PROGRESS REPORT ON THE DISTRIBUTED COMPUTING SYSTEM

JANUARY, 1972

THIS WORK HAS BEEN SUPPORTED BY THE NATIONAL SCIENCE FOUNDATION UNDER GRANT GJ - 1045

DEPARTMENT OF INFORMATION & COMPUTER SCIENCE

UNIVERSITY OF CALIFORNIA, IRVINE

IRVINE, CALIFORNIA 92664

PROGRESS REPORT ON THE DISTRIBUTED COMPUTING SYSTEM

JANUARY 1972

The progress that has been made on the design of the DCS since July 1971, has been sufficiently promising that we want to extend our plans for constructing a prototype system. The work to date has been primarily concerned with the design of a prototype system including the operating system kernel, communication protocols, the ring interface, and the file system. An important next step in the development of this design is the construction of a prototype system. The prototype will serve as a vehicle to study the design and demonstrate the use of these concepts.

A BRIEF FUNCTIONAL DESCRIPTION OF THE PROTOTYPE

The prototype will be capable of providing general purpose interactive computing. To keep the size of the effort within reasonable bounds only a single application will be provided initially. The supervisory programs and implementation will be kept general so that additional functions can be added at a later time. This approach will allow us to experiment with a general purpose system while still keeping the amount of programming less than would be required for a general purpose.

The system will be composed of (1) general purpose supervisory routines and (2) an applications program

Initially all of the supervisory routines will be provided. However, some of these will be somewhat abreviated in the initial implementation. As stated previously a single application will be provided.

This organization will permit the addition of new applications without changes to the supervisory routines. The application to be written will be a simple manuscript text editing system. The capabilities of the editor will be similar to the IBM Administration Terminal System (ATS) and will facilitate entering, correcting, and printing documents such as form letters, reports, and manuscripts.

The text editing application is appropriate for a number of reasons. The requirements of such a system typify many of the problems which the DCS is designed to solve. In particular an administrative text editing system should be available whenever needed. Down time is a nuisance in the office environment. To be acceptable for practical use such a system must use a low cost technology. It is also desirable to be able to easily make incremental changes in the size and capacity of such a system.

The text editing system is also appropriate because it requires to some extent most of the facilities required for any application. These include a very reliable file system, as well as the usual requirements for main memory and central processor resources.

The logic of a text editor is relatively simple and it will be possible to implement without too much difficulty.

Another characteristic which is important to the modeling and measurement aspects of the project is that the way in which the application is used is relatively constant. As a result the demand for system resources does not vary too much. This is in contrast to an application such as BASIC where the resources required vary greatly with different users and at different times.

ANATOMY OF THE PROTOTYPE

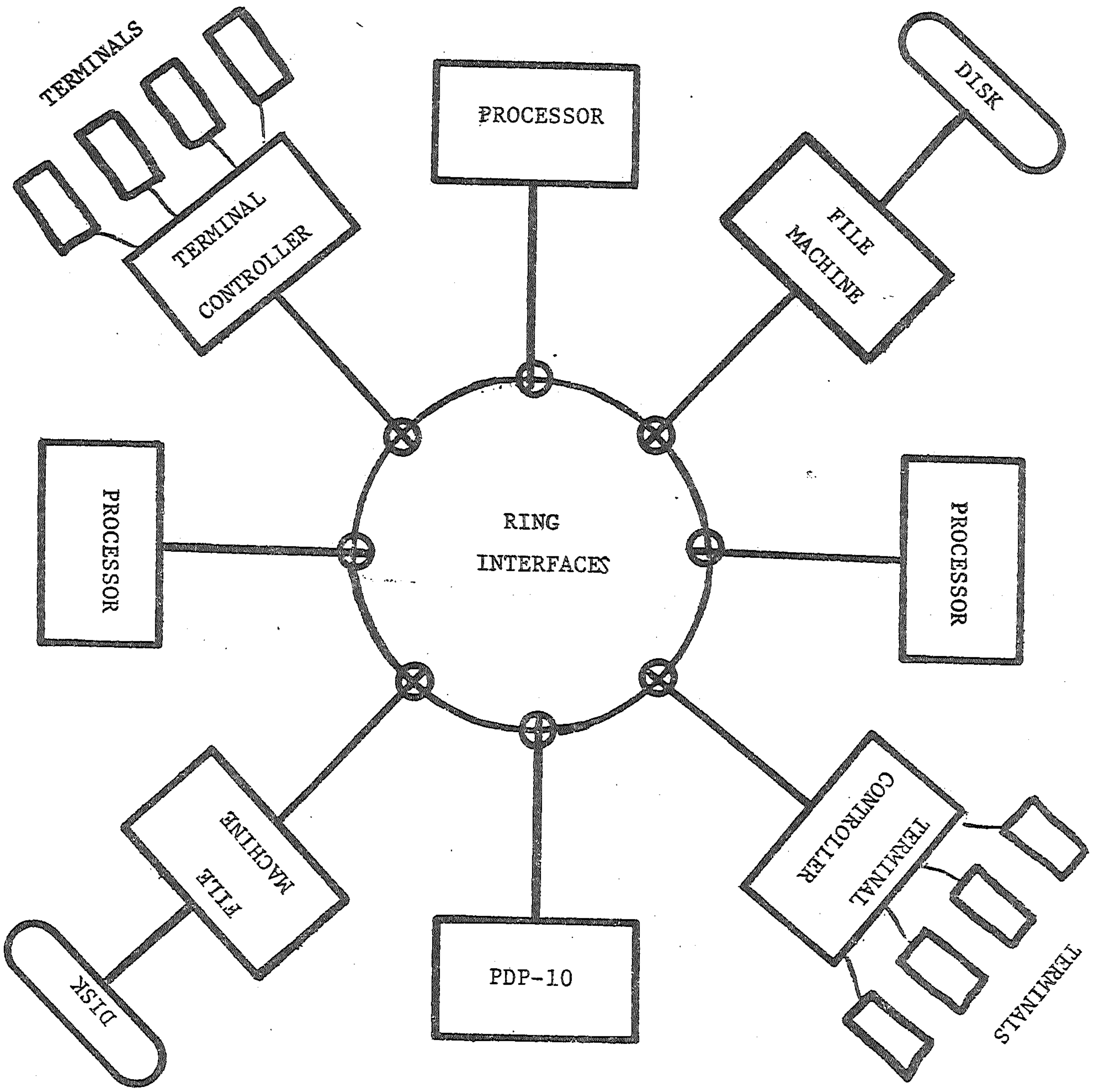
The distributed computer system approach is based on the following principles: (1) resource sharing, (2) redundancy, (3) compartmentalization, (4) simple structures.

The DCS system will provide mechanisms for allocating the work to be done among all functioning components. Redundant components will not be treated as standby units in the usual parallel redundancy sense. Instead all components will be allocated a portion of the work. When a component fails it will be leiminated from the pool of available components. The total capacity of system will be reduced, but since other components of this type exist, all functions will continue to be available. Compartmentalization will facilitate the logical exclusion of non-functioning components. This is analogous to the independent compartments on a large ship that facilitate sealing of leaking compartments so that the ship continues to float. The simple structures make it possible to design the above principles into an easily implementable framework.

The system will provide processors, file machines, and terminal controllers connected in a ring. Each of these components will be connected through a ring interface to a undirectional circular communications ring. In addition a PDP-10 will be connected to the ring through a ring interface to facilitate testing of the system.

The prototype system will consist of the following hardware units: (1) minicomputers, (2) ring interfaces, and (3) interconnecting cable.

Three of the minicomputers will share the processing load. All computing except that performed in the file machines and by the terminal controllers will be performed by these machines. If one fails, the other two will continue to provide service. Two minicomputers with disks will serve as file machines. Two other minicomputers will be terminal controllers for the connection of terminals to the system. The minicomputers will be purchased.


TERMINALS
TERMINAL CONTROLLER
PROCESSOR
DISK
FILE MACHINE
PROCESSOR
RING INTERFACES
PROCESSOR
FILE MACHINE
DISK
PDP-10
TERMINAL CONTROLLER
TERMINALS

The ring interfaces will provide for the connection of the minicomputers to the communication ring. They will be designed as general purpose ring interfaces which can be customized slightly through microprogramming to accomodate the particular component to be connected. A brief decription of the ring interface is included in Appendix G. Design is currently in progress. Construction will be either at UCI or an outside facility.

The interconnecting cable will be purchased.

The principal software parts are (1) the operating system, (2) the file system, and (3) the editor program. Preliminary descriptions of the operating system are in UCI Technical Report 11 (included as Appendix E). An overview of the file system is included as Appendix F. The text editing program is envisioned as similar to IBM's ATS. All programming will be done by the DCS staff. The campus PDP-10 will be used to prepare programs in the development of the system.

PROGRESS ON THE INITIAL GRANT

The work to date has principally been to determine the scope of the project and to design at a high level basic a system which would achieve the goals set forth. This work falls into five general areas:

1) Specification of communications protocol for the ring
2) Design of an operating system which resides at each node and is responsible for resource allocation, scheduling and fail-soft
3) Design of a distributed file system which also exhibits the fail-soft characteristics
4) Specifications of the form and scope of a prototype which will be complex enough to yield realistic measurements
5) Preliminary design of hardware for a ring interface

6) Development of a simulation model

Significant progress has been made in all of these areas. Appendix E (UCI Tech Report 11) describes work done on the communications protocol and operating system. Since UCI Tech Report 11 was published, the operating system has been specified and flowcharted in much greater detail. The current design of the distributed file system is included in Appendix F. Design of the ring interface has proceeded in pace with the development of simulation model and preliminary hardware design.

FUTURE ANALYSIS OF THE PROTOTYPE

The construction of a prototype DCS system will result in more than just a working system. Prior to, concurrent with, and after the construction of the system careful attention will be paid to the task of gathering quantitative and qualitative information. There are a number of areas to be examined in detail. These bear on such issues as:

1) the load vs response behavior
2) the reaction of the DCS to induced errors in the transmission system
3) the reaction of the DCS to induced faults and failures of the processors, file devices, and software
4) the incremental costs incurred by the distribution of control in the DCS. This includes where and how much time is spent.

We intend to also develope a reasonably complete simulation model of the prototype system. We will use the prototype to verify the simulation model and then use both the model and the protoype to explore the tuning and behavior of the system.

There are a number of other goals which will be achieved by the prototype construction. There is of course, the verification of the idea. There are also such items as:

1) What does it really cost to program? We claim advantages inherent in the DCS design. We will keep records of costs and problems.

2) Verification of the modularity goal. Can we maintain the inherent modularity of the DCS design through an implementation?

3) Verification of the fail-soft design goals -- Can we really achieve this end?

4) The problems that occur when we add a new type machine. What are the cost and problems?

We expect that our prototype construction exercise will enable us to make reasonably accurate estimates of the cost of a production DCS.

RESULTS

We will produce a series of reports on the following subjects:

1) distribution effectiveness

2) a ring interface design

3) a T-node design

4) a terminal multiplexer design

5) design of a distributed computer software system

APPENDIXES

APPENDIX A

SCHEDULE

<u>Major Tasks</u>

1. Ring interface construction
2. Supervisor program development
3. File system development
4. Text-edit program development
5. System assembly
6. System integration
7. Experimentation
8. Analysis of progress and results
9. Study of new and related topics

<u>Task Steps</u>

1. Ring interface construction
    a. Preliminary design
    b. Selection of components
    c. Detailed design
    d. Procurement
    e. Test specifications
    f. Construction
    g. Test construction
    h. Preliminary tests
    i. Test
    j. Rework
    k. Final Test

2. Supervisor program development
    a. Preliminary design
    b. Specification of functional modules
    c. Simulation of activity
    d. Detailed design of modules
    e. Design of tests
    f. Coding of modules
    g. Coding of tests
    h. Test of modules
    i. Test of supervisor

3. File system development
    a. Preliminary design
    b. Specification of functional modules
    c. Simulation of activity
    d. Detailed design of modules
    e. Design of tests
    f. Coding of modules
    g. Test of modules
    h. Test of supervisor

4. Test-edit program development
    a. Preliminary design
    b. Specification of functional modules
    c. Simulation of activity
    d. Detailed design of modules
    e. Design of tests
    f. Coding of modules

   g. Test of modules

   h. Test of editor

5. System assembly

   a. Detailed layout

   b. Materials procurement

   c. Interconnection

   d. Test of interconnection

6. System integration

   a. Plan for integration

   b. Test sub-unit relation

   c. Rework

   d. Test complete system

   e. Rework

   f. Final test

7. Experimentation

   a. Design of models

   b. Analysis of models

   c. Design of experiments on models

   d. Simulation experiments

   e. Design of experiments on real system

   f. Experiments on real system

8. Analysis of progress and results

   a. Analysis of progress

   b. Analysis of simulation results

   c. Analysis of experimental results

d. Ideas for further study

e. Publication

9. Study of new and related topics

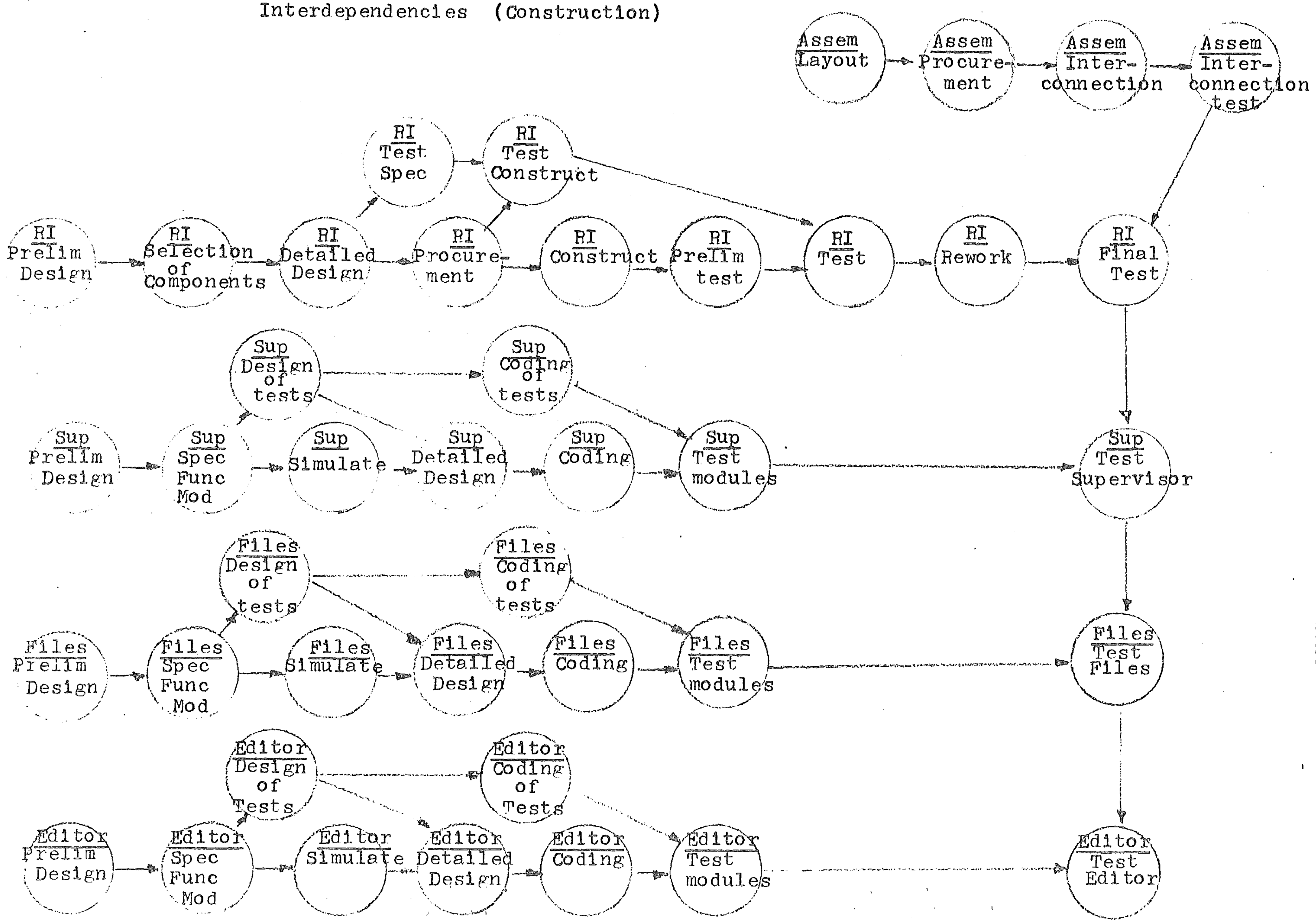
   a. Identification of topics

   b. Analysis of topics

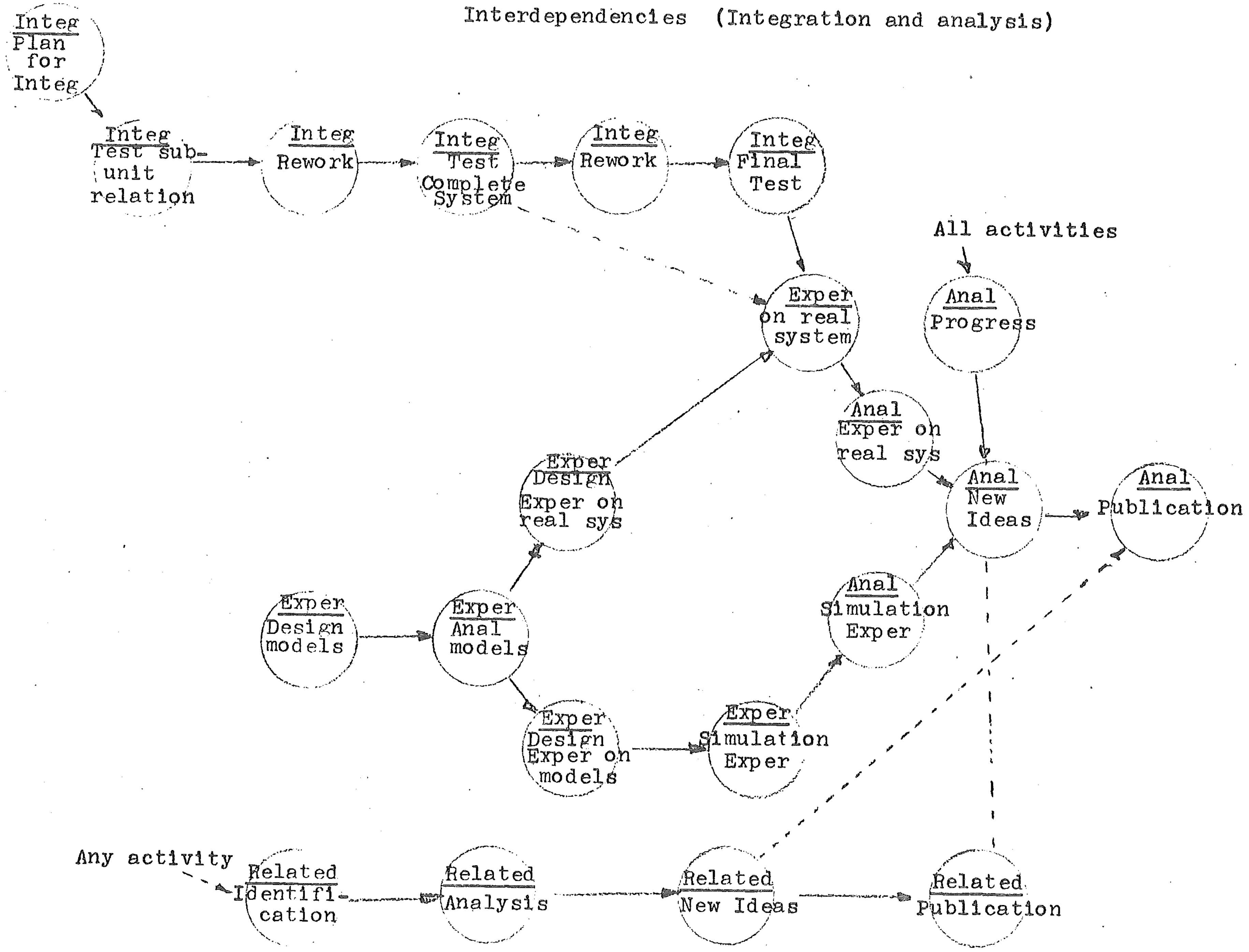
   c. Ideas for further study

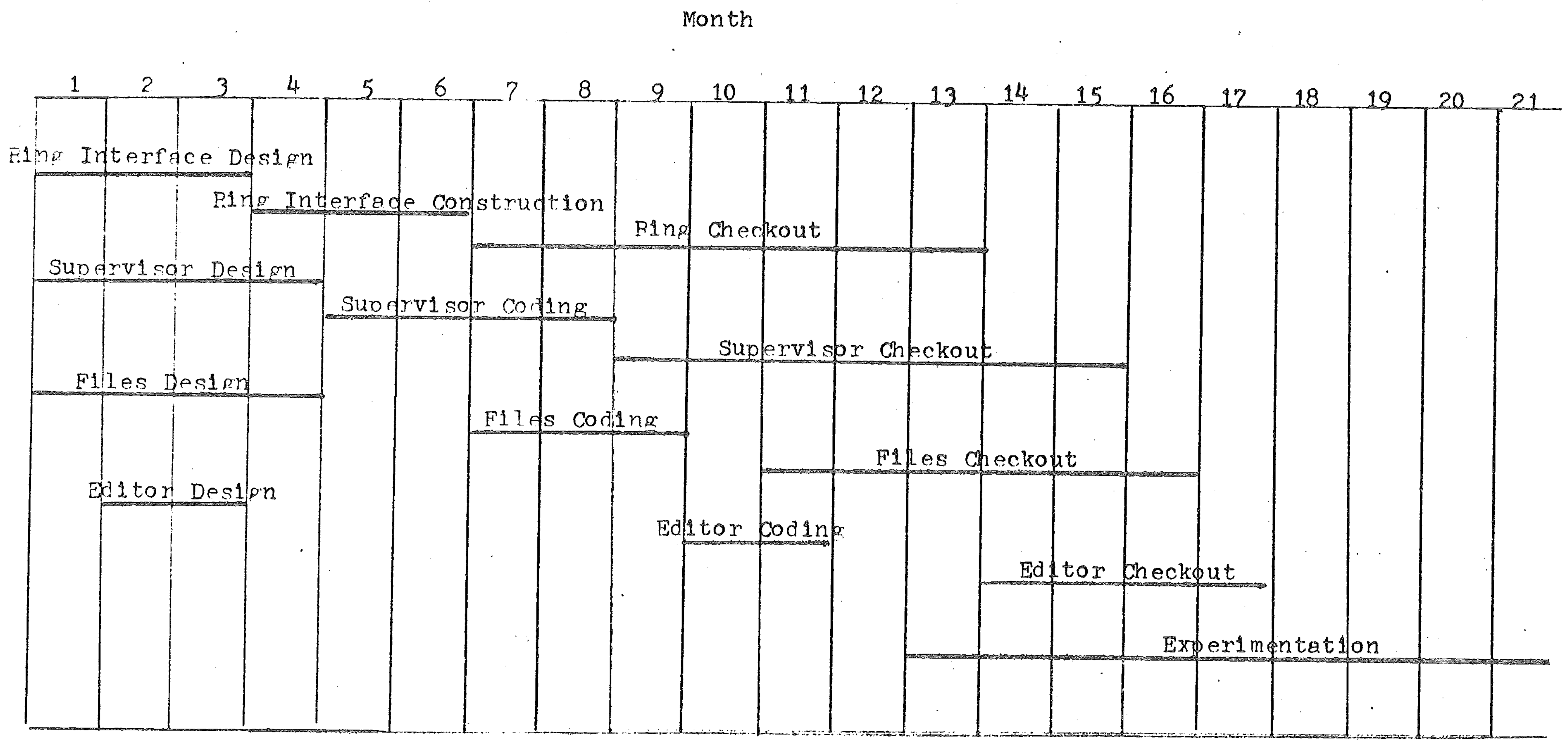
   d. Publication

Interdependencies (Construction)



Interdependencies (Integration and analysis)




Month
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
Ring Interface Design
Ring Interface Construction
Ring Checkout
Supervisor Design
Supervisor Coding
Supervisor Checkout
Files Design
Files Coding
Files Checkout
Editor Design
Editor Coding
Editor Checkout
Experimentation

Schedule Summary

1. Design (4 months)
2. Construction and coding (6 months)
3. Checkout and Rework (6 months)
4. Experimentation (6 months)

APPENDIX B

BUDGET INFORMATION NOT INCLUDED

APPENDIX C

HARDWARE BUDGET

MATERIALS LIST FOR A PROPOSED DCS SYSTEM

3 Processors	@$13K	$ 39K
2 File Machines		
CPU	5K	
Disk	15K	
	20K	40K
2 Terminal controllers	8K	16K
8 Ring Interfaces	1.5K	12K
		107K
Contingency		10K
		117K

(Additional requirements include rented terminals and purchase of PDP-10 time from campus facility) .

RING INTERFACE MATERIALS

Control memory (8bits x 512 words=4K bits 4K x 10¢/bit=$400)	$400
Buffers (8 buffers x $20/buffer)	160
Associative stores (15 x $20/name)	300
Line drivers and receivers for ring (8 chips x $5/chip)	40
Ring relays and connectors	50
Control logic (12 chips x $4/chip)	50
Interpreter (20 chips x $5/chip)	100
Drivers and receivers for CPU	50
Power supply (for MOS and TTL)	100
Hardware (sockets, boards, etc.)	200
Spares	50
	$1500

APPENDIX D

THE STRUCTURE OF A DISTRIBUTED COMPUTER SYSTEM -- COMMUNICATIONS

Dave J. Farber
Kenneth Larson

Department of Information & Computer Science
University of California, Irvine
Irvine, California

ABSTRACT

The distributed computing system (DCS) is an experimental computer network under study at the University of California at Irvine under NSF funding. The network has been designed with the following goals in mind: reliability, low cost facilities, easy addition of new processing services, modest startup cost, and low incremental expansion cost. The structure chosen to achieve these goals is a digital communications ring using T1 technology and fixed message lengths. The computers used are small to medium scale and are interfaced to the ring using a fairly sophisticated piece of hardware called a ring interface (RI).

There are two features which make the communications protocols unique. First, messages are addressed to processes, not processors. This is accomplished by placing an associative store in each RI. The store contains the names of all processes active on the attached processor When a message arrives over the ring, the destination process name is matched against the associative store. If a match occurs the message is copied and passed over the ring to the next RI. Second, messages are only removed at the RI from which they originate. The ring may be thought of as a series of message slots. To transmit a message the RI waits for an empty slot and places the message on the ring. The message is copied

when necessary as it proceeds around the ring and checked against the original when it returns to the originating RI where it is removed from the ring. If errors are detected or the message fails to return in a specific amount of time the message is retransmitted. The retransmission causes problems since RI's may receive multiple copies of the message. The paper describes a scheme for sequencing messages which removes these problems. Note that this scheme allows messages to be broadcast to all processes or a class of processes. The DCS/OS software uses this feature extensively. The paper also discusses the error detection and maintanence features. Basically, each RI has a "short circuit" which removes it from the ring while maintaining the ring connectivity. This "short circuit" can be activated through internal checks within the RI or externally by specific messages. Redundancy communication paths in the ring protects the ring connectivity. A method for connecting such rings using an inter-ring interface processor is described. The scheme perserves the above protocols on the constituent rings and extends the notion to inter-ring operation. The paper also describes the hardware innovations used in the implementation of this system, some design ideas, and expected costs.

THE STRUCTURE OF A DISTRIBUTED COMPUTING SYSTEM - SOFTWARE

Dave J. Farber
Kenneth Larson

Department of Information & Computer Science
University of California, Irvine
Irvine, California

ABSTRACT

The distributed computer system operating system (DCS/OS)is an operating system designed to control a set of processors interfaced to a shared communications ring. Some features of the ring which greatly affect the design of DCS/OS are:

1) Messages are addressed to processes and not processors; routing is done by hardware.

2) Control of the ring does not reside in one central processor but must be spread out over the processors on the ring.

3) A message may be addressed to all processes or a class of processes as well as a specific process, so messages may be "broadcast".

Certain design goals set for the DCS/OS have forced it to be a unique multiprocessing system. First, there is no central processor which controls scheduling and resource allocation. These functions are distributed among the various processors. Second, the system is insensitive to the failure or malfunctioning of any processor on the ring, provided the hardware can maintain the connectivity of the ring. Third, in the case of a small processor, it is possible to distribute the processes controlling the processor among other distinct processors. Fourth, DCS/OS has a modular, hierarchial structure which allows it to be streamlined for

specific applications without making major modifications to the entire system and which allows the use of pre-existant software.

To achieve the above attributed we have divided DCS/OS into autonomous operating systems (AOS) one of which is associated with each processor on the communications ring. The structure of each AOS is hierarchial. Level Ø is a round robin scheduler which schedules monitor routines when necessary. Level 1 is the software required for interprocess communication which is described in detail in the paper. The communications scheme used relies on messages and process addressing to allow two processes to communicate with each other in a uniform manner whether they are on the same processor or distinct processors on the ring. This allows the distribution of an AOS. Level 2 comprises the routines necessary to check the ring and the processors on the ring for malfunctions. The basic strategy is to assume a processor is malfunctioning only if two other processors independently determine it is so. Level 3 comprises the monitor routines. Principally the paper deals with the resource allocator. DCS/OS uses a bidding scheme to achieve resource allocation and scheduling. When a process wishe to spawn a process it sends out an RFQ to all resource allocators, each of which estimates a "cost" of creating the process on the associated processor. Some return bids which the spawning process uses to determine with which to sign a contract for the creation of the process. This procedure is described at length in the paper. Level 4 is the service and user programs.

APPENDIX E

THE SYSTEM ARCHITECTURE OF THE DISTRIBUTED COMPUTER SYSTEM - AN INFORMAL DESCRIPTION

DAVID J. FARBER & KENNETH C. LARSON

UNIVERSITY OF CALIFORNIA, IRVINE

TECHNICAL REPORT NO. 11

SEPTEMBER 1971

THIS WORK HAS BEEN SUPPORTED BY THE NATIONAL

SCIENCE FOUNDATION UNDER GRANT GJ - 1045

I. INTRODUCTION

THE REQUIREMENTS OF A DISTRIBUTED COMPUTER SYSTEM

Our desire to improve computing services in our own environment has interested us in the design of a computing system which will have the following characteristics: reliability, low initial cost, incremental expansion capability, variety of language systems, and modest system programming requirements.

We do not believe in absolute reliability. Rather, we seek to provide a system that has a high probability of responding to a high percentage of requests for services. Inevitably, some user will be affected if a component fails; however, most users should get most of what they ask for most of the time. While this requirement is loosely stated, it does imply a system that is invulnerable to partial failure.

Low initial cost means simply that the minimal configuration have a modest cost, say $250,000. This criterion excludes most of the currently available medium scale systems, e.g., IBM 360/50, XDS SIGMA 7, and DEC PDP-10. As the needs of the users increase, and as additional funds become available, incremental expansion of the system should be possible without disturbing the system. The ability of the system to fruitfully use small additions of equipment is also desirable.

Most communities of users and especially university users require a variety of languages for teaching and research. In some cases it is easier to buy a computer for the language it has on it, than to implement the language processor on another machine.

A major cost of extending and maintaining a computer system is system programming. We hope a modular, regularized system design can do much to reduce costs which usually occur when a new component or service is added to an existing system.

We believe that small machines dedicated to servicing one type of task can provide economical service to a large percentage of our users. They can be customized by programming and/or microprogramming for increased efficiency, because the machines will be dedicated to providing a single type of service. The overhead caused by doing different tasks can be eliminated. In short, small processors can be very cost effective for those jobs they can handle. This approach is directly contrary to the common one in which the choice of a machine is dictated by the largest problem to be solved. Then the hardware resources are often shared (uneconomically) by users having modest requirements. We would send the few jobs that are at the upper tail of the distribution of hardware requirements to another computing system, so that we could contend with the vast majority of computing requirements. In an ultimate version of the system to be discussed here, large jobs could be handled, but it will be less expensive to experiment with small machines. Perhaps more important a system of small machines can have the features we desire and also handle a large percentage of the computing tasks in an environment such as ours.

We also believe that we can obtain improved system performance by connecting a set of small machines. Linking the small machines will permit the sharing of a pool of peripheral facilities. Interconnecting the small machines will also improve the reliability of the system. Should a single processor fail, the user could be shifted to an alternate processor.

Models for interconnecting processors can in general be described by the topology of the net. In a rich topology each processor could be connected to every other processor via a separate communications path. As the number of processors becomes significant and spread out geographically, the cost of this approach becomes high. One solution to this problem is to connect the processors through a switching center. All traffic first goes through this center and then out. The switching center is a critical and vulnerable component. As the number of processors grows, the bandwidth of the center becomes a troublesome and limiting problem. We may in turn solve these problems by a scheme utilizing digital multiplexing. The basic feature of this approach is to pass a common broadband digital transmission system through each processor. Thus each processor has a physical connection with only two other processors. One can visualize the transmission system as a ring with processors at points on the circumference. By sending a message around the ring, a processor can talk to any other processor. The number of processors placed on this system is limited by the bandwidth of the transmission system. (higher than that of a switching center) and the addressing limitations of the hardware.

Having eliminated central switching centers we find ourselves with no central authority to look after the condition of the communications system Rather than designate one of the processors as the only one capable of performing this task, (if we did we would suffer from reliability problems) we let this control reside in any processor which is free to do this task. We have likewise distributed the control of other aspects of the system and its resources.

We have roughly sketched a system consisting of a set of different processors interconnected by a common transmission system. The complete control of the system is distributed in time and space as needs arise and facilities are available. The system we have sketched we call a Distributed Computer System (DCS).

II. COMMUNICATIONS

THE COMMUNICATIONS PHILOSOPHY

We would like to take a different point of view of the communications system than is normally taken in computer networks. We believe that this new view will ultimately yield a clearer image of the problem and thus will yield a more extendable base.

This point of view is that all addressing of messages which are placed on the ring be in terms of processes names; not processors. The only addresses (at least the only pertinent addresses) are the names of the issuing process and desired destination process. We propose an encoding of the process addresses such that they contain the following extractable information:

Name of general class of process

Enumeration of particular subclasses within this case

Serialization of an instance of a particular member.

In particular, the processes are identified by a general classification such as control, language, file, etc. Within each of these classes are subcategories such as FORTRAN, PL/1, BASIC, etc. In addition, since there may be a number of instances of a subcategory, there is an enumeration of the subcategory that is 1,2, etc.

The communications system is designed to deliver these process-oriented messages to their appropriate destinations. Many of the messages which will be present on the ring will be multi-destination messages, that is, they will be addressed to a class of processes, rather than to a particular one. Given this plus the inherently more time-consumming task of recognizing a process address rather than a prewired equipment address, we propose to design the system so that it behaves in a broadcast mode of operation. By this we mean that a message will circulate around the ring until it is received by the node which sent it. At that point it will be absorbed and removed from the communications system. More will be said on the operation and implications of such a system later in this document.

WHY THE CHANGE OF METHOD

We will give some of the reasons which lead us to propose this new approach. The most attractive feature of this approach is that it allows a uniform conceptual point of view. The processor oriented view required a rather continual translation from process name to the processor that was supplying the service. This continual translation was required for reliability and flexibility. Also, it was not clear that it was easy to supply multi-user processors with the old scheme. An advantage of this method of addressing is the easier and more dynamic entry and exit protocols available to processors on the ring. This new approach also allows a type of scheduling which we have referred to as the RFQ (Request for Quotation) method.

COMMUNICATIONS SYSTEM PROTOCOL

This section deals with the philosophy and operating protocol of the communications system used in the Distributed Computer System. The architecture used is that of a data ring, that is all nodes of the system are connected together via a uniderectional single data path. Thus when any node wishes to communicate with some other node it can do so without the knowledge of either the location of the node or the connectivity of the path

We shall use a fixed block length message format, that is, the ring is is analogous to a "lazy susan" with dishes on it that are rotating past a set of people (the nodes) gathered around it. To continue the analogy, suppose a process, P.1 desired to send a message to one named P.2 The principal that we will use is as follows:

1) P.1 waits until an empty tray appears.

2) When it finds an empty one, it places two items in the tray, The message and the name of the process it wishes to send it to.

3) It then waits until it sees the message passing it again. At that point it removes the message from the tray and checks to see if process P.2. received the message. If not it repeats the sequence. From P.2's point of view it looks for messages with its name on them as the dishes pass it. If it sees one it makes a copy of the message and drops in the tray a note telling the sending process that it has received the message. Note that P.1 does not know either where P.2 is or for that matter whether or not P.2 stays put.

Let us now look at the proposed actual system. There are three critical parts of the system:

1) The idle slot detector and transmitting-node-detector.

2) The terminating process recognizer.

3) the node controller.

The message itself is composed of the following fields along with their approximate sizes:

1) The originating physical node number (ON) (9 bits) parity checked.

2) The terminating process name (TPN) (16).
3) The originating process (OPN) (16).
4) Header check bit-parity (1).
5) Serial field (SF) (1).
6) Message Definition Field (MDF) (8).
7) Messager portion (N1000 bits).
8) Matched bit (MB) (1).
9) Accepted bit (AB) (1).
10) Ring Check (1).
11) Whole message check (N error detecting).

The idle slot detector has the following function in the operation of the ring. When an attached processor (host) desires to transmit a message, it loads one of a set of buffers in its ring node. The node then monitors the ring looking for a message slot with an originating physical node number equal to zero, indicating an idle slot. It then places the message from the transmission buffer into the idle slot. The node circuitry then waits until it detects a message entering its transmitting-node-detector. If the originating physical node number in the message is that of the node, the node assumes that this is the message it last transmitted. Note this technique demands that a node transmit one and only one message at a time. If a host demands a broader bandwidth than can be supported by this technique, the section of the node which does the transmission and detection can be duplicated and assigned a separate node number. The node controller and terminating process recognizer, however, would not be duplicated.

When the transmitting-node-detector detects its node number in an originating node number slot with proper parity, the rest of the message is checked; if it passes, the accepted bit is checked. This bit is set by any node which accepts a message as addressed to it. The message is then considered successfully transmitted and the host is thus notified. If the parity check of the originating node number is successful, the message slot is marked idle. If the parity check failed the attached host is notified of a ring error, but the node continues to wait for the transmitted message.

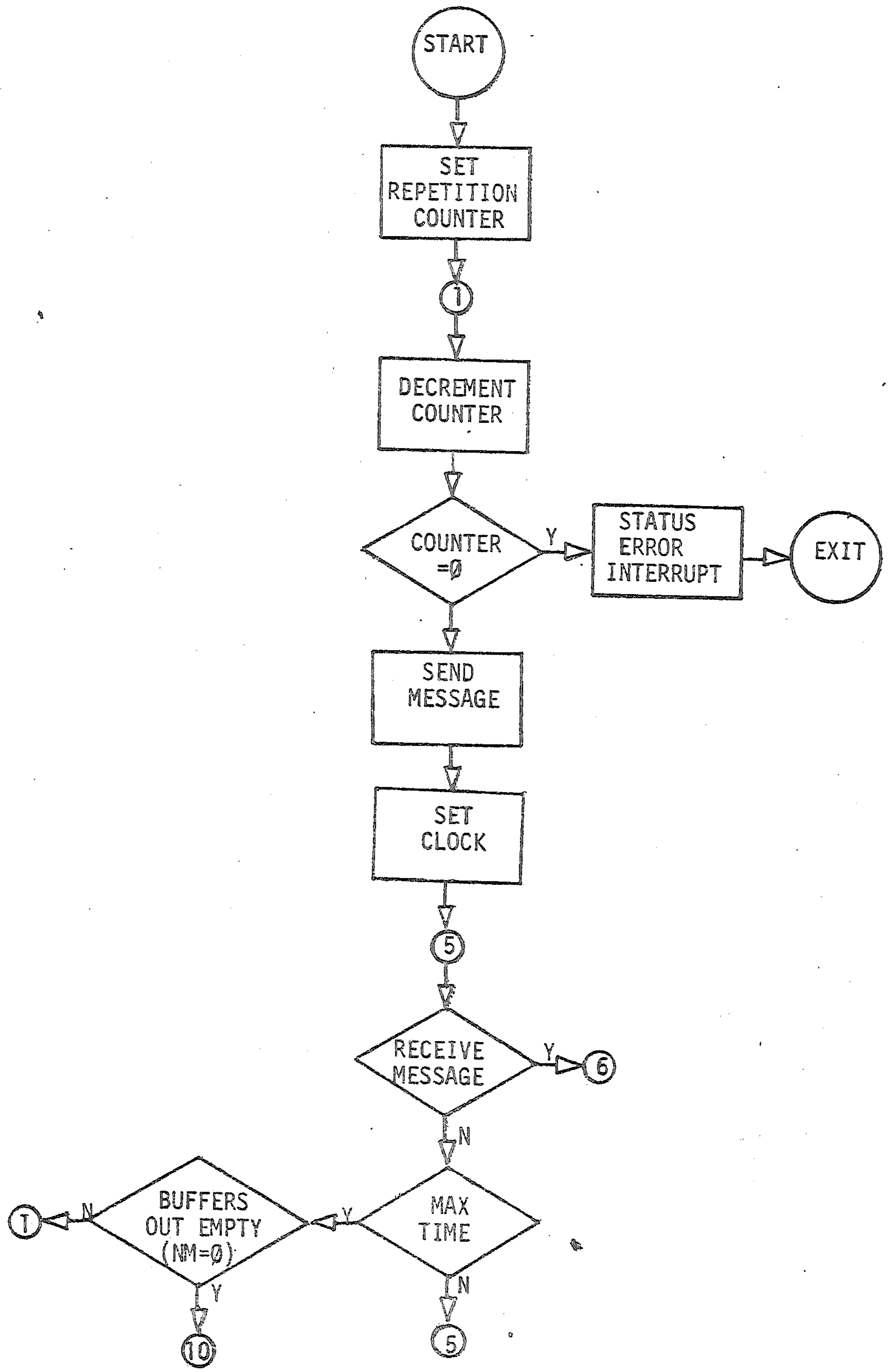
The role of a destination node in the DCS is substantially different from that of nodes in other similar communications systems. As has been mentioned previously, messages are addressed by the name of a process of a class of processes. Thus each message is broadcasted to all nodes, there to be accepted or disregarded depending on whether or not the desired destination process is active on the associated host. This fact and the need on the part of the error protocols for certain responses forces the recognition of active process names to be done within the node. Thus we have packaged within each node a small associative storage device which holds the names of the active processes within the node's host machine. When a message is passed through a node, other than the node which originated the message, the terminating process name is extracted from the message. This name is matched against the associative store of the node. If a match is made, then the match-bit of the message will be set as the message passes through the node. Note: It is only absorbed at the transmitting node. If the name is not found in the associative store, the message is ignored. If the match takes place, an input buffer will be siezed in the node or in certain cases in the host.

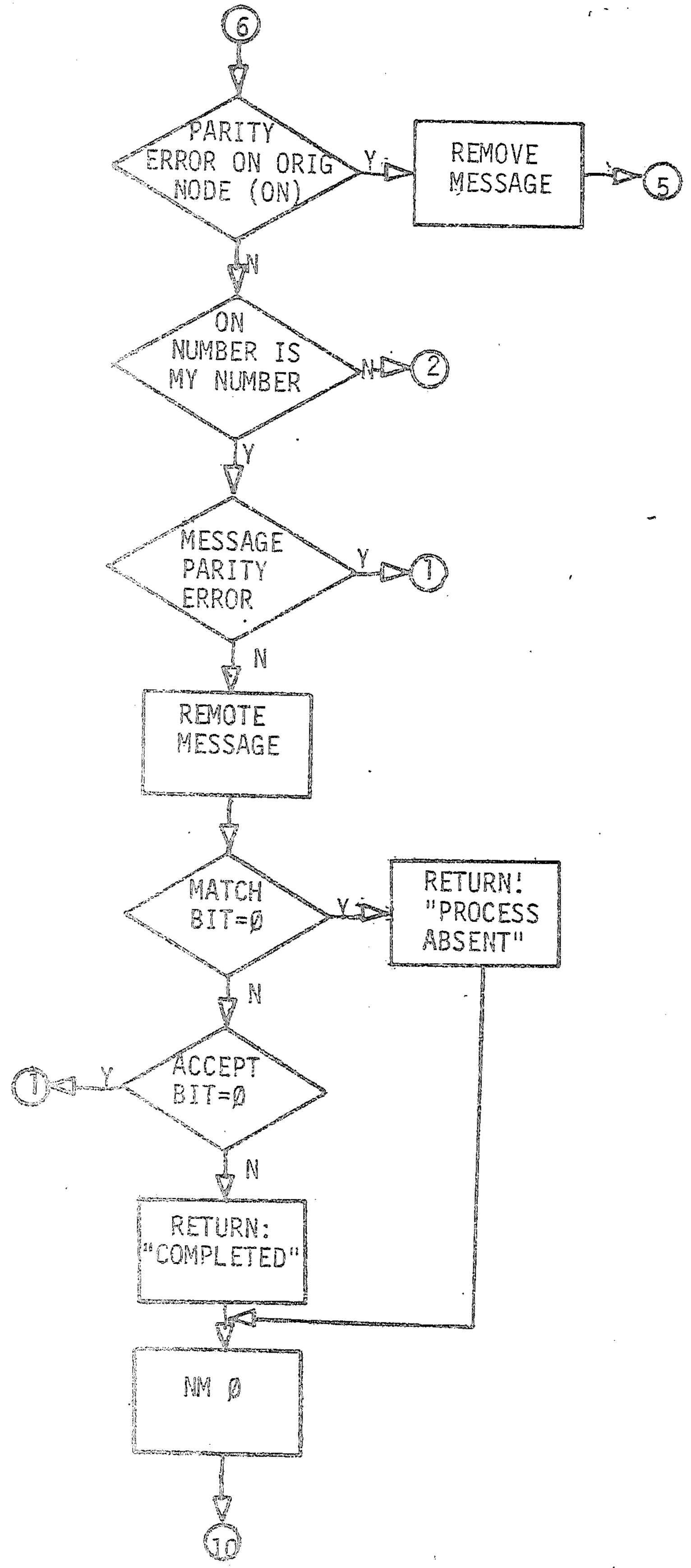
If such a buffer exists, the message will be copied and the accepted bit set in the message as it is passed on. When the message has been copied, the host will be notified.

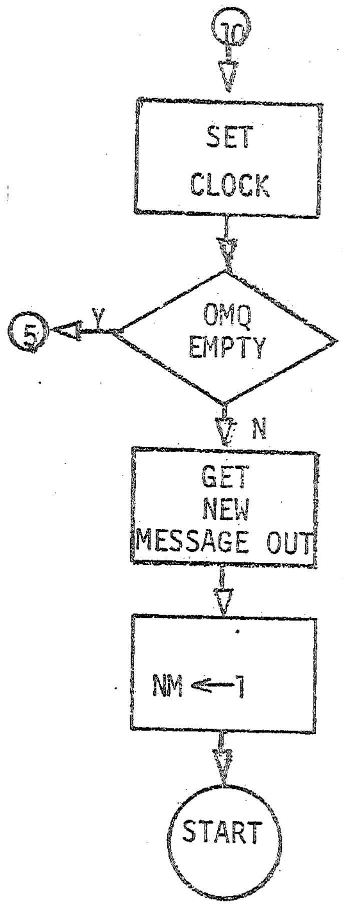
Message sequencing is used to insure that if an error occurs in a message its valid receipt by an addressed process, the addressed process does not receive a copy of the message without knowing it's a copy. This is achieved by means of a sequence field of size one bit which sequences messages sent by a transmitting process to a particular process of class of processes. Thus each process has a table of processes to which it is sending messages. Associated with each entry in this table is a sequence field that is updated at the start of a new message. This protocol assures the correct disposal of messages in the case of one process transmitting to another process. In the case where it is to a class of processes, certain types of errors will cause some messages to be mishandled. To correct this in the communications protocol is difficult. But the basic properties of our bidding type messages insures that there will be no malfunction in the above case. At worst a best choice will not be used or a retransmission will be necessary. Thus the communication system and the software architecture are well matched in this critical area.

The general node flow of control is shown in Figure 1.

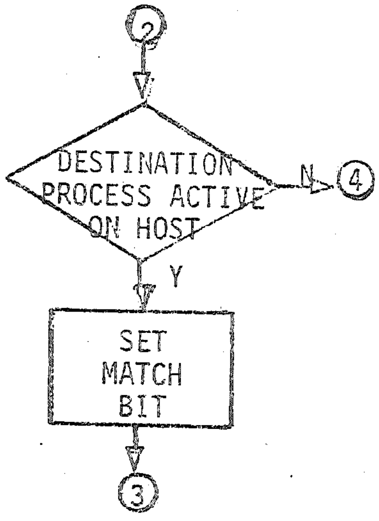
FIGURE I. FLOW OF CENTRAL WITHIN THE NODE



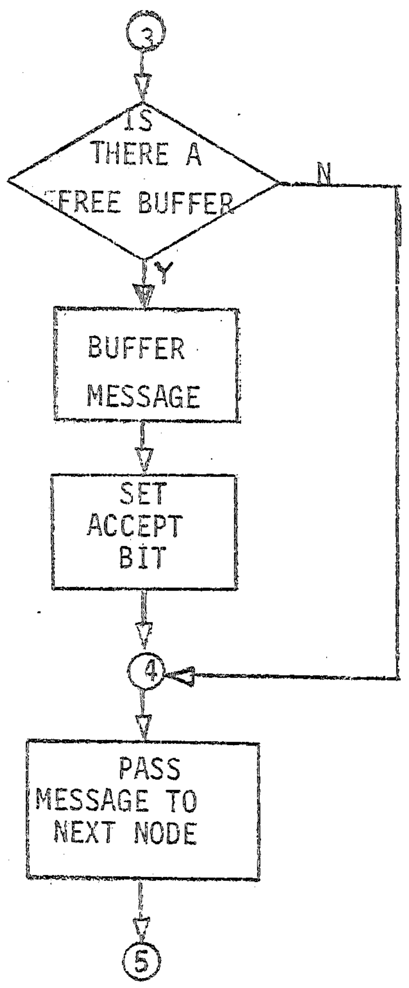

6
PARITY ERROR ON ORIG NODE (ON)
Y
REMOVE MESSAGE
5
N
ON NUMBER IS MY NUMBER
N
2
Y
MESSAGE PARITY ERROR
Y
1
N
REMOTE MESSAGE
MATCH BIT=Ø
Y
RETURN! "PROCESS ABSENT"
N
ACCEPT BIT=Ø
Y
1
N
RETURN: "COMPLETED"
NM Ø
10



10
SET CLOCK
OMQ EMPTY
Y
5
N
GET NEW MESSAGE OUT
NM ← 1
START



2
DESTINATION PROCESS ACTIVE ON HOST
N
4
Y
SET MATCH BIT
3


3
IS THERE A FREE BUFFER
N
Y
BUFFER MESSAGE
SET ACCEPT BIT
4
PASS MESSAGE TO NEXT NODE
5

III. SOFTWARE

SOFTWARE PROTOCOL

Before we begin an example, some backround information concerning interprocess communication paths, process addressing, and the teletype node (T-node) will be necessary.

The general flow of information within a host-node pair is shown in FIGURE 2. The Resident Message Router (RMR) routes messages from the ring onto the Incoming Message Queue (IMQ). It also places messages from the Outgoing Message Queue (OMQ) onto the ring. Should a message on the OMQ be destined for a process active on the host it is routed directly to the IMQ by the RMR and is not placed on the ring. Note that interprocess communications between processes on the same host is carried out via this "short circuit" by the RMR. Hence all communication appears to be carried out over the ring, as far as the process is concerned, whether it actually is or not. Thus if a process is communicating with many others it uses the same format for all, and if a process is moved, no one need know except the RMR.

All messages are addressed to processes rather than physical nodes or processors. Since in general there will be more than one process active per node, this requires an active process list in each node. Since files will be referenced in much the same way, a list of active files is also necessary. Each node, then has an associative store which contains a list of all active processes and files on the host. When a message is received in the node the process name in the address field is checked against this list. If a match occures the message is routed to the host. A routine called the ROUTER then routes the message to the correct process buffer within the host. This scheme of addressing allows a process to keep the same address, its name, even though it has been moved from one host to another.

Each terminal may be connected directly to the ring via a T-node. T-nodes are a special form of the more general node. There is only one active process, the user, in a T-node. In general, the user doesn't want to type in the name of the process he wishes to talk to each time he sends a message so this information must be held in a register, the Send Register (SR). If a process crashes, the user would like to be able to talk to his Logger agent again, hence a Logger Register (LR). Unless the user is to perform all transfers, external processes must have the ability to modify the SR. We have elected to give this power only to the processes named in the LR or SR (before modification). There must also be a register which gives the address of the T-node on the ring. This is the user register (UR) which is set to the T-node's physical name initially and changed to the user-name after the user has logged in. To prevent another process from jamming a T-node by sending it volumes of garbage, incoming messages addressed to the T-node must be screened. We do this by allowing the T-nodes to accept only those messages addressed to it which originated with the process named in either the SR or LR. Finally, the T-node must have two wired messages, a request-for-logger-services and an acknowledgement if a bid is returned.

The T-node connection protocol begins with the user pressing the "red button" which sends out the prewired request-for-logger-service message. This message is received by a general routine LOGGER on one or more of the processors. Each LOGGER sends back a response containing its specific name. The T-node loads its SR with the name of the first LOGGER to respond, say LOGGER.A, and sends the wired acceptance message to this LOGGER. LOGGER.A after receiving the acceptance examines the contents of the T-nodes LR to determine if the T-node is currently logged in. This examination is a privileged node control command. If the node is not logged in (LR=∅) LOGGER.A creates, in a way to be described later, a login process for that user, say LOG.7. LOGGER.A now places "LOG.7" in the T-node's LR and SR after giving LOG.7 the T-nodes name. LOG.7 logs in the user and places his user-name, say US.3, in the UR. If LOG.7, or some other name, was already in the LR, i.e. a login had already taken place, LOGGER.A would merely awaken the login process and no new login would be required.

Assume that the user now wishes BASIC service. He communicates his desires to LOG.7, who acting as agent creates a BASIC process, BA.2, and sends it the name US.3 as the input (SIN) and output (SOUT) channel bindings. LOG.7 now changes the SR of the T-node to BA.2 and the connection is complete. The user may dissolve the connection in one of two ways. He may exit normally in which case BA.2 awakens LOG.7, changes the SR to "LOG.7", which reconnects the user, and then deallocates itself. He may also push the "red button" which causes him to be reconected to LOG.7 as described before while BA.2 continues to run. He may now initiate another job using LOG.7 as an agent. BA.2 continues to run until US.3 fails to accept its messages since the SR no longer contains "BA.2". BA.2 will now send a message describing its current status to LOG.7 who will determine a course of action (possibly notify US.3).

SCENARIO

It is hoped that the following examples in which we follow through in detail some interactions on a small sample system will do a better job of describing the message flow than a prose description. Should detail lead to repetition in certain areas, we will delete the detail in these areas after an initial description.

Let us consider a ring of four inhomogeneous machines. Machines A and B have large core memories (12K) and a disk. They are capable of control functions such as checking the ring. They can support BASIC service, LOGGER, text editing (TEXT), and file service. They also can supply an economics game (ECON) in which the computer functions as one player and accepts up to three user players. In general each machine will be devoted to one service for a sizable period of time to reduce data movement. The service processes are reentrant so if we stay with one service we need only move the user context.

Machine C has the same amount of core as A and B but has no disk. Thus machine C must depend on A and B for secondary storage. Since this requires that all references to secondary storage by C be over the ring which will be slow, it seems a good strategy to devote C to one service and only as many users as will fit in core. We will assume that C is devoted to BASIC service.

Machine D is the primitive machine. It has only enough core storage to support a TEXT service, one user, and the most primitive monitor usable on the ring. D must depend on A, B, or C which can support full monitors to perform its monitor tasks. Its first order of business, as you will see later in the example, is the establishment of the monitor services it needs on the other machines. Thus to some degree D functions as a useful parasite, both using and providing services.

In the example we will consider four users, a TEXT user (AD.1), a BASIC user (SC.2), and two students (ST.3 and ST.4) who wish to use the economics game. We begin our example with the system shut down and follow it through power up and the four sample users.

We begin by loading A with the monitor and starting it in the normal fashion along with its node. Since each node is equipped with a "short circuit" which maintains the connectivity of the ring when a node is down, the ring now consists of one machine, A. A's first duty is to test the operation of the ring by performing a RING CHECK.

The RING CHECK routine in each monitor (A, B, and C) is solely responsible for checking for bad transmission, sick nodes or sick machines. It does this by sending out a request for all other machines to respond with a machine-O.K. message. If one node fails to respond a special diagnostic message is sent to that node. If the node still fails to respond, it or the attached host is believed to be sick and the node is told to shut itself down. A primitive section of the node recognizes this one message and short circuits the ring around the node, effectively shutting it off, for some time T. If a second shut-down message is received during T, while the node is temporarily off, from another machine which has independently determined that the node is sick the node shuts down entirely. If none is received the node returns to life and the host begins to pick up the pieces and returns to normal processing.

B and C are now started in the same way A was with one difference, none of their routines are given control status. In the associative store of each node in each entry for an active process there is a bit which indicates whether that process has control status or not. Only processes with control status can send control messages which alter the ring such as those used in the RING CHECK. Since the bit can only be set by a control message, only processes with control status can confer control status. Obviously the routines on one machine, in this case A, must begin with control status. In A's RING CHECK it detects the presence of B and C. Recognizing that they have suitable memory protection, A confers control status via control messages, on B and C's RING CHECK and other routines.

We will assume that D is started at a later time, so now AD.1 sits down at a terminal and turns on the terminal and node. The T-node sends out the prewired request-for-LOGGER which is received by each node (A, B, and C) in turn. Assume A's LOGGER.A is accepted first. LOGGER.A determines that the T-node is not currently logged in and so sends out a request to all RESOURCE ALLOCATOR'S (RA(BID)), for a login process. Since the message is addressed to a RESOURCE ALLOCATOR, a routine which each machine must have, each node places the message on the IMQ of its host. Within each host the ROUTER places the message on the RA(BID) Queue. When the RA(BID) routine is activated as part of the normal monitor sequence in A and B it recognizes the request for a LOGGER and using the "from" field of the message returns a service-available bid. Since C annot supply login service, its RA(BID) returns nothing to LOGGER.A A and B can both support login service,so their RA(BID)'s each return to bid to LOGGER.A . LOGGER.A picks the minimum bid, say A's and returns an acknowledgement message containing the T-node's name to A's RA(ACK). As before with A's RA(BID), A's RA(ACK) receives the message, set up the login process. LOG.A1, and sends a message to LOGGER.A giving it the name of the created process. When LOG.A1 is initiated it is given the name of the requesting T-node. LOGGER.A after receiving the acknowledgement completes the connection as described earlier. LOG.A1 now sends the terminal (via the T-node of course) a prompt for login information. The user and LOG.A1 communicate until the login is complete at which time LOG.A1 sends a message placing the user name, in this case AD.1, in the UR of the T-node.

Our user, AD.1, now would like to connect to a TEXT process. AD.1 sends a message to LOG.A1 requesting TEXT service.

LOG.A1 sends out a request to the monitor routine INITP requesting TEXT service. INITP sends out a request for TEXT service to all RA(BID)'s. To send the message INITP places the message on the OMQ of A. The RMR recognizes that the message should be routed to A as well as B and C, hence it places it on A's IMQ and then sends it around the ring to B and C. C's RA(BID) finds that the service is not available on C and so returns no messages to A. A's RA(BID) responds with a bid which it places on the OMQ where it is routed to the IMQ and then to INITP's buffer. In the meantime B's RA(BID) has responded with a bid over the ring to INITP's buffer some time before or after A's bid. After INITP has waited a suitable period of time it examines the returned bids. It picks the lowest bid and returns an acknowledgement message to the RA(ACK) of that host. The RA(ACK) sets up a TEXT process, say TEXT.1, and returns the process name with an acknowledgement to LOG.A1. What follows now is the T-node connection game previously described.

Should INITP receive no bids in response to its request, it calls the Resource Finder (RF). It is the RF's job to find the desired object program file and return its location to INITP. INITP then sends out requests for bids again, providing the bidding RA(BID)'s with the information. Things now proceed as before with the normal bid. If no bids are returned INITP returns a message to the calling process stating that no RA will bid on the process.

The RF routine is very helpful in activating dormant processes. To illustrate this assume the BASIC user SC.2 logs in and creats a BASIC process, BA.1 on C. He discovers he must leave before he has finished and so he wishes to cause his process BA.1 to go dormant. He communicates this to BA.1 which erases their connection as described in the section on T-nodes and calls the monitor indicating that it wishes to go dormant. The monitor sends out a request-to-create-file to the RA(BID)'s who return bids on a file process. As before with TEXT, the file process is created, FILE.1, on either A or B since only they have disk. C now transmits the core image to the files as a series of messages, and closes the file. SC.2 logs off. At a later time when SC.2 returns he may, after logging in, request service from BA.1. Since some hosts have BASIC but none have specifically BA.1 as an active process, none will respond. The RF, however, will locate the file BA.(SC.2) and so requests will now be sent out for activating this file. Since the machines are inhomogeneous, only C can bid since it created the file. If C is free, the process file is loaded in much the same way it was created. If C is not free, SC.2 must wait until it is before he can reenter his process. Note that this is not a problem on machines such as A and B which support more than one user. In the future if C is busy we may try to transfer the process to another host but at this time the mechanisms for performing this have not been fully explored. Note that C's monitor can itself load C with the BASIC processor from A or B at startup using almost exactly the same mechanism.

Assume now that ST.1 has logged in and is connected to ECON.1. ST.2 now logs in to LOG.6 and wishes to enter the game. A small routine is provided on the system which allows users to find out what processes other members of their group are connected to. ST.2 requests this service and is given the names of all St's on the system and the corresponding processes. Among these is ST.1... ECON.1. ST.2 now requests connection with ECON.1. LOG.6 receives the message and sends out a normal bid request for ECON.1 service. Assume ECON.1 is currently active on A. A's RA(BID), before it attempts to bid, recognizes that ECON.1 is a currently active process. It therefore sends no bid but places a message with the name of LOG.6 and a special escape bit set on in ECON.1's message buffer. ECON.1 recognizes the escape bit and sends a message to LOG.6 requesting the name of the process (T-node) to be connected. The connection game proceeds as before with LOG.6 and ECON.1 in command.

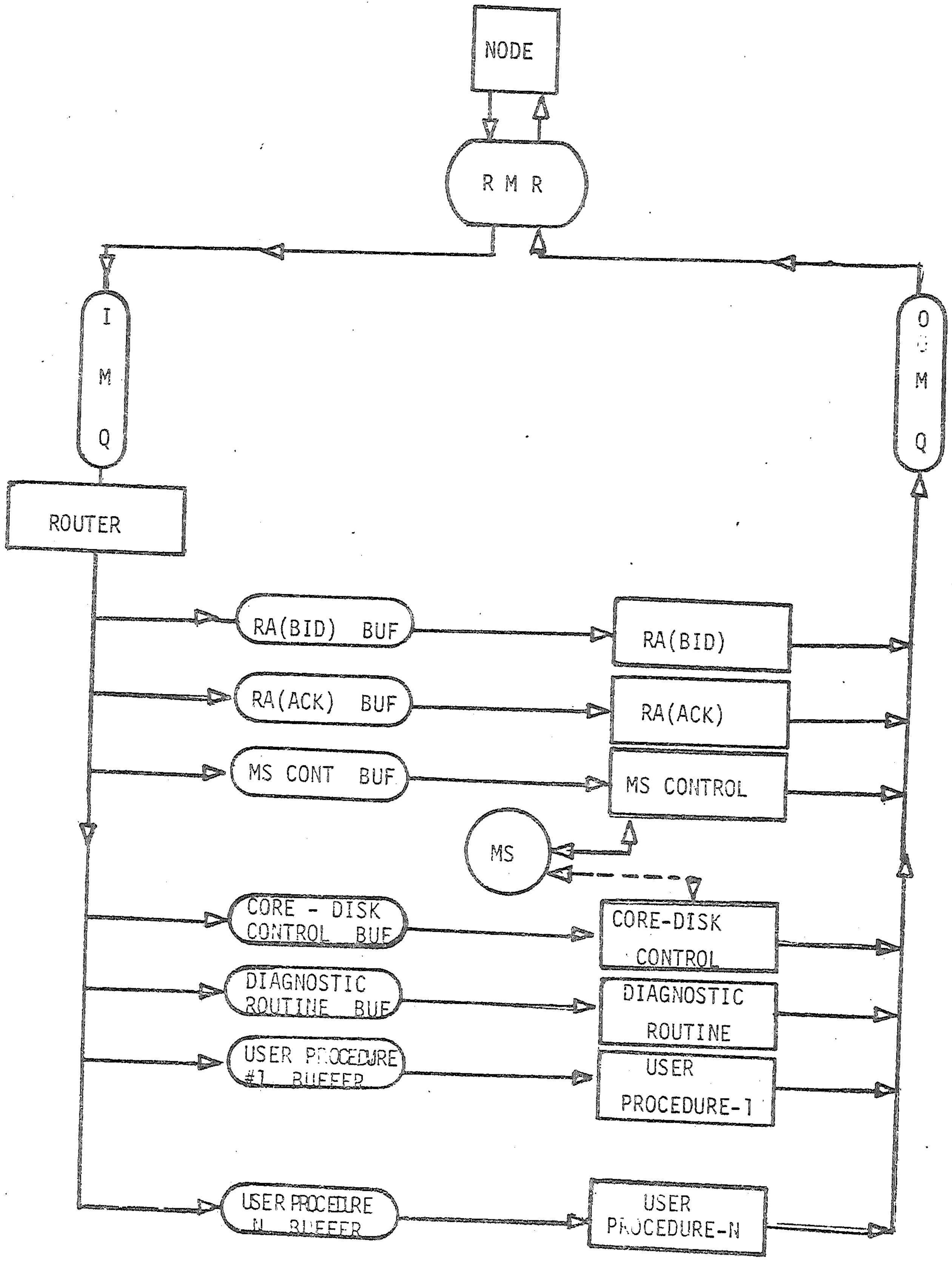
The last point to be considered is the connection of D. One of the jobs of the larger machines is to provide the smaller machines with certain monitor routine services. As with any other user process this is accomplished through bidding. Note that due, to the communication paths within the hsot, most of the monitor service routines may reside on a different host or may move from host to host without affecting any other routine, since communication is to processes and not physical addresses.

When D is started it is loaded with the minimal monitor. The first job of the monitor is to establish missing routines, notably RA(BID) and RA(ACK), on other hosts. To do this the monitor acts as any other user, requesting bids, acknowledging one of the returned bids, and setting up communication with the process after it has been created on another host. After the routines have been established, D acts much like C excluding the fact that its monitor is a bit slower since it must perform communication over the ring rather than through the "short circuit" in the RMR on which C's monitor messages are routed since all communication is within the host.

Before we end this discussion of the software there are two points regarding the monitor we would like to clear up: Timing and garbage collection. If the host supports more than one user at a time, so a job queue is necessary, a system of two queues is employed, a job queue (JOB Q) and an I/Q pending queue (I/OPQ). When a process calls the monitor, it may communicate two time intervals, a time-in (TI) and a time-out (TO). The monitor adds a third time, the origination time (TØ) before it places these three times and the process name on the job queue. Processes start at the rear of the JOBQ and are processed when they reach the front, unless they have a pending I/O command in which case the job is placed on the I/OPQ. The monitor processes jobs first from the I/OPQ if the pending condition has been satisfied, and if not then from the JOBQ. Note that a job must wait the entire length of the JOBQ before it can be placed on the I/OPQ. There are two exceptions to the above rules for processing from the queues. If T is the time on the system clock, then a job is never processed until T≥TØ + TI and it is always processed if T≥TØ + TO. Hence a process can choose to be dormant for a certain period of time or to be awakened after a certain period of time, unconditionally.

In this system there is always the chance that a job will be left in a permanent pending state due to some failure elsewhere on the ring. Therefore, there is a monitor routine called the Process Garbage Collector whose job it is to clean up obviously dead processes. It does this by examining the TØ of each job on the I/OPQ. If the systems clock time is greater than TØ + TG, where TG is a parameter, the process is deleted from the host.

FIGURE 2. COMMUNICATION PATHS



BIBLIOGRAPHY

Farber, David J., "A New Scenario for a Distributed Computer System", DCS Working Notes, April 19, 1971

________, "Supplement to Proposal for Research on Distributed Computer System", submitted to the National Science Foundation, October, 1970, by the University of California, Irvine.

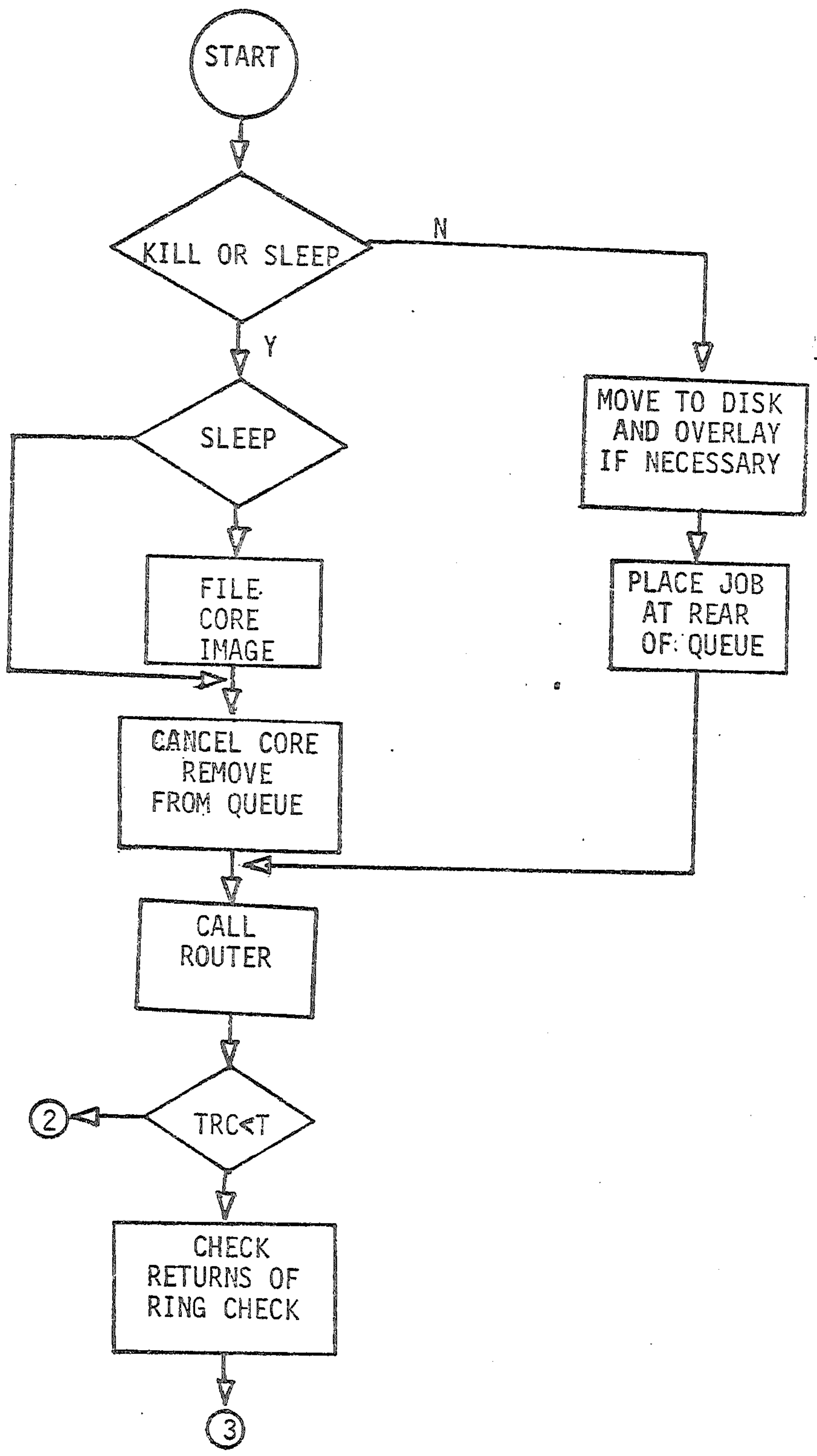
APPENDIX

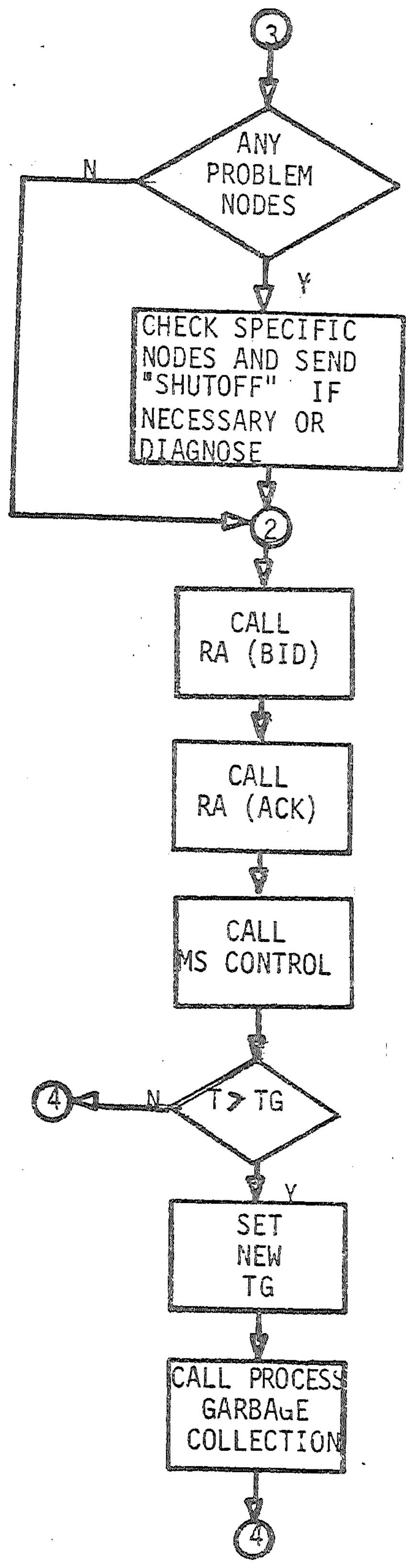
High Level Software Flowcharts of Fundamental Routines

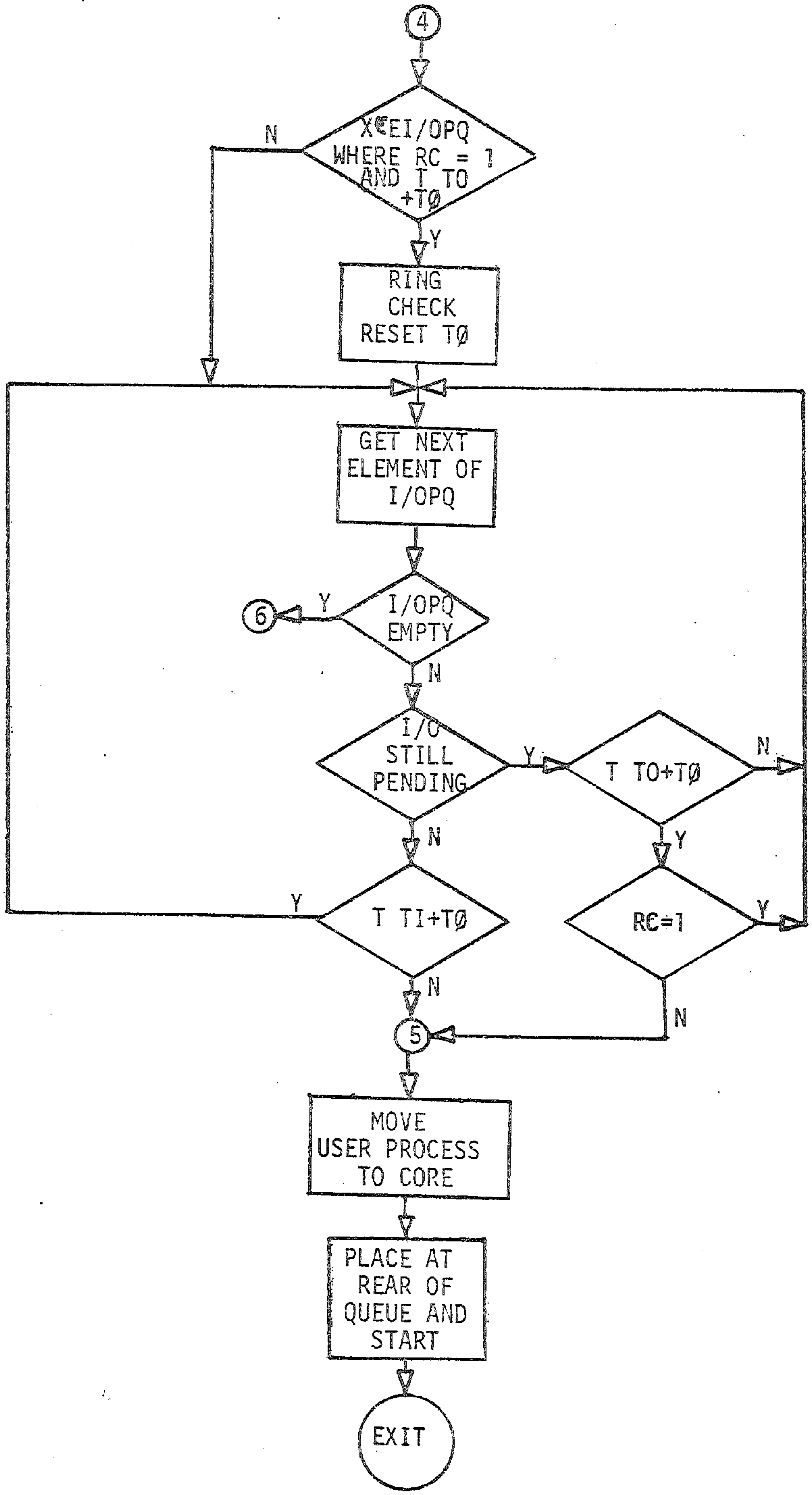
The following figures constitute a preliminary description of the functioning of certain key routines. Since we are still in the early stages of developement, the emphasis is on flow of control and information rather than detail. Most of the routines have been functionally described in the text to a sufficient degree to allow fairly easy reading of the flowcharts.

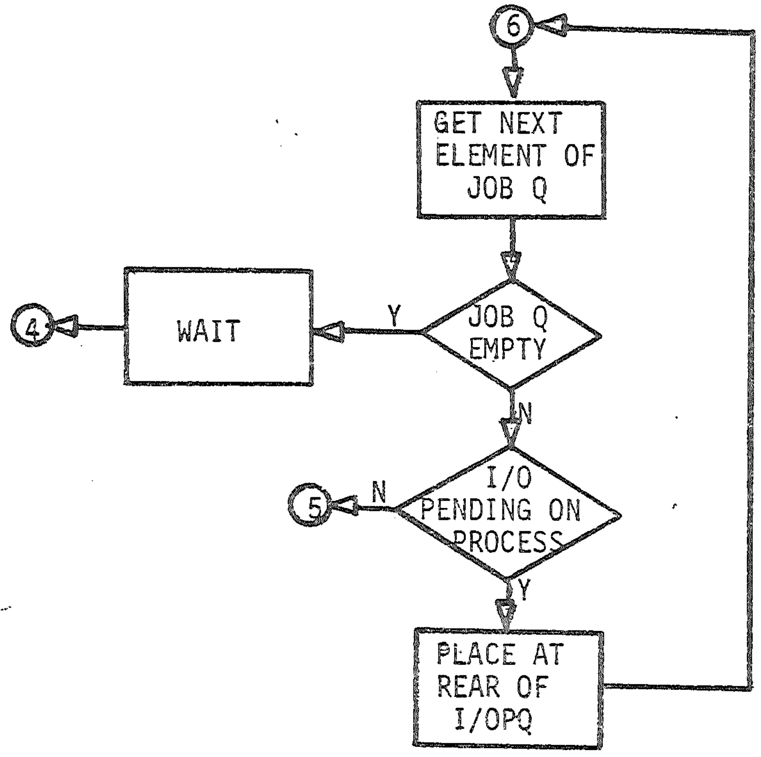
One minor point, however, which may cause one trouble is the RING CHECK bit (RC) in the monitor routine. When a user process calls the monitor in addition to passing the TI and TO it passes an RC. The RC tells the monitor what to do in the case of a time-out, that is when the waiting time, TO is exceeded. If RC equals zero the process is awakened; if RC equals one a RING CHECK is performed and the process is not awakened. This option proves valuable since many messages expect responses; hence if no response is received, it most probably is due to ring failure.

MONITOR ROUTINE

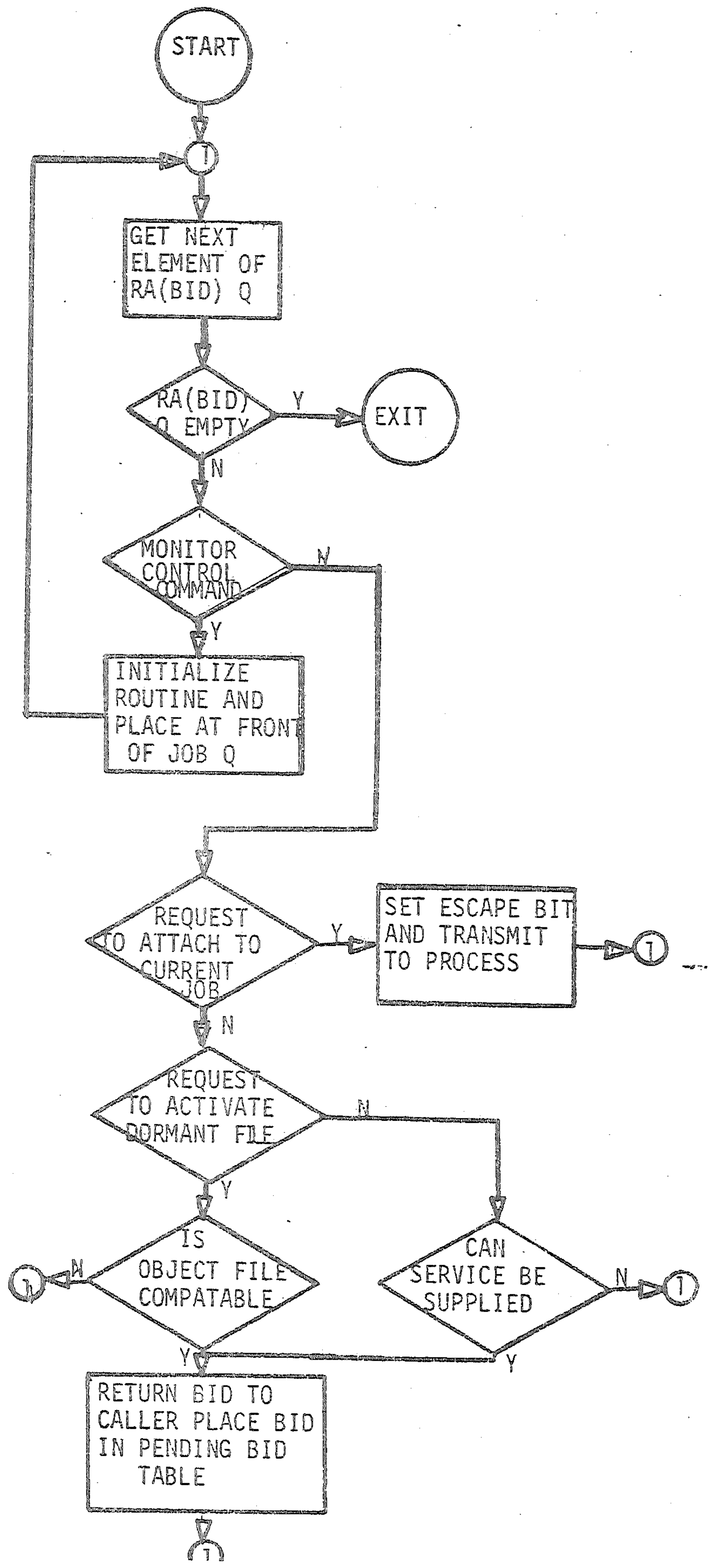

START
KILL OR SLEEP
N
Y
SLEEP
FILE CORE IMAGE
CANCEL CORE REMOVE FROM QUEUE
MOVE TO DISK AND OVERLAY IF NECESSARY
PLACE JOB AT REAR OF QUEUE
CALL ROUTER
TRC<T
2
CHECK RETURNS OF RING CHECK
3


3
ANY PROBLEM NODES
N
Y
CHECK SPECIFIC NODES AND SEND "SHUTOFF" IF NECESSARY OR DIAGNOSE
2
CALL RA (BID)
CALL RA (ACK)
CALL MS CONTROL
T > TG
N
4
Y
SET NEW TG
CALL PROCESS GARBAGE COLLECTION
4

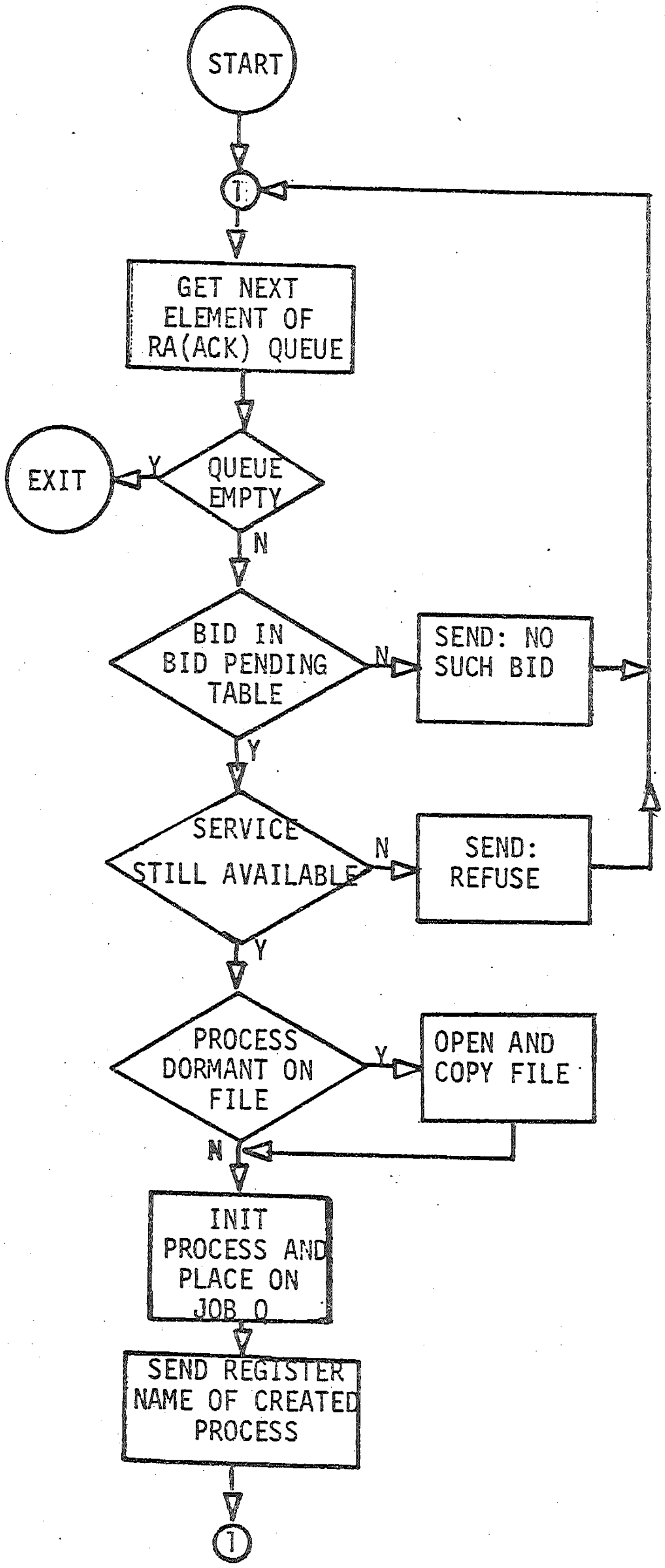

4
X∈I/OPQ WHERE RC = 1 AND T TO +TØ
N
Y
RING CHECK RESET TØ
GET NEXT ELEMENT OF I/OPQ
I/OPQ EMPTY
Y
6
N
I/O STILL PENDING
Y
T TO+TØ
N
Y
RC=1
Y
N
N
T TI+TØ
Y
N
5
MOVE USER PROCESS TO CORE
PLACE AT REAR OF QUEUE AND START
EXIT


6
GET NEXT ELEMENT OF JOB Q
JOB Q EMPTY
Y
WAIT
4
N
I/O PENDING ON PROCESS
N
5
Y
PLACE AT REAR OF I/OPQ

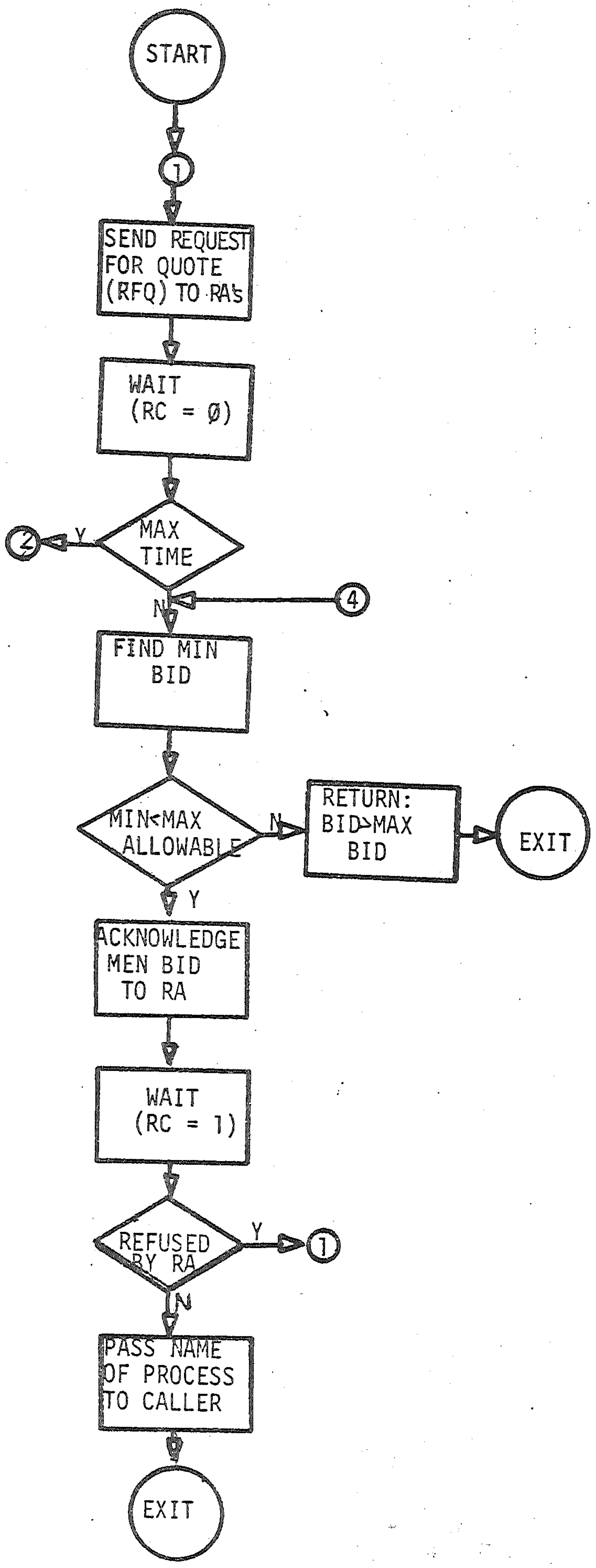
RA(BID) ROUTINE

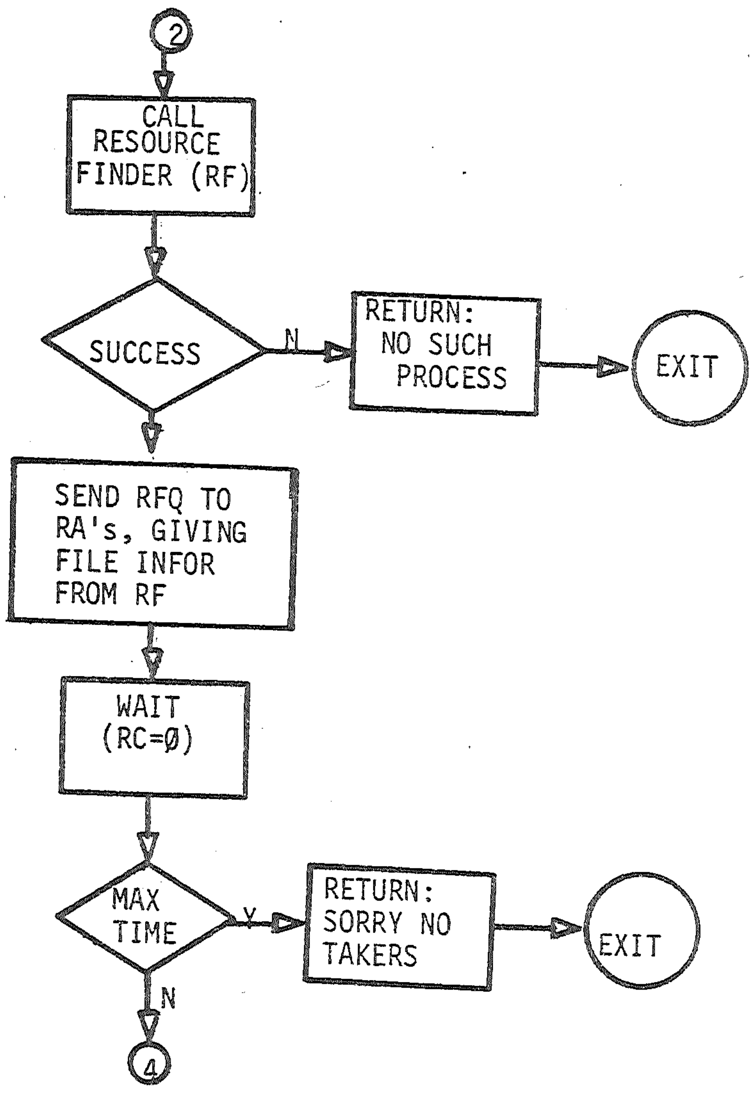

START
1
GET NEXT ELEMENT OF RA(BID) Q
RA(BID) Q EMPTY
Y
EXIT
N
MONITOR CONTROL COMMAND
N
Y
INITIALIZE ROUTINE AND PLACE AT FRONT OF JOB Q
REQUEST TO ATTACH TO CURRENT JOB
Y
SET ESCAPE BIT AND TRANSMIT TO PROCESS
1
N
REQUEST TO ACTIVATE DORMANT FILE
N
Y
IS OBJECT FILE COMPATABLE
N
1
CAN SERVICE BE SUPPLIED
N
1
Y
Y
RETURN BID TO CALLER PLACE BID IN PENDING BID TABLE
1

RA(ACK) ROUTINE

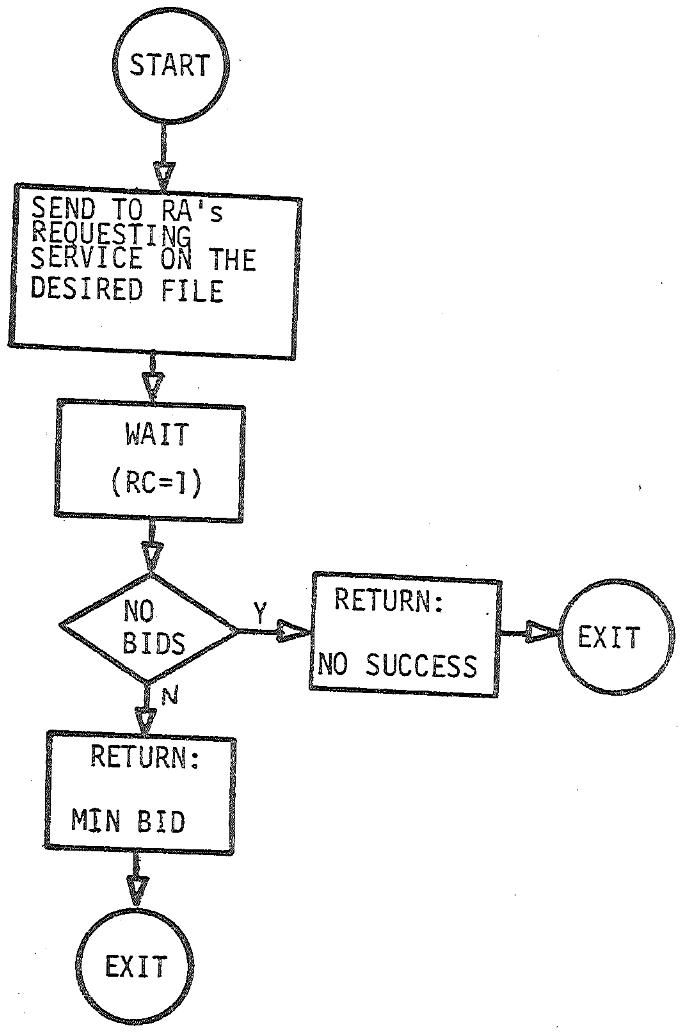


INITP ROUTINE




2
CALL RESOURCE FINDER (RF)
SUCCESS
N
RETURN: NO SUCH PROCESS
EXIT
SEND RFQ TO RA's, GIVING FILE INFOR FROM RF
WAIT (RC=Ø)
MAX TIME
Y
RETURN: SORRY NO TAKERS
EXIT
N
4

RESOURCE FINDER ROUTINE



APPENDIX F

THE DISTRIBUTED FILE SYSTEM-A FILE SYSTEM FOR THE DISTRIBUTED COMPUTER SYSTEM

Frank Heinrich

ABSTRACT

The stucture of a reliable, fail-soft file system is described. Failure of any a processor or storage media devices will affect only a portion of the files for a portion of the users. The system also provides backup to minimize the information lost when a particular file becomes permanantly unavailable due either to system failure or user error. The system is single level, that is, each file maintains its fully qualified global name. There is no hierarchy of directories, however there is a hierarchy of processes to assist in locating a file.

The system resides on several processors in a communication network. Messages in the network are broadcasted to all locations, addressed to process names rather than hardware processor addresses. Thus an originator of a message need know nothing about the topology of the network, nor the location of the process to which the message is being sent, in fact that process need not remain in a fixed location.

The system has a single central component which gives initial information to assist in locating files. This component is n-plexed on different processors for reliability, all n components being identical. This component is keyed by owner name and provides the name of a process, a catalog, which references all files for that owner. Several different catalogs exist on distinct processors. A catalog will serve more than one owner and each owner is served by only one catalog. For each file, the catalog lists another process, a volume, which has access to the actual physical location of the file. A volume is associated with each physical storage media and

contains a table of contents for that media, which contains file name, owner name, protection information, and the name of the catalog which referenced it. When a file is located and opened, a process having a name corresponding to the file is created to run in the processor to which the storage media is attached. Henceforth all traffic with the file takes place through this process and is addressed as a general process in the overall communication network.

A failure of the hardware storage media or the processor containing a volume affects only those files of those users which may be on that particular storage media device. Since process addressing does not require topological knowledge, the media could be moved to another processor where the volume and file access processes could be reestablished with no effect on the rest of the system. The failure of a processor containing a catalog causes temporary inconveniences only to those users who are listed in that catalog. The catalog can easily be reconstructed by reading the table of contents of all volumes, extracting the file information for those files which were referenced by the missing catalog.

The backup of individual files is provided by generation structure. When a file is updated or modified, a new generation of that file is created, the old version still exists as generation 'current minus one'. Effort is made to ensure that successive generations are on different volumes. Users have access to all previous generations, as well as automatic deletion of files older than a certain number of generations, as specified by the user. Appropriate archival backup proceedures (i.e. File System Dumps) will also be used.

The issues of protection, access modes, concurrent access, and structural sharing of files are also discussed.

THE DISTRIBUTED FILE SYSTEM-A FILE SYSTEM FOR THE DISTRIBUTED COMPUTER SYSTEM

Frank Heinrich

The Distributed File System was designed to provide a reliable, fail-soft file system for the Distributed Computer System. The file system provides means for storing and retrieving information in addition to the use of core storage during a working session. The file system also performs a library function, storing information between sessions.

The fail-soft criteria implies a system which is vulnerable to partial failure, yet relatively secure from total failure. In addition it should be relatively easy to recover from partial failure. If the system is to be fail-soft, the failure of either a processor or storage media device should only affect a portion of the files for a portion of the users. The individual user's environment should also be fail-soft. That is, if a user's file becomes permanantly unavailable or damaged either through system failure or his own error, the work necessary to recreate that file should be minimized.

To provide the fail-soft characteristics of the file system, it is distributed over several processors in the Distributed Computer System. Each processor should be autonomous and independent, capable of providing it's portion of the file service without the assistance of any other particular process, any assistance necessary should be available from more than one source. The system accomplishes this with a minimum of redundancy.

Since the components are to be independent and distributed, there could be no distributed hierarchy of directories, which limit context and allow partially specified file names. The system is single level, that is all files retain their fully qualified global names so a file can be uniquely identified regardless of the availability of other components of the file system.

There is a hierarchy of processes which assist in locating a file; however, the processes are expendable and, if destroyed, can easily be recreated.

The system does have one central component which provides initial information to assist in locating a file. To prevent failure of this component from tying up the rest of the system, it is n-plexed on different processors, all n components being identical. It is keyed by owner name and for each owner provides the name of a process, a <u>catalog</u>, which references all files for that owner.

Several different catalogs exist on distinct processors. A catalog will serve more than one user, and each user will be served by only one catalog.

For each file, the catalog provides the name of another process, a <u>volume</u>, which is associated with the physical storage on which the file is located. A volume contains files for many users; the files of any one user are distributed over many volumes. A volume process runs in the processor to which the physical media containing the file is attached, and there is one volume process associated with each storage media device (e.g. disc pack or cartridge) which contains the table of contents for that device. The entry for each file in the table of contents includes the fully qualified global name, dates of creation and modification, protection information, the name of the catalog process which referenced that file, and the location (physical address) of the file header on the storage device. The fully qualified global name includes, in addition to the file name, the owners name and the generation number. Note that all this information is accessed through a process, although the data itself will be stored on the media, in a predefined, reserved location. Device names and physical addresses are never used except by the volume process itself. The volume process will also manage the storage allocation on its device for creating new files and deleting old ones.

In summary, the following steps are the normal way of locating a file in the Distributed File System. These steps will most reasonably be carried out by a process which is acting as an agent for the user at his terminal or for the user process which requires access to a file; however, there is no reason why the user or his process cannot carry them out. Step 1: Broadcast a message to all the central components, specifying the owner name for the file that is desired. Accept a reply from the first one to respond. The reply should specify the catalog name which serves the owner of the file desired. Step 2: Send a message to the catalog specifying the file name and owner of the file desired. This message is to a specific process; there will be only one process which can provide the information. The reply will specify a volume name which will give the final access to the file. Step 3: Send a message to the volume (this is still a general message in the communication system, not a hardware address) specifying the file name and owner of the file that is desired. Step 4: The volume will reply requesting verification for the protection specification in the table of contents. If the user is authorized to access the file in the way he has requested, the volume will request the operating system to create a process with a name which corresponds to the file name. The process name will be communicated to the requestor and all further traffic with the file will take place through this process. Thus files are addressed as general processes in the communication scheme. Files retain the same flexibility and mobility of general processes both in initially locating and in the transfer of data into and out of the file.

How does this structure provide the fail-soft characteristics claimed? Consider the possible failures that could affect the system.

(1) The storage media itself could fail (e.g. head crash on a disc pack).

(2) The device on which the media is mounted or the processor which controls that device may fail.

(3) The volume may fail, either the software or hardware in the processor which contains it.

(4) The catalog may fail, either software or hardware.

(5) The central component may fail, either software or hardware.

To recover from a failure of type (1) is near impossible. It would require that all volumes have a duplicate; however, we are trying to avoid this type of redundency. It is possible that archival backup may be able to restore some or all of the lost information onto another volume. In any case, only those files on the damaged volume are affected. All the files on the other volumes remain intact, the access to them in no way affected.

Failures of type (2) are much easier to recover from. Assuming the media itself remains intact, all that is necessary is to move the media to another device or processor. Traffic with the files on that media and access to the information in the volume table of contents is only temporarily interrupted. When the media is on the other processor, the volume process is reactivated in that processor and the access processes are reactivated there also. Then access to the volume can begin again and the traffic with the files can resume. Since all addressing is to processes, the user processes are not affected by the change in location. It appears to them only a slight delay has occurred.

Failures of type (3) are also fairly easy to contend with. All that is needed is to reestablish the volume process, possibly on the same processor if only a software error occured. Since the table of contents information is stored on the media in a predetermined location, it should be simple to establish a process to have access to that information. In this case, however, the access process for all the files in that volume that were active will probably be lost, effectively closing the file. Either a hardware or software error in a processor will probably destroy all information about what processes are active. So in this case users who were trying to open a file will gain that access with only a delay; however, the users who already opened files will have to reopen the file and continue where they left off. It is also possible a file that was in use may be damaged beyond the ability of the user to recover and the file will have to be recreated.

Failures of type (4) are less serious in consequences to users, but will probably take more time to recover from. The failure of a catalog will not directly affect any files, however it will make locating a file inconvenient. For all practical purposes those files owned by the users served by the missing catalog will be inaccessable for a short time. Recovery requires reconstructing the catalog information. This is easy to do but may take some time. All that is necessary is to read the table of contents for each volume, extracting the information for those files which were referenced by the missing catalog. A problem can occur if there is a volume missing at the time the catalog is reconstructed; however, when that volume is again made available, all the catalogs which are referenced in that table of contents can be checked to insure that they are up to date.

Failures of type (5) have virtually no effect. Since the central component is n-plexed on different processors, the failure of any one of them does not prevent access to the information contained in the others. It is possible there may be a slight degradation in response if the traffic with the other copies is high, but this is fairly insignificant.

The structure provides for fail-soft behavior for the failure of any component of the file system, either software or hardware. Of course multiple failures of more than one component can cause much more chaos, but it is expected that the probability of simultaneous multiple failures is much lower than for single isolated failures.

Providing fail-soft behavior for the individual user's environment implies that the loss of any file, either due to system failure or user error, should cause a minimum of lost information. It should be easy to restore the file to its current state without having to recreate it entirely from scratch.

Standard archival backup (i.e. file system dumps) does provide some measure of backup in this area, however it is often the case that the archival backup is too old to be of any real value. File system dumps are usually not taken more often than once a day, but it is usually the case, especially with program files, that if any changes were made at all, there were several successive revisions within a single session. Thus the archival backup provides either an exact copy of the file needed or a previous version too old to be of any real value.

To provide backup facility with a finer increment, and thus a better chance of minimizing the work necessary to recreate the file, all files will have a generation structure. When a file is updated or a new version of a file is created it becomes the new current generation of that file.

The old current generation still exists and is available as generation 'current minus one'. All previous generations are available, being accessed as generation current minus some increment. It is possible to update or modify a file without creating a new generation, but the user assumes the risk involved in loosing the incremental backup. Effort is made by the file system to insure that successive generations are on different volumes to prevent permanent loss of a volume from affecting the incremental backup.

To prevent the generation structure from becoming too extensive, the file system will provide an automatic facility for deleting files older than a certain number of generations, that number being specified for each file. Thus when a new generation is created, the oldest generation of that file will be deleted, providing that at least the minimum number of generations for that file exist. Also, users will be able to explicitly delete any generation.

To provide additional backup, and to guard the user against his own inadvertent errors, files which are deleted will not actually be removed from the file system. Files which are deleted will only be marked as deleteable, they will remain in storage and in the catalog until the space is needed. At that time they will be expunged, that is, removed from the file system and the catalog. A file which is deleted but not expunged can be undeleted if the need should arise. Files which have been expunged can be reloaded from archival backup if that should be necessary. Users may expunge files explicitly when it is known that the file is of no value or if it contains information best not left in the system.

Thus the users environment is made fail-soft mainly through his own efforts. It is the user's responsibility to maintain an easily reconstructable environment, and to do the actual recovery should that become necessary. The file system provides several facilities to assist the user in managing his environment, but the ultimate responsibility lies with the user.

The preceeding has been concerned with the structure of the system for existing files and how that structure provides fail-soft characteristics. The generation structure has raised the issue of how files are created. When a new file or a new generation of a file is created, Request For Quotation and Bid Response must take place as for any service allocation in the Distributed Computer System. Again, it is most reasonable for an agent process to perform the dialog necessary to create a file, but there is nothing to prevent a user or his process from doing it himself.

When a file is to be created, first the proper catalog must be checked to determine if the file is a new file or a new generation of a previously existing file. If it is a new generation, the volume which contains the current generation will be considered only as a last resort if no other volumes can provide the service. Next an RFQ must go out to all processors containing volumes. After a suitable delay, the bids which have been received are evaluated, the volume contain the current generation is only considered as a last resort. The confirmation and actual request for service then takes place with the chosen volume. The volume performs the necessary storage management, expunging deleted files if necessary, to provide space for the file. Then the volume requests the operating system to create an access process for the file and this access process becomes the channel for all traffic into the file. The volume must also request the appropriate catalog to update itself, reflecting the new file as the current generation. Thus the creation of a new file is carried out in essentially the same manner as any resource allocation in the Distributed Computer System.

Rather than create a new file, a user may wish to add to his catalog a file which already exists in another users catalog. This is not a redundent copy in the second users area, but is true structural sharing; the two file names reference the same file. Changes by one user are apparant to the other user regardless of the name they use, since the files actually refer to the same physical location on the storage media. Since the entries in the volume table of contents contain the fully qualified global name, including the owners name, as well as reference to the catalog which contains that file name, seperate entries must be made in the volume table of contents for each file name, each pointing to the same header record on the storage media. This permits different global names to refer to the same file. However, if either user creates a new generation of that file the other will still be accessing the old generation. Thus a file may be the current generation for one user and an older generation for another. Structural sharing of files with a dynamic generation structure is probably best done only in specialized circumstances.

The issues of protection, access modes and concurrent access are not rigidly fixed by the system structure. In fact, all three areas are very flexible and easily changed.

Traditional schemes for protection vary from explicit listing of all permitted users to reliance on structure in the user names (e.g. account number, user number) to permit or exclude classes of users. Any of these schemes could be implemented with little difficulty. The protection conventions only effect the interface with the volume process. Protection mechanisms should be easy to build in the process, and can be changed with little trouble. Since the user never actually accesses the file directly, but always through an acces process that the volume has initiated, the user has no way of gaining access to a file without the assistance of the volume process.

The types of access modes are also very flexible. The protection for granting initial access based on type of access requested is provided by the volume process as part of the general protection mechanism. To insure all transactions with the file are in accordance with this permission, the access process will check each transaction against the type of access permission granted. Although the system in no way fixes the types of access, the following is a reasonable set for general purpose use. (1) Update-includes read, write, execute (2) Read-includes read and execute (3) Execute-execute only (4) Append-allows only writing on end of existing file.

The access process also provides flexibility for the file's internal structure and the way the data is accessed. Since all access takes place through this process, the structure and access can be sophisticated as desired. It is anticipated that this flexibility will lead to the development of data independent access techniques. When a file is opened, the user will specify how he wants to access that file and regardless of how the file is formated the access process will make the data available in the format the user requests, doing any translation necessary. It is hoped that a proceedural language for operating on data bases can be developed. The access proceedure would then be an interpreter for the language, providing more generalized file operations than the simple read and write.

The access process provides additional flexibility which can be implemented if desired. The process can be used to resolve the problems associated with concurrent access by more than one user to the same file. This is generally no problem if all the users are reading the file, however if one user is modifying the file, the other users might be prevented access until the modification is complete. More sophisticated solutions to the problem exist and can be implemented without difficulty in the access processes.

It would also be possible to allow the user to write prologues and/or epilogues for the access process supplied by the system. Different user supplied routines could be associated with each file, and would allow the user to create a more sophisticated environment. The user routine could be used to provide a sophisticated, interactive protection system much more extensive and flexible than the system supplies. Other possiblities include user supplied special access techniques and internal file structure which might not be available in early versions of the system.

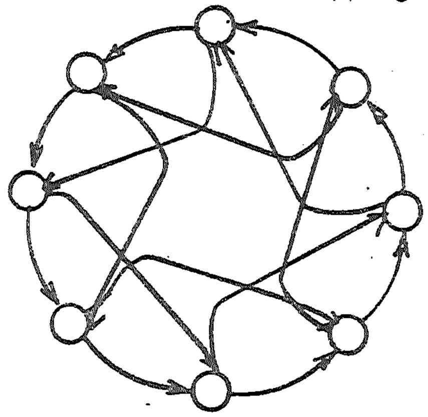
In summary, the Distributed File System is a modular, reliable fail-soft file system for use in the Distributed Computer System. The system is made up of independent modules, the failure of any one of these modules has only limited effect, in fact much of the system will remain completely intact. The file system also provides facilities to help the user create a fail-soft environment for his files, protecting him against system failures and, probably as important, from his own errors. The system is flexible in the areas of protection and access methods, allowing for changes and future extensions without modification of the basic file system structure.

APPENDIX G

Ring Interface

The ring interface must (1) transmit messages onto the ring from the computer it interfaces, (2) recognize messages destined for the computer it interfaces and allow that computer to accept them, and (3) pass on all other messages without change. The following is a summary of the ring interface hardware to implement these functions.

The major consideration is to maintain the connectivity of the ring. The failure of a node must not prevent other nodes from communicating with each other. Several precautions will be taken to prevent this. In addition to the primary data-ring, alternate node-skipping paths will be provided:



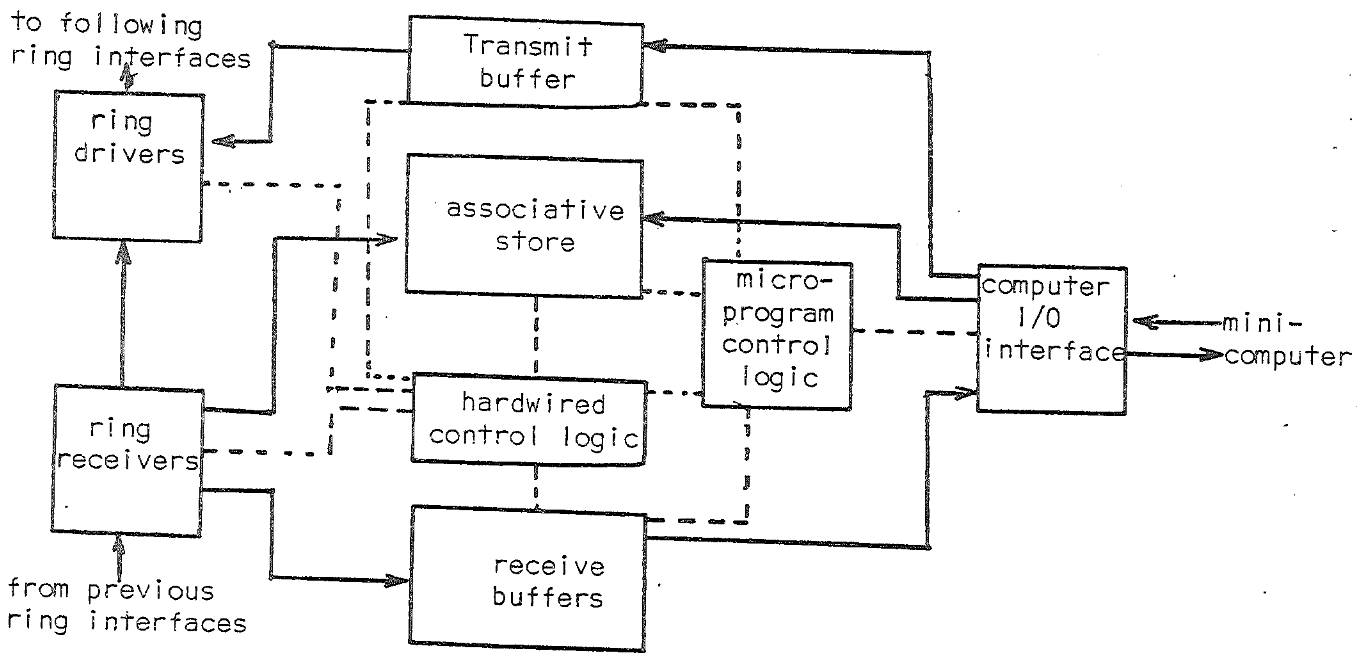
By appropriate switching, complete failure of non-contiguous nodes can be tolerated

Additional steps to avoid the possibility that two adjacent ring interfaces fail in such a way to sever the ring involve a series of bypass data paths in the ring interfaces. A physical path independent of the power-on/off status of the ring interface and computer will allow connection of the ring interface inputs and outputs. This would be used if the ring interface is known to be defective, during maintenance, and when the computer is not programmed to operate on the ring. A path through the logic of the ring interface will also allow unaltered passing (through) of messages.

The normal operation of the ring interface is as follows; When the computer at the node needs to transmit a message on the ring, the message text is transferred to a transmit buffer in the ring interface. When space becomes available the message will be transmitted onto the ring. After the message has traversed the ring, the message will be absorbed and status flags associated with the message will be returned to the processor.

As each message on the ring reaches the ring interface, the destination process name will be compared with a list of names in the ring interface. If a match is found the message will be copied into one of several receive buffers in the ring interface. Subsequently it will be transferred to the attached computer. The list of names in the ring interface can be changed dynamically to reflect the processes currently residing in the attached computer.

The ring interface will consist of the functional parts and data paths shown.



Hardwired logic will control the operations and flows of data closest to the communications ring (left of diagram). A simple micro-programming scheme will control most of the operations and data flows near the computer (right). This will allow flexible adaptation to the requirements of interfacing with different computers and their specific I/O interface requirements.