AD-A008 842

INTERFACE MESSAGE PROCESSORS FOR THE ARPA COMPUTER NETWORK

BOLT BERANEK AND NEWMAN, INCORPORATED

PREPARED FOR
ADVANCED RESEARCH PROJECTS AGENCY

JANUARY 1975

UNCLASSIFIED
Security Classification

AD-A008842

DOCUMENT CONTROL DATA - R & D

Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Bolt Beranek and Newman Inc. 50 Moulton Street Cambridge, MA 02138	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED
	2b. GROUP

3. REPORT TITLE

QUARTERLY TECHNICAL REPORT NO. 8, INTERFACE MESSAGE PROCESSORS

4. DESCRIPTIVE NOTES (Type of report and, inclusive dates)

1 October 1974 to 31 December 1974

5. AUTHOR(S) (First name, middle initial, last name)

Bolt Beranek and Newman Inc.

6. REPORT DATE January 1975	7a. TOTAL NO. OF PAGES 38	7b. NO. OF REFS
8a. CONTRACT OR GRANT NO. F08606-73-C-0027	9a. ORIGINATOR'S REPORT NUMBER(S) Report No. 2988	
b. PROJECT NO. 2351		
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	
d.		

10. DISTRIBUTION STATEMENT

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Advanced Research Projects Agency Arlington, Virginia 22209

13. ABSTRACT

The ARPA computer network is a packet switching store-and-forward communications system designed for use by computers and computer terminals. This report concentrates on the new Pluribus IMP design; in particular on those aspects of the design which make a highly reliable system. Both the multiprocessor hardware and the software which operates on it include a large number of features designed to insure reliable operation; the design is applicable to a much broader set of uses than the IMP algorithm.

DD FORM 1 NOV 65 1473 (PAGE 1)
S/N 0101-807-6811

UNCLASSIFIED
Security Classification

UNCLASSIFIED
Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Computers and Communication						
Store and Forward Communication						
ARPA Computer Network						
Packets						
Packet-switching						
Interface Message Processor						
IMP						
Terminal IMP						
TIP						
High Speed Modular IMP						
Lockheed SUE						
Pluribus						
Fault-Tolerant Computing						
Reliability						
Redundancy						
Error-Recovery						

ia

DD FORM 1 NOV 65 1473 (BACK)
S/N 0101-807-6821

UNCLASSIFIED
Security Classification

A-31409

January 1975

INTERFACE MESSAGE PROCESSORS FOR THE ARPA COMPUTER NETWORK

QUARTERLY TECHNICAL REPORT NO. 8

1 October 1974 to 31 December 1974

Submitted to:

IMP Program Manager
Range Measurements Lab.
Building 981
Patrick Air Force Base
Cocoa Beach, Florida 32925

This research was supported by the Advanced Research Projects Agency of the Department of Defense and monitored by the Range Measurements Laboratory under Contract No. F08606-73-C-0027.

TABLE OF CONTENTS

1. OVERVIEW

This Quarterly Technical Report, Number 8, describes aspects of our work on the ARPA Computer Network under Contract No. F08606-73-C-0027 during the fourth quarter of 1974. (Work performed from 1969 through 1972 under Contract No. DAHC-69-C-0179 has been reported in an earlier series of Quarterly Technical Reports, numbered 1-16.)

During this quarter three new network nodes were installed. Interestingly, all three nodes were installed at sites which already possessed network nodes; the motivation for the new node equipment was the desire to attach more Host computers than can be supported by a single node. It is therefore noteworthy that the new Pluribus IMP, which is essentially ready for field installation, has the physical ability to handle many more Host connections than the current line of Honeywell-based IMPs. The new nodes this quarter include IMPs at the Seismic Data Analysis Center and Stanford Research Institute, and a TIP at the USC Information Sciences Institute. At the end of the quarter the IMP located at Case Western Reserve University was removed from the network.

Most of the implementation of Host access controls and algorithms for equal distribution of IMP resources among Hosts was completed during the fourth quarter. Due to difficult problems in the final target system, however, the complete implementation of these mechanisms was postponed until the first quarter of 1975. In addition, the IMP software was modified to permit, under controlled experimental conditions, Host use of packets for which message processing is not performed.

Early in the fourth quarter we began field testing of the TIP software which implemented the access control and user accounting mechanisms described in our Quarterly Technical Report No. 6. This testing revealed several unanticipated difficulties, and thus installation of this software did not begin until early December, about a month later than anticipated. By the end of the quarter the TIP software was installed in many operational TIPs, but some installations will be made in early 1975. The RSEXEC programs for maintaining the various access control and accounting data bases were operational by the end of the quarter.

As noted in our Quarterly Technical Report No. 7, by the end of the third quarter the two production copies of the 14-processor Pluribus IMP hardware (currently 13 processors plus a paper tape reader to simplify testing) were essentially ready for delivery, and the software nearly so. During the fourth quarter software debugging and tuning continued, and toward the end of the quarter we felt the system was ready for a field trial. Accordingly, one of the two production machines was "shipped" from our development area to the BBN Research Computer Center. This will provide an environment similar to field installation for testing during early 1975. Section 2 of this report describes in detail our approach to system reliability issues in the Pluribus IMP.

During the fourth quarter we published and distributed revisions to three operational documents, BBN Report No. 1822, *Specifications for the Interconnection of a Host and an IMP*, BBN Report No. 2183, *TIP User's Guide*, and BBN Report No. 2184, *TIP Hardware Manual*. In addition, four professional papers were prepared and submitted as follows: "The Evolution of Message Processing Techniques in the ARPA Network," by J. M. McQuillan, to appear in International Computer State of the Art Report No. 24: Network Systems and Software, Infotech, Maidenhead, England; "Pluribus -- A Reliable Multiprocessor," by S. M. Ornstein, W. R. Crowther, M. F. Kraley, R. D. Bressler, A. Michel, and F. E. Heart, submitted to the AFIPS 1975 National Computer Conference; "Issues in Packet-Switching Network Design," by W. R. Crowther, F. E. Heart, A. A. McKenzie, J. M. McQuillan, and D. C. Walden, submitted to the AFIPS 1975 National Computer Conference; and "Some Considerations for a High Performance Message-Based Interprocess Communication System," by J. M. McQuillan and D. C. Walden, to be presented at the ACM SIGCOMM-SIGOPS Interface Workshop on Interprocess Communications, March 24-25, 1975.

2. RELIABILITY ISSUES IN THE DESIGN OF THE PLURIBUS IMP

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. Usually the objectives have been to gain increased operating power through parallelism and/or to gain increased system reliability through redundancy.

In this section, we describe our approach to the objective of high system reliability in the multiprocessor IMP system which has been under development since 1972. Aspects of this new IMP have been described in several of our previous Quarterly Technical Reports (in particular, No. 4). To review, the design objectives for the machine are restated here.

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 developed to meet these goals has been named the Pluribus line. The machines are highly modular at several levels and have a minicomputer/multiprocessor architecture as described in the past. 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 which we describe here.

We will begin with a brief review of the structure of our system. We then discuss our reliability goals and introduce three broad strategies used to achieve these goals. Next the system reliability structure is described, and finally we present a number of examples which illustrate how the system works.

2.1 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 multi-processor control and reliability jobs, treating all jobs uniformly. We view the processors 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 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.

2.1.1 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 (see Figure 1) 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 processor 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. In a large configuration, about half of the machine consists of standard parts from the Lockheed SUE line; the remainder is of special design.

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. 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.

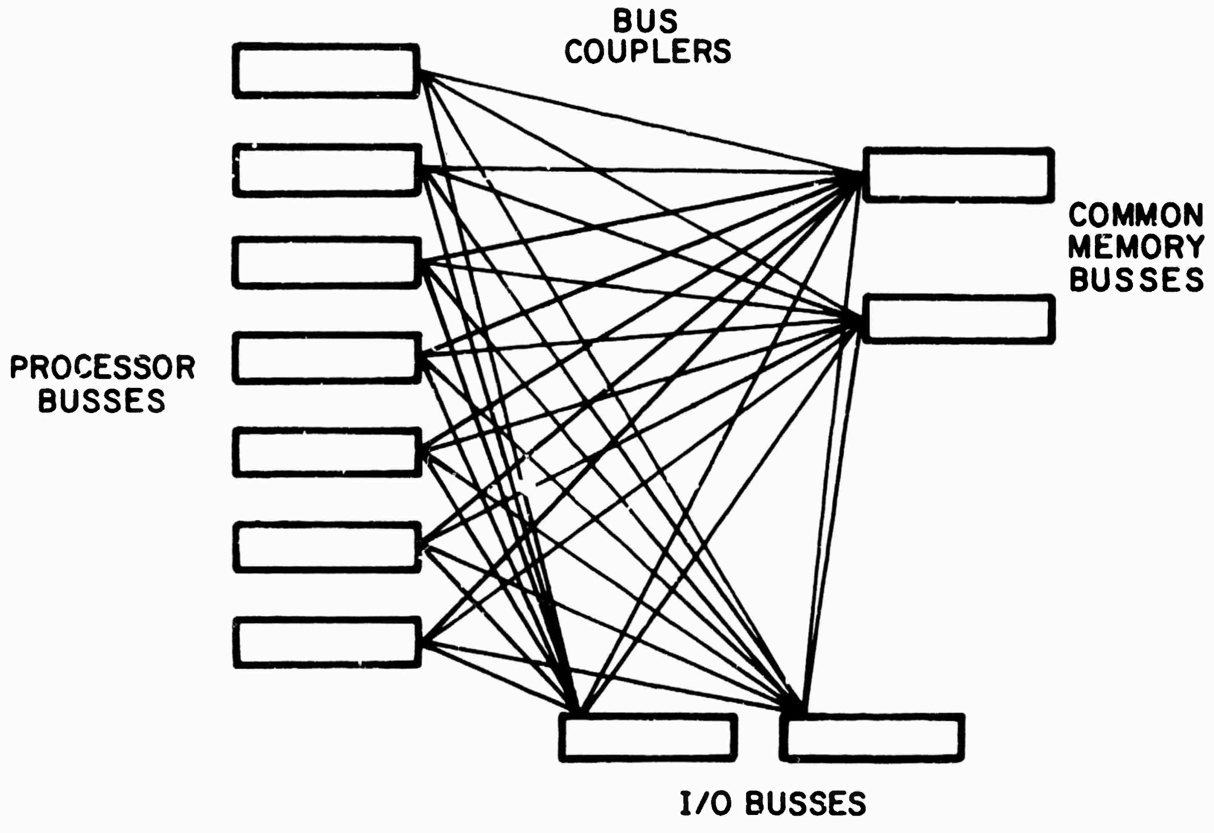

Figure 1: Pluribus IMP (14-processor system)

Bus Interconnection

2.1.2 Software Strurture

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, routing computations, reliability tasks, etc., can also be easily performed in parallel. Thus, we have been able to devote our attention to the problems of multiprocessor implementation rather than struggling to force our algorithm into a parallel form.

The structure that we have 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. We try to break the program 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 has higher priority 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 for it is very high. For efficiency 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 be longer than that.

An inherent part of multiprocessor operation is the locking of critical resources. This is the mechanism by which the algorithm enforces sequentiality when it is needed. Our system uses a load-and-clear operation as its primitive locking facility. 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.

2.1.3 Status

Two production Pluribus IMPs have been constructed and are running. Each contains 7 processor busses (generally, each holds 2 processors), two memory busses, and two I/O busses. These machines have been connected intermittently into the ARPA Network for testing purposes and one has now been "shipped" to the BBN Research Computer Center for testing under field conditions. A single Pluribus IMP has been running on the network for an extended period in order to validate performance during routine operation. Three Satellite IMP configurations are presently under construction.

2.2 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; 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 absolutely imperative to operate continuously and perfectly (as it may be in a space vehicle guidance computer); it suffices to operate correctly most of the t me 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. The IMP system happens to be embedded in a larger system which is quite forgiving (not an uncommon situation). Thus infrequent 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 unsable 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 before the code is put into the field. On the other hand, we do assume that all manner of perverse and infrequent bugs will forever be latent within the code. Thus, we must be able to survive even when all code, data structures, etc., have been randomly destroyed.

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 reconfigured 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 continues to run. Finally, the system should absorb repaired or newly introduced parts gracefully.

2.3 Strategies

In order to understand our system it is convenient to think of the strategies used to achieve our goals as divided into two parts; these more or less parallel the traditional division of computer system 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" implies 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.

2.3.1 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 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. We have thus avoided the vulnerable centralized switch generally found at the heart of a multiprocessor. There are multiple racks, each with its own power cord (to separate power sources if desired), and busses are assigned to racks in such a way that the loss of even an entire rack does not remove any unique central system resource.

With respect to non-central resources, such as individual I/O interfaces, one has more freedom. When a particular line 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, however, the line need be only singly connected.

In order for the system to adapt to different hardware configurations, the software must be able to determine, without human aid, what hardware resources are available to it. We have made it convenient to search for and locate those resources which are present and to determine the type and parameters of those which are found.

Since we must allow for active failures, there must be a mechanism which allows the program to turn off or isolate a unit which is acting in a malevolent manner. To arrange for this, 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 connected to that bus. These switches are protected from capricious use by requiring a password. Naturally an amputated processor has no access to these switches.

A final rule that we have followed in maintaining adequate isolation is to prohibit any common signal that would connect to all busses. Normally a common reset line is considered essential in any hardware system; however, we have avoided a common reset line since a single failure on its driver could render the entire system inoperative. In our system there is 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.

2.3.2 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.

2.3.2.1 Simplicity

It is always good to keep a system simple, for then one has a better chance of making 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 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 mechanism (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 multiprocessor 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 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. For example, there is a subsystem that, among other things, determines which common memory pages are available for message buffers. 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.

2.3.2.2 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 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.

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. (One can easily deduce from the state of our buffers what processing they need.)
- A device on the bus requesting a bus cycle, and a flip-flop in the bus arbiter capturing the fact of the request.
- One processor seeing a memory word at a particular system address and another seeing the same word at the same address.
- The PID level which a particular device uses to request service and the device which the software services in response to that level.

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.

We can now see how these methods are applied to our four examples:

- If the buffer and queue are inconsistent, the buffer will not be processed. Each buffer has its own timer; if the buffer is not processed in a reasonable time, it will be replaced on the queue.
- If the bus arbiter and devices disagree, the bus may hang. We have added a timer which times out bus transactions and does a bus reset after one second of complete inactivity.
- The software verifies that all processors see the same memory at a given address and when they do not, decides whether to declare the memory or one of the processors unusable.
- An inconsistency between the PID level that an interface is using and the software response to that level cannot persist because a process periodically forces the tables driving the response to agree with the hardware. (The PID level(s) used by each device are program-readable.)

2.3.2.3 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 reset 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.

2.4 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 2 and Table 1 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 2) serves three functions: first, it checksums all low level code (including itself); second, it insures that control is operating properly, i.e., that all subsystems are receiving a share of the processor's 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.

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 IMP algorithm 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, but the

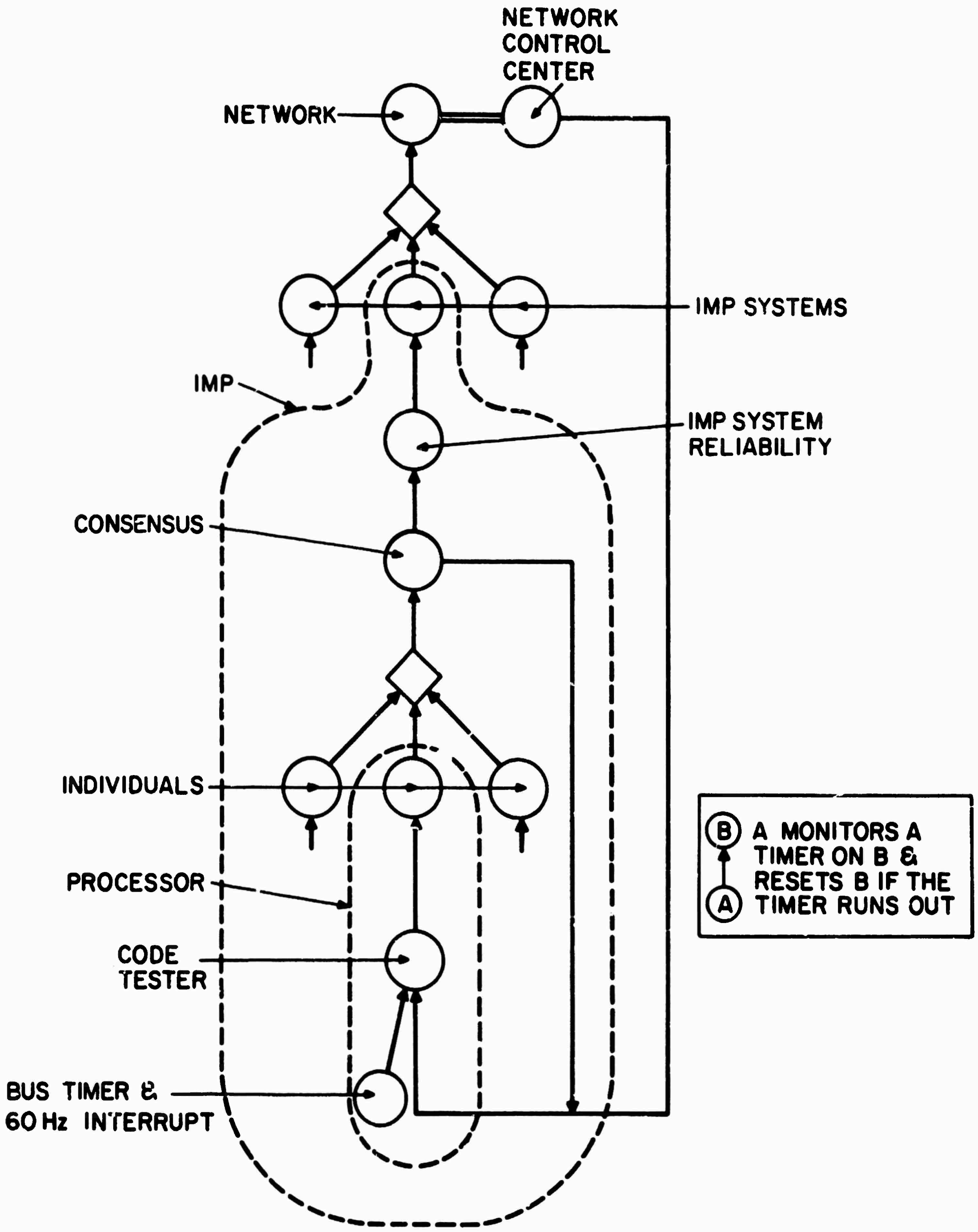

Figure 2: Pluribus Subsystem Monitoring Structure

Table 1: 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.

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 always 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 2. 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 therefore 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, for example 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 and technical staff for successful operation.

2.5 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. The severity of the direct consequences of these examples is rated on the following scale, from least severe to most severe:

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 delivering 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 reset 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. (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)

<u>Example 3.</u> 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)

<u>Example 4.</u> 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; switchover 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 number 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)

2.6 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 are now beginning to collect this information.

While we have discussed principles of Pluribus reliability 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 and, in fact, we are using the reliability subsystems intact in another application of the Pluribus machine.