at which to record the state of the system so that one can always recover by restarting from the checkpoint just preceding an outage.[11,12] The IMP system happens to be embedded in a larger system which is quite forgiving. (This is not an uncommon situation.) Thus brief outages of a few seconds are tolerated easily, and outages of many seconds, while causing the particular node to become temporarily unusable, will not in general jeopardize operation of the network as a whole.

Occasionally, despite all efforts, the system will break so catastrophically that it will be unable to recover. Our goal is to reduce the probability of such total system failure to the probability of a multiple hardware failure. We do not try to

protect against all possible errors; some of our procedures will fail to detect errors whose probability of occurrence is sufficiently low. For example, all code is periodically checksummed using a 16-bit checksum. A failure that does not disturb the validity of the checksum may not be detected. We do not mind if a failure renders large sections of the machine unusable or inaccessible, providing enough remains to run the system. The presence of runnable hardware, however, is not sufficient to guarantee that operation will be resumed; in addition, the software must be able to survive the transients accompanying the failure and adapt to the remaining hardware. This may include combating and overcoming active failures (for example, when an element such as a processor goes berserk and repeatedly writes meaningless data into memory).

All code is presumed to be debugged -- i.e., all frequently occurring problems will have been fixed. On the other hand, we must be able to survive infrequent bugs even when they randomly destroy code, data structures, etc.

In order to avoid complete system failure, a failed component must be repaired or replaced before its backup also breaks. The system must therefore report all failures. The actual repair and/or replacement will of course be performed by humans, but this will generally take place long after the system has noted the failure and reconfigures itself to bypass the failed module. The ratio of mean-time-to-repair to mean-time-between-failures will

determine overall system reliability. It must also be possible to remove and replace any component while the system continued to run. Finally, the system should absorb repaired or newly introduced parts gracefully.

STRATEGIES

In order to understand our system it is convenient to consider the strategies used to achieve our goals in two parts which more or less parallel the traditional division into hardware and software. The first part provides hardware that will survive any single failure, even a solid one, in such a way as to leave a potentially runnable machine intact (potentially in that it may need resetting, reloading, etc.). The second part provides all of the facilities necessary to survive any and all transients stemming from the failure and to adapt to running in the new hardware configuration.

Appropriate Hardware

We have two basic strategies in providing the hardware. The first is to include extra copies of every vital hardware resource. The second is to provide sufficient isolation between the copies so that any single component failure will impair only one copy.

To increase effective bandwidth in multiprocessors, multiple copies of heavily utilized resources are normally provided. For reliability, however, *all* resources critical to running the algorithm are duplicated. Where possible the system utilizes

these extra resources to increase the bandwidth of the system.

It is not sufficient merely to provide duplicate copies of a particular resource; we must also be sure that the copies are not dependent on any common resource. Thus, for example, in addition to providing multiple memories, we also include logically independent, physically modular, multiple busses on which the memories are distributed. Each bus has its own power supply and cooling, and may be disconnected and removed from the racks for servicing while the rest of the machine continues to run.

All central system resources, such as the real time clock and the PIC, are duplicated on at least two separate I/O busses. All connections between bus pairs are provided by separate bus couplers so that a coupler failure can disable at most the two busses it is connecting.

Non-central resources, such as individual I/O interfaces, are generally less critical. Provision has been made, however, to connect important lines to two identical interface units (on separate I/O busses) either of which may be selected for use by the program.

To adapt to different hardware configurations, the software must be able to determine what hardware resources are available to it. We have made it convenient to search for and locate those resources which are present and determine the type and parameters of those which are found.

To allow for active failures, all bus couplers have a program-controllable switch that inhibits transactions via that coupler. Thus, a bus may be effectively "amputated" by turning off all couplers from that bus. This mechanism is protected from capricious use by requiring a particular data word (a password) to be stored in a control register of the bus coupler. Naturally an amputated processor is prevented from accessing these passwords.

Finally, although a common reset line is normally considered essential, we have avoided such a line since a single failure on its driver could jeopardize the entire system. There is thus no central point (not even a single power switch) where one can gain control of the entire system at once. Instead, we rely on resetting a section at a time using passwords.

Software Survival

With the above features, the Pluribus hardware can experience any single component failure and still present a runnable system. One must assume that as a consequence of a failure, the program may have been destroyed, the processors halted, and the hardware put in some hung state needing to be reset. We now investigate the means used to restore the algorithm to operation after a failure. The various techniques for doing this may be classified under three broad strategies: keep it simple, worry about redundancy, and use watchdog timers throughout.

Simplicity

It is always good to keep a system simple, for then one

has a fighting chance to make it work. We describe here three system constraints imposed in the name of simplicity.

First, as mentioned above, we insist that all processors be identical and equal: they are viewed only as resources used to advance the algorithm. Each should be able to do any system task; none should be singled out (except momentarily) for a particular function. The important thing is the algorithm. With this view it is clear that it is simplest if the algorithm is accessible to all processors of the system. A consequence of this is that the full power of the machine can be brought to bear on the part of the algorithm which is busiest at a given time.

One might argue that for some systems it is in fact simpler (or more efficient) to specialize processors to specific tasks. One could, in such a case, then duplicate each different type for reliability. With that approach, however, one must worry about the recovery of several different types of units, and all the possible interactions between them. We consider the recovery problem for a group of identical machines formidable enough.

One consequence of treating all processors equally is that a program can be debugged on a single machine up to the point where the multiple machine interaction matters. Once this has been done, we have found that processor interaction does not present a severe additional debugging problem. On the other hand, finding routine software bugs when a dozen machines are running

is a difficult problem.

A second characteristic of our system which arose from a desire to keep things simple is passivity. We use the terms active and passive to describe communication between subsystems in which the receiver is expected to put aside what it is doing and respond. The quicker the required response, the more active the interaction. In general, the more passive the communication, the simpler the receiver can be, because it can wait until a convenient time to process the communication. On the other hand the slower response may complicate things for the sender. We believe that there is a net gain in using more passive systems. An example of this is our decision to make the task disbursing mechanics (the PID) a passive device. Neither the hardware interfaces nor other processors tell a processor what to do; rather, processors ask the PID what should be done next. There are some costs to such a passive system. The resulting slower responsiveness has necessitated additional buffering in some of our interfaces. In addition, the program must regularly break from tasks being executed to check the PID for more important tasks.

The alternatives, however, are far worse. In a more active system, for example one which uses classical priority interrupts, it is difficult to decide which processor to switch to the new task. Furthermore, it is almost impossible to preserve the context of a processor[13] while making such a switch because of the interaction

with the resource interlocking system. The possibilities for deadlocks are frightening, and the general mechanism to resolve them cumbersome. With a passive system a processor finishes one task before requesting the next, thus guaranteeing that task switching occurs at a time when there is little context, e.g., no resources are locked.

Passive systems are more reliable for another reason: namely, the recovery mechanisms tend to be far simpler than those for active systems.

As a third example of simplicity we introduce the notion of a reliability subsystem. A reliability subsystem is a part of the overall system which is verified as a unit. A subsystem may include a related set of hardware, program, and/or data structures. The boundaries of these reliability subsystems are not necessarily related at all to the boundaries of the hardware subsystems (processors, busses, memories, etc.) described earlier. The entire system is broken into these subsystems, which verify one another in an orderly fashion.

The subsystems are cleanly bounded with well-defined interfaces. They are self-contained in that each includes a self-test mechanism and reset capability. They are isolated in that all communication between subsystems takes place passively via data structures. Complete interlocking is provided at the boundary of every subsystem so that the subsystems can operate asynchronously with respect to one another.

The monitoring of one subsystem by another is performed using timer modules, as discussed below. These timer modules guarantee that the self-test mechanism of each subsystem operates, and this in turn guarantees that the entire subsystem is operating properly.

Redundancy

Redundancy is simultaneously a blessing and a curse. It occurs in the hardware and the software, and in both control and data paths. We deliverately introduce redundancy to provide reliability and to promote efficiency, and it frequently occurs because it is a natural way to build things. On the other hand the mere existence of redundancy implies a possible disagreement between the versions of the information. Such inconsistencies usually lead to erroneous behavior, and often persist for long periods.

It was not until we adopted a strategy of systematically searching out and identifying all the redundancy in every subsystem that we succeeded in making the subsystems reliable. This process therefore constitutes one of our three basic strategies for constructing robust software.

We use the term redundancy here in a somewhat subtle sense, not only for cases in which the same information is stored in two places, but also when two stored pieces of information each imply a common fact although neither is necessarily sufficient to imply the other.

There are several methods of dealing with redundancy. The

first and best is to eliminate it, and always refer to a single copy of the information. When we choose not to eliminate it, we can check the redundancy and explicitly detect and correct any inconsistencies. It does not really matter how this is done as the system is recovering from a failure anyway. What is important is to resolve the inconsistency and keep the algorithm moving. Sometimes it is too difficult to test for inconsistency; then timers can be used as discussed in the next section.

Let us consider a few examples of redundancy to make these ideas more concrete.

- A buffer holding a message to be processed, and a pointer to the buffer on a "to be processed" queue -- if the buffer and queue are inconsistent, the buffer will not be processed. Each buffer has its own timer and if not processed in a reasonable time, it will be replaced on the queue.
- A device requesting a bus cycle, and a request capturing flip-flop in the bus arbiter -- if the arbiter and device disagree, the bus may hang. A timer resets the bus after one second of inactivity.
- One processor seeing a memory word at a particular system address and another seeing the same word at the same address -- The software watches for inconsistencies and when they occur declares the memory or one of the processors

unusable.

- The PID level used by a particular device and the device serviced in response to that level -- The PID level(s) used by each device are program-readable. A process periodically reads them and forces the tables driving the program's response to agree.

Timers

We have adopted a uniform structure for implementing a monitoring function between reliability subsystems based on watchdog timers. Consider a subsystem which is being monitored. We design such a subsystem to cycle with a characteristic time constant and insist that a complete self-consistency check be included within every cycle. Regular passage through this cycle therefore is sufficient indication of correct operation of the subsystem. If excessive time goes by without passage through the cycle, it implies that the subsystem has had a failure from which it has not been able to recover by itself. The mechanism for monitoring the cycle is a timer which is restarted by every passage through the cycle. We have both hardware and software timers ranging from five microseconds to two minutes in duration. Another subsystem can monitor this timer and take corrective action if it ever runs out. To avoid the necessity for subsystems to be aware of one another's internal structure, each subsystem includes a reset mechanism which may be externally activated. Thus corrective action consists

merely of invoking this reset. The reset algorithm is assumed to work although a particular incarnation in code may fail because it gets damaged. In such a case another subsystem (the code checksummer) will shortly repair the damage.

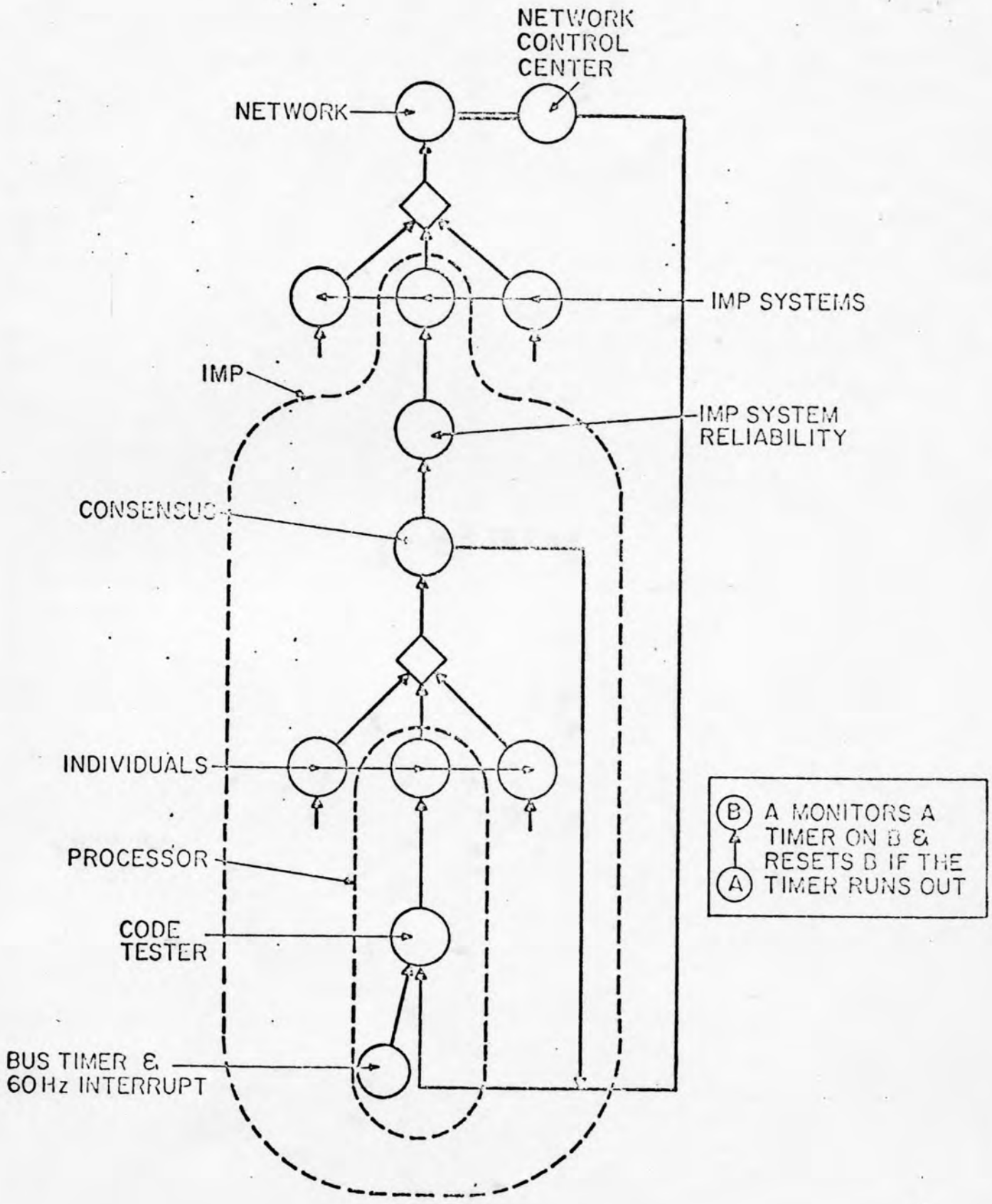
Note that we have introduced an active element into our otherwise totally passive system. These resets constitute the only active elements and furthermore are invoked only after a failure has occurred. This approach seems to provide for the maximum isolation between subsystems.

SYSTEM RELIABILITY STRUCTURE

In the previous section we described a mechanism whereby one subsystem can monitor another. Our system consists of a chain of subsystems in which each subsystem monitors the next member of the chain. Figure 1 and Table I show this structure in the system we have built for the IMP. An efficient way to build such a chain is to have lower subsystems provide and guarantee some important environmental feature used by higher level systems. For example, a low level in our chain guarantees the integrity of code for higher levels which thus assume the correctness of code. Such a system is vulnerable only at its bottom. (We are assuming here that we have runnable hardware although it may be in a bad state, requiring resetting.) The code tester level (see Figure 1) serves three functions: first, it checksums all low level code (including itself); second, it insures that control

Fig. 1

FIGURE 1 RELIABILITY STRUCTURE



is operating properly, i.e., that all subsystems are receiving a share of the processors' attention; third, it guarantees that locks do not hang up. It thus guarantees the most basic features for all higher levels. These will, in turn, provide further environmental features, such as a list of working memory areas, I/O devices, etc., to still higher levels. The method by which the code tester subsystem itself is monitored and reset will be discussed shortly.

Table I

Major Subsystems and their Functions

IMP SYSTEM: Watches network behavior - will not cooperate with irresponsible network behavior.

IMP SYSTEM RELIABILITY: Watches IMP SYSTEM (data structures mostly).

CONSENSUS: Watches IMP SYSTEM RELIABILITY, verifies all Common Memory Code (via checksum), watches each processor, finds all usable hardware resources (interfaces, PIDs, memory, processors, etc.), tests each and creates a table of good ones. Makes spare copies of code.

INDIVIDUAL: Watches CONSENSUS, finds all memory and processors it considers usable, determines where the Consensus is communicating and tries to join it.

CODE TESTER: Watches INDIVIDUAL, verifies all Local Memory Code (via a checksum), guarantees control and lock mechanisms.

BUS TIMER + 60Hz INTERRUPT: Watches CODE TESTER, guarantees bus activity.

The mechanisms we have described ensure that the separate processor subsystems have a satisfactory local environment in which to work. Before they can work together to run the main system it is necessary that a common environment be established for all processors. We call the process of reaching an agreement about this environment "forming a consensus", and we dub the group of agreeing processors the Consensus. The work done by the Consensus is in fact performed by individual processors communicating via common memory, but the coordination and discipline imposed on Consensus members make them behave like a single logical entity. An example of a task requiring consensus is the identification of usable common memory and the assignment of functions (code, variables, buffers, etc.) to particular pages. The members of the Consensus will not in general agree in their view of the environment, as for example when a broken bus coupler blinds one member to a segment of common memory. In this case the Consensus, including the processor with the broken coupler, will agree to run the main system without that processor.

The Consensus maintains a timer for every processor in the system, whether or not the processor is working. The Consensus will count down these timers in order to eliminate uncooperative or dead processors. In order to join the Consensus, a processor need merely register its desire to join by holding off its timer. Within the individual processors it is the code tester subsystem which holds off the timer.

The Consensus, then, acting as a group, provides the monitoring mechanism for the individuals as shown by the feedback monitoring path in Figure 1. This monitoring mechanism run by the Consensus includes the usual reset capability which in this case means reloading the individual's local memory and restarting the processor. Since all of the processors have identical memories, reloading is not difficult. We provide (password protected) paths whereby any processor can reset, reload, and restart any other processor. This reliance on the Consensus is indeed vulnerable to a simultaneous transient failure of all processors. However, the Network Control Center has access to these same reset and reload facilities and these enable it to perform the reload function remotely (a path also shown in the figure).

Thus the Consensus and/or Network Control Center are the ultimate guarantors of the lowest level subsystem. While this process is sufficient it is sometimes slow. For many cases in which the Consensus is disabled (as for example when all of the

processors halt), a simpler reset without reloading will suffice. For this reason we have provided a simpler and more immediate (if redundant) mechanism in each processor for resetting the control and lock systems. We implement this mechanism in software with the assistance of a 60Hz interrupt and a one-second timer on the bus. Together these provide a somewhat vulnerable but much quicker alternative to the more ponderous NCC/Consensus resets.

There is a problem about what area of common memory the processors should use in which to form the Consensus, since failures may make any predetermined system address inaccessible. To allow for this, sufficient communication is maintained in all pages of common memory to reach agreement both as to which processors are in the Consensus and where further communication is to take place.

To protect itself from broken processors, the Consensus amputates all processors which do not succeed in joining it. There is a conflict between this need to protect itself and the need to admit new or healed processors into the Consensus. The amputation barrier is therfore lowered for a brief period each time the Consensus tries to restart a processor. This restart is in fact the reset based on the timer held off by the code tester subsystem, as discussed above. In the case of certain active failures, even this brief relaxation may cause trouble. In these cases the Consensus will decide to keep the barrier up continuously.

Certain active failures may prevent the formation of a consensus. In such a situation each processor will behave as if it were a Consensus (of one) and will try to amputate all other processors. At the point when the actively failing component is amputated, the remaining processors will be able to form a consensus. From this point the system behaves as described above.

Further up in the figure there is another joining of independent units, namely IMPs joining to form the network. The analogy here is incomplete because the ARPA Network was not built with these concepts in mind. There is collective behavior, e.g., routing, and individual behavior which accepts collective decisions only after they pass reasonability tests. However, the reliability features of the network are concentrated in the Network Control Center, which depends on the continual presence of human operators for successful operation. It is correspondingly powerful, resourceful, and erratic in its behavior.

SOME EXAMPLES OF FAILURES

In order to explain in more practical terms some of the reliability mechanisms, we now discuss a number of specific failures and describe the methods which detect and repair the resulting damage. In each case, we focus on the component that failed and the particular mechanism that takes care of that failure. Derivative failures may well take place, and other mechanisms will handle these, since all mechanisms operate all the time.

These examples are set in the context of the IMP application and the severity of their direct consequences rated on the following scale:

1. Momentary slowdown - no data loss
2. Loss of data (a network message)
3. Temporary loss of some IMP function (a network link)
4. Momentary total IMP outage with local self-recovery
5. Outage requiring reloading via the network
6 Failure requiring human insight for debugging.

Example 1. Suppose first that a bus coupler experiences a transient failure on a single reference to common memory, which leaves one word of common memory with the wrong contents but correct parity. Suppose further that the failure is subtle, in the sense that there is no obvious ill effect on processor control, like halting or looping, which will be caught by lower level mechanisms. We will focus first on examples which cause minimal disruption and where detection and gentle recovery are the primary concerns. We consider four examples of transient memory failures:

Example 1.a Suppose that a word of text in one of the messages we are deliverying becomes smashed. There is a checksum on all messages and the network will notice at one of its checkpoints that the message has gone bad. The source will be prompted to send a new copy. (Severity 2)

Example 1.b Near the heart of our system is a queue of unused buffers called the free list. Suppose the failure is in the structure of this queue. The system explicitly tests for both a looped queue and wrong things on the queue. A more subtle form of error occurs when some buffers which should be on the queue are missing from it. Our system is designed so that a buffer should be removed from the free list for at most two minutes at a time. A timer is maintained on each buffer, which is restarted whenever the buffer returns to the free list. Should any timer ever run out, its buffer is forced back onto the free list. The result of this failure will be a degradation of system performance as it attempts to run with fewer buffers for a short while, followed by complete recovery within two minutes. The IMP will stay up and no messages will be lost. (Severity 1)

Example 1.c Suppose that one of the locks on a resource is broken so that it wrongly locks the resource. Any subsystem which tries to use the resource will put a processor into a tight loop waiting for the resource to become free. In about 1/15 sec. this will cause the processor's timer, driven off its 60Hz clock interrupt, to run out. Upon investigation, the program will notice that the subsystem is waiting for a locked resource, and arbitrarily unlock it. Aside from the 1/15 sec. pause, the system will be unaffected by the transient. (Compare the simplicity of this scheme with [14].) (Severity 1)

Example 1.d Suppose now that a failure strikes common memory holding code, and that the trouble is subtle -- either the code is not run often or the change has no immediate drastic effect. In a few seconds the processors will begin to notice that the checksum on that piece of code is bad and stop running it. Shortly the whole Consensus will agree, and will switch over to use the memory holding the spare copy of that code. Unless the broken code has already caused some other trouble, the problem is thereby fixed, with only momentary slowdown. (Severity 1)

Example 2. Suppose a processor fails by suddenly and permanently stopping. The immediate effect will be that some task will be left half done, with a high probability that some resource is locked. This looks to the system like a data failure, as in examples 1.a, 1.b, and 1.c above. The recovery will be identical. In a few seconds the Consensus will notice that the processor has dropped out and processor recovery logic will be invoked. Since the processor is solidly broken the recovery will be unsuccessful, and the system will settle into a mode where every so often recovery is retried. Eventually a repairman will fix the processor, at which time recovery will proceed and the processor will rejoin Consensus. It is hard to predict whether the IMP system will momentarily go down because of the failure; experience indicates that it usually stays up, but our experience is limited to lightly loaded machines. (Severity 2-4)

Example 3. Suppose a power supply for a processor bus breaks. This is similar to the failing processor described above except that both processors on the bus are affected and the processors are given a hardware warning sufficiently far in advance that they can halt cleanly. The system will surely stay up through this failure. (Severity 1)

Example 4. Now consider a case in which some page of common memory ceases to answer when referenced. Each processor will get a hardware trap when it tries to use that memory, forcing it directly to the code which routinely verifies the environment. As a result, the failing memory will be deleted from the memory list by the Consensus and another module will be pressed into service to take its place.

If the failed page contained code, a spare copy will normally be available and a new spare copy will be made if possible. If it contained data, an unused page will be pressed into service. In either case, the system will be reinitialized, momentarily bringing the IMP system down. If the failed page contained the Consensus communication area, a new Consensus must be formed and thus recovery will take a little longer. (Severity 4)

Example 5. Let us now consider a failure of the PID. Suppose that the PID reports a task not previously set. When invoked, each strip checks to make sure that it is reasonable for the strip to be run. If not, another task is sought. Suppose instead that the PID "drops" a task. A special process periodically sets all PID flags independent of what needs to be done. This causes no harm, because

superfluous tasks will be ignored (as described above), and serves to pick up such dropped tasks. Thus we have both a consistency check on redundant information and a timer built into our use of the PID. If a PID fails solidly, another PID will be switched in to operate the system. Transient failures cause slowdown; switch-over may momentarily bring down the IMP system. (Severity 1, 4)

All of this leads to a slightly different image of the PID. Instead of being the central task disburser, with all processors relying on it to tell them what to do, the PID is a guide, suggesting to processors that if they look in a certain place, they will probably find some useful work to do. The system would in fact run without a PID, albeit much more slowly and inefficiently.

Example 6. Suppose a halt instruction somehow gets planted in common memory and that all processors execute it and stop. There is thus no Consensus left to come to the rescue. Furthermore, 60Hz interrupts are ineffective in a halted processor. After one second of inactivity, the bus arbiter timer will reset the processors, making them once more eligible for 60Hz interrupts which will restart them. Before the broken code is run, it will be checksummed, the discrepancy found, and a spare copy used. (Severity 2-4)

Example 7. Let us consider now what happens when, in common memory, an end test for a storing loop is destroyed, causing each processor to wipe out its 60Hz interrupt code in local memory. In this case not only are there no processors left to help, but the 60Hz interrupt will not help either, since the interrupt code itself is

broken. This is a case in which the machine is incapable of rescuing itself and will go off the network as a working node. When the Network Control Center notices that the IMP is no longer up, it will commence an external reload, restoring the IMP to operation. (Severity 5)

Example 8. Consider the case of a processor whose hardware is solidly broken such that it repeatedly stores a zero into a location in common memory. Mechanisms described above will repeatedly fix the changed location, but it is desirable to eliminate the continuing presence of this disrupting influence. The Consensus will notice that one of its number has dropped out and will endeavor to help the errant processor. After a few tries, a longer timer will run out, and the Consensus will take a more drastic action: final amputation. In this case there will be a rather lengthy IMP outage but the system will recover without external help. (Severity 4)

Example 9. One failure from which there is no recovery, either automatic or remote, is a program which impersonates normal behavior but is still somehow incorrect. That is, it holds off the right timers, has a valid checksum, and simulates enough normal behavior so that higher levels (e.g., the NCC) are satisfied. For example, if it were not for the fact that the NCC explicitly checks the version vumber of the program running in each IMP, a previous, compatible, but obsolete version of the program would exhibit this behavior. (Severity 6)

Example 10. Another class of failures which is hard to isolate and deal with is low-frequency intermittents. Consider the case of a single processor which is broken such that its indexed shift instruction performs incorrectly. Since this instruction only occurs in some infrequently executed procedures, the failure only manifests itself, on the average, once every period t. If t is large, for instance one year, then we can safely disregard the error, since its frequency is in the range of other failures over which we have no control. If t is small, say 100 milliseconds, then the Consensus will isolate the bad processor and excise it. At some intermediate frequency, however, the Consensus will fail to correlate successive failures and will instead treat each as a separate transient. The system will repeatedly fail and recover until some human intervenes. (Severity 6)

RESULTS AND CONCLUSIONS

Some strategies and techniques for building a reliable multiprocessor have been described above. We have, in fact, actually built and programmed such a machine using these strategies. We have found this machine straightforward to debug, both in hardware and software. Furthermore, the system continues to operate when individual power supplies are turned off, when memory locations are altered, when cables and cards are torn out, and through a variety of other failures. We have yet to establish field performance (which must be measured both in rate of recoverable

incidents and in rate of unrecoverable failures), but we expect to start gathering this information shortly.

We believe there are many important problems in the world today which could benefit from the principles described here. While we have discussed these principles in terms of a specific application (the IMP), most of the concepts are application independent. We have been able to separate the application code from the reliability subsystems intact in another application of the Pluribus machine.

ACKNOWLEDGEMENTS

Many people in addition to the authors have contributed to the ideas described herein, notably Benjamin Barker, John Robinson, David Walden, John McQuillan, and William Mann. In addition, there is a long list of those who helped to bring these machines into existence. Foremost among these are Martin Thrope, David Katsuki and Steven Jeske. The work reported here would not have been possible without the continued support of the ARPA/IPT office. Finally, a word of thanks to Robert Brooks and Julie Moore, who helped to prepare the manuscript.

REFERENCES

1. W. B. Riley, "Minicomputer Networks -- A Challenge to Maxicomputers?" Electronics, March 29, 1971, pp. 56-62

2. F. E. Heart, R. E. Kahn, S. M. Ornstein, W. R. Crowther, and D. C. Walden, "The Interface Message Processor for the ARPA Computer Network," AFIPS Conference Proceedings, Vol. 36, June 1970, pp. 551-567; also in Advances in Computer Communications, W. W.Chu (ed.), Artech House Inc., 1974, pp. 300-316

3. L. G. Roberts and B. D. Wessler, "Computer Network Development to Achieve Resource Sharing," AFIPS Conference Proceedings, Vol. 36, June 1970, pp. 543-549.

4. F. E. Heart, S. M. Ornstein, W. R. Crowther, and W. B. Barker, "A New Minicomputer/Multiprocessor for the ARPA Network," AFIPS Conference Proceedings, Vol. 42, June 1973, pp. 529-537; also in Selected Papers: International Advanced Study Institute, Computer Communication Networks, R. L. Grimsdale and F. F. Kuo (eds.) University of Sussex, Brighton, England, September 1973; also in Advances in Computer Communications, W. W. Chu (ed.), Artech House Inc., 1974, pp. 329-337.

5. S. M. Ornstein, W. B. Barker, R. D. Bressler, W. R. Crowther, F. E. Heart, M. F. Kraley, A. Michel, and M. J. Thrope, "The BBN Multiprocessor," Proceedings of the Seventh Annual Hawaii International Conference on System Sciences, Honolulu,

Hawaii, January 1974, Computer Nets Supplement, pp. 92-95.

6. W. R. Crowther, J. M. McQuillan, and D. C. Walden, "Reliability Issues in the ARPA Network, " Proceedings of the ACM/IEEE Third Data Communications Symposium, November 1973, pp. 159-160.

7. A. A. McKenzie, B. P. Cosell, J. M. McQuillan, and M. J. Thrope, "The Network Control Center for the ARPA Network," Proceedings of the First International Conference on Computer Communication, Washington, D.C., October 1972, pp. 185-191.

8. E. W. Dijkstra, "Cooperating Sequential Processes," in Programming Languages, ed. F. Genuys, Academic Press, London and New York 1968, pp. 43-112.

9. S. C. Butterfield, R. D. Rettberg, and D. C. Walden, "The Satellite IMP for the ARPA Network, " Proceedings of the Seventh Annual Hawaii International Conference on System Sciences, Honolulu, Hawaii, January 1974, Computer Nets Supplement, pp. 70-73.

10. A. L. Hopkins, Jr., "A Fault-Tolerant Information Processing Concept for Space Vehicles," IEEE Transactions on Computers, Volume C-20, Number 11, November 1971, pp. 1394-1403.

11. A. Avizienes, G. C. Gilley, F. P. Mathur, D. A. Rennels, I. A. Rohr, and D. K. Rubin, "The STAR (Self-Testing and Repairing) Computer: An Investigation of the Theory and

Practice of Fault-Tolerant Computer Design," IEEE Transactions on Computers, Volume C-20, Number 11, November 1971, pp. 1312-1321.

12. IBM Corporation, "OS Advanced Checkpoint/Restart," IBM Manual GC28-6708.

13. R. J. Gountanis and N. L. Viss, "A Method of Processor Selection for Interrupt Handling in a Multiprocessor System." Proceedings of the IEEE, Vol. 54, No. 12, December 1966, pp. 1812-1819.

14. L. Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem," Communication of the ACM, Volume 17, Number 8, August 1974, pp. 453-455.