THE DISTRIBUTED COMPUTING OPERATING SYSTEM

Lawrence A. Rowe

This work was supported by The National Science Foundation under grant GJ-1045, "The Distributed Computing System Project."

TECHNICAL REPORT #66 - June 1975

ABSTRACT

The Distributed Computing System (DCS) is a computer network architecture emphasizing reliable, fail-soft service at a relatively low cost. This paper describes the design of an operating system for a DCS. Issues discussed include interprocess communication, system initiation, and failure detection and recovery. Features of the implementation of a prototype system and some experiences gained from building and using the prototype are also described.

Conclusions made from this work are that problems and solutions discovered while developing minicomputer networks are the same as those encountered in developing networks of larger machines. Specifically, DCS and its operating system demonstrate that systems without centralized control can be constructed, that broadcast messages are useful, and that messages which are sent to a process but are intercepted and acted upon by the environment of the receiving process are necessary to achieve location independence.

INTRODUCTION

The Distributed Computing Operating System (DCOS) is a multiprogrammed, multiple processor operating system designed for the Distributed Computing System (DCS) [FAR73a, FAR75a, HOP73], a geographically local computer network architecture developed at the University of California, Irvine. The design goals for this system are:

(1) to distribute control,

(2) to provide a flexible testbed to perform experiments in computer networking and distributed processing,

(3) to minimize the complexity of programs to be executed by the system,

(4) to execute processes without regard to their physical location,

(5) to use message communication for interprocess communication, i.e., not to allow memory sharing,

(6) to maximize possibilities for detection and automatic recovery from hardware or software failures, and

(7) to minimize the complexity of the operating system.

DCOS is process oriented, that is most operating system services are processes. Within each processor connected to a DCS is a resident software nucleus which provides local resource management (processor scheduling, memory allocation, and servicing of physically connected devices) and interprocess communication services. The nucleus may

allow many processes to execute simultaneously in the processor (multiprogramming) or may be tailored to a single specialized process (uniprogramming).

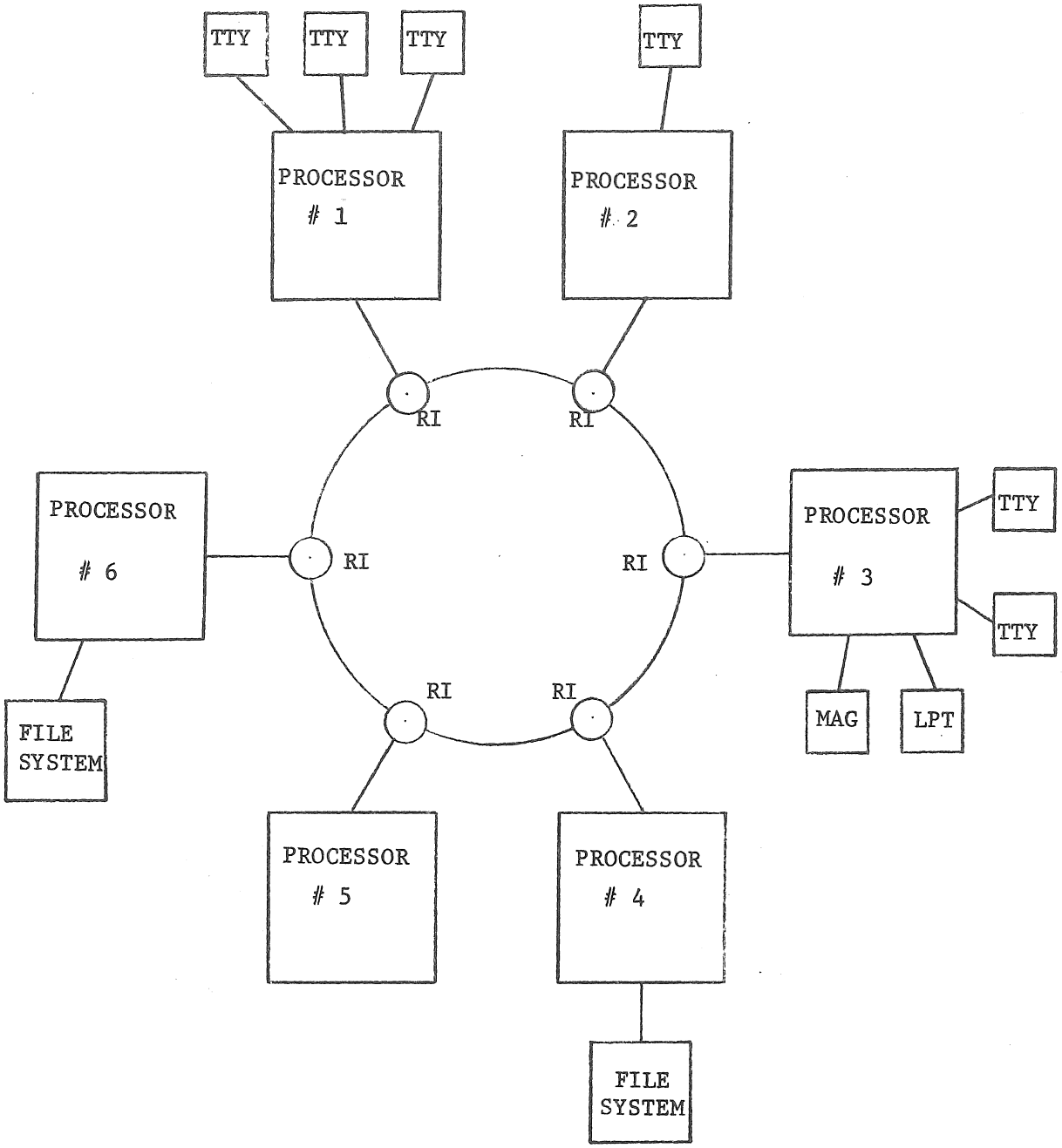
This paper describes the design of the process structure, the interprocess communication facilities, and the system initiation and recovery facilities in the DCOS for a nucleus intended to support multiprogramming. Following this, details of an operational prototype DCS are given. The last section summarizes experiences with the prototype system.

SYSTEM OVERVIEW

Most system services, such as resource allocation, input-output services, and file management, are provided as processes (or collections of cooperating processes) and details of their design are contained elsewhere [FAR72c, LEV73, ROW74]. Processes communicate by sending and receiving messages. To use a service, a process sends messages to and receives messages from the process providing the service.

Figure 1 shows a six processor DCS. Each processor is connected to a unidirectional high speed communication ring by a ring interface (RI). Processors 4 and 6 support file systems which involve physically connected mass storage devices. This example of a DCS is an interactive system; processors 1, 2, and 3 in the figure support terminals. Processor 3 also provides magnetic tape and line printer service.

Messages are directed to a process by name, as opposed to physical location, so that where a particular process resides in the network is not important to the message sender (called "location independence"). Message transmission is accomplished through a combination of hardware and software. Transmiting a message from one process to another causes it to be passed around the ring

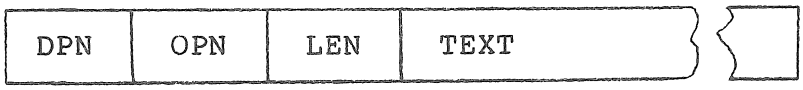


LPT: Line Printer
MAG: Magnetic Tape Drive
RI: Ring Interface
TTY: Terminal

Figure 1: Six Processor DCS

from processor to processor and to be copied into the processor on which the destination process resides by the processor's RI. Each RI has a list of the processes executing in its processor. As a message passes by, a ring interface compares the destination process name in the message with its list of process names, copies those messages for which there is a match into the attached processor, and sets two status bits at the end of the message indicating whether the message was not matched, matched and not copied, or matched and copied. (Actually, each RI "or's" in its value for the two status bits.) The message continues around the ring and is removed by the sending RI which returns the status bits to its attached processor as a response to the send request. Thus, DCS uses an implicit acknowledgment to signal the success or failure of a send request. The format of a message is shown in figure 2 and the possible values of the response status bits and their interpretation are shown in figure 3. (These response status bits are called the "match" and "accept" bits.) Details of the RI and hardware level protocols (control passing scheme) on the ring are described in a paper by Loomis [LOO73].

DCS supports two forms of interprocess communication: one process to one process (process-to-process) and one



where:

DPN Destination process name

OPN Originating process name

LEN Length of message text in characters

TEXT Text of message

Figure 2: Message Format

Match	Accept	Meaning
Ø	Ø	The destination process name was not matched by any ring interface.
Ø	1	The message has been copied by at least one ring interface.
1	Ø	The destination process name was matched by at least one ring interface but not copied by any of them.
1	1	The destination process name was matched by at least two ring interfaces. At least one ring interface copied the message and at least one did not.

Figure 3: Message Transmission Response Status Bits

process to many processes (broadcast). Process-to-process messages are used when two individual processes are communicating with each other. Broadcast messages are used when one process wants to communicate with several processes. Broadcast messages provide a convenient mechanism for supporting two methods of distributing information: maintaining multiple copies of the information or separating the information into disjoint subsets. More details on these two information distribution methods are given in the section on experiences with the prototype.

There are three distinct types of messages: process messages (messages sent by a process which are received by the destination process), control messages (messages sent by a process to the nucleus of the processor on which the destination process is executing), and sequence messages (messages used to manage the logical communication paths between processes). Process messages are used when one process wants to communicate with one or more other processes. This is the standard message type. Control messages are used when one process requests that a nucleus function (such as suspend or start a process) be performed on another process. Because the sending process does not know on which processor the destination process is executing (location independence), it cannot send the message directly

to the nucleus. So, a control message is sent to the destination process requesting the function. The message is intercepted and acted on by the nucleus in the processor on which the destination process is executing. Sequence messages and their use are described in the section on interprocess communication. It is important to realize that all messages, whether process-to-process or broadcast, have a type. For example, a broadcast control message is a control message directed to several processes.

Each nucleus is composed of three processes and a kernel. The three processes are:

(1) a null process -- executes whenever the processor is idle, thereby providing a convenient way to measure this statistic;

(2) a nucleus process -- services nucleus requests (e.g., load process, list names of processes executing in the processor, or terminate process), made indirectly by a control message or directly by a system call (trap to the processor nucleus on which the process is executing); and

(3) a sequence bit process -- described in the section on interprocess communication.

The kernel is that portion of the nucleus which operates in privileged mode and provides processor scheduling, interrupt servicing, message formatting and routing, event handling, and memory management. On those machines to which input or output devices are physically connected, there is also an

input/output handler process which controls the devices.

Within a DCS there are a number of other processes. These include command processes (the monitors or executives to which terminal users or batch streams are connected), resource allocators (processes that manage system resources via the bid-request scheme [FAR73a]), status checkers (processes that monitor the status of the system and its resources [ROW73]), record-keeping processes (e.g., sign-on, accounting, and measurement processes), and application processes (e.g., text editors, file directory listers, file copiers, language processes, and text preparation processes).

One purpose of this project is to investigate a system architecture that provides high reliability. DCS attempts to minimize the probability of undetected errors and to maximize the possibility for recovery from errors. This is achieved by distribution (a combination of separation and redundancy of system components), isolation (keeping local failures from spreading) and dynamic reconfiguration. System architectures designed in this way admit the possibility that a failure, whether due to hardware or software, may interrupt the service to a subset of the active users while minimizing the possibility of interrupting service to all active users. A system

exhibiting this behavior (called "fail-soft") requires that there not be a critical component, either in the software or hardware. For a functioning DCS this means that there must be more than one copy of the command process, resource allocator, status checker, and other selected system processes.

PROCESS STRUCTURE

This section describes the components of a process, process names, process creation and destruction, and the system calls provided by the nucleus.

The Components of a Process

A process is composed of a task control block (TCB), a context block (CB), and a program segment. The TCB contains process information needed by the system all the time, such as its name, execution status, message queue, program segment descriptor, execution statistics, and file directory descriptor. This information can not be changed by the process.

The CB contains two kinds of information: (1) information the process may change (e.g., name of initializing process, name of process to notify on termination, and descriptor of the terminal, if any, connected to the process); and, (2) information not needed by the system if the process is in a dormant state (e.g., machine state when last interrupted and receive message buffer descriptor).

The program segment contains the actual program code and data space.

Process Names

Process names in DCS are of the form

CLASS.NAME.PROCESSOR.ID.SEQUENCE

where:

CLASS	is the class (e.g., SYS or USER) the process belongs to,
NAME	is the program name (e.g., BASIC, QED, or RUNOFF),
PROCESSOR	is the type of processor (e.g. Lockheed SUE or Varian DATA 620/i),
ID	is the particular processor identification number, and
SEQUENCE	is a sequence number on PROCESSOR.ID.

A process name is created at process creation time and thus reflects the place the process was initiated, not necessarily where it is presently executing. Notice that the triple <PROCESSOR,ID,SEQUENCE> guarantees uniqueness of names throughout the system. Examples of process names are: SYS.RA.SUE.1.5 (the resource allocator process RA executing on a Lockheed SUE, in particular the fifth system class process initiated on SUE number one), and USER.TXT.DATA620/i.3.52 (a user class process TXT executing on Varian DATA 620/i number three, the fifty-second process initiated on that processor).

To send a process-to-process message, the process name

of the destination process is used. To send a broadcast message, a general name is used. A general name is a process name in which one or more fields in the name are marked to match any possible value. For example, SYS.RA.*.*.* is a general name used to broadcast to all resource allocators.

Name representation in the prototype system is constrained to 16 bits. Because this is not enough bits to represent a complete name, a shortened representation is used in the prototype to encode the process name. (Details of the prototype system and the process name encoding are described in a later section.)

Managing Collections of Processes

Facilities for creating and managing collections of processes are primitive. (Another project goal is to investigate process organizations to achieve improved performance and reliability.) The present facilities result in a nonhierarchial process structure (as opposed to the rigid hierarchical structure described by Dijkstra [DIJ68] and Brinch Hansen [BRI70]).

Associated with each process are the names of two other processes: the initiating process (INIT) and the notification process (NOTIF). When a process wants to create a new process, it sends a request to do so to a

resource allocator agent, who selects a machine on which to initiate the process and causes it to be initiated with INIT for the new process being set to the name of the requestor and NOTIF set to INIT unless otherwise specified in the create request.

A process can be terminated by its own request, by a request from the process named in INIT, or by an authorized system process. Upon termination, all resources bound to the terminating process are released, and a notification of the termination is sent to the process named in NOTIF. Should the notification process not exist, the message is sent to the process named in INIT. Termination of a process does not imply that processes it created are terminated. They continue to execute until their eventual terminations. Descendant processes request and release resources on their own, and upon termination their resources are returned to the free pool as opposed to being passed to their initiator.

System Calls

There are four system calls provided by the nucleus: send message, receive message, terminate process, and read time.

To send a message, a process issues a send message system call passing the destination process name and a reference to the message text to the nucleus (actually the

system call server is part of the nucleus kernel). Packeting, retries on errors, and sequencing are handled by the nucleus interprocess communication facilities as described in the next section. After completing the transmission, the nucleus returns the response status bits, described in figure 3, to the process.

Message sending in DCOS is asynchronous, that is, before a process can request that another message be sent, the previous transmission must be complete (i.e., the response status bits returned). This means that a process is blocked when a send message call is made and unblocked after the transmission is completed. Asynchronous communication was chosen for two reasons. First, this results in a conceptually simpler system from a user's point of view; and second, the operating system is less complex. By contrast, if synchronous communication were allowed, complex protocols and conventions would be necessary. For example, to handle situations such as might occur if four send message requests are made and a transmission error occurs on the second. A convention about what to do with the two queued requests (e.g., perform regardless or abort) must be adopted and a protocol for notifying the process about the transmission response status must be developed. To avoid these complications, DCOS uses asynchronous

communication.

To receive a message, a process issues a receive message system call, passing a descriptor for a message buffer (location and size) to the nucleus. The first message in the process's message queue, if one is present, is copied into the message buffer in the format shown in figure 2. The destination process name is copied so that a process can determine whether the message was sent to its process name or its broadcast name. When making a request to receive a message, a process may specify the specific process from which it wants to receive a message and may specify a time after which, if a message has not been received, control is returned to the process.

The third system call is terminate process. A process uses this call to terminate itself.

The last system call is read time. This call returns the local processor date-time block or system clock (hardware dependent local clock).

INTERPROCESS COMMUNICATION

As described in the previous section, a process requests that a message be transmitted to another process (or processes in the case of a broadcast message) by issuing a send request call and requests that a message be received by issuing a receive request call. After the originating process requests the transmission, the nucleus assembles the message in a message buffer. If the destination process exists on the same processor as the originating process, the message buffer is placed on the nucleus input message queue for the processor. Thus, the message is not needlessly sent around the ring. (However, all broadcast messages must be sent around the ring.) If the message is to be sent around the ring, it is placed on the nucleus output message queue. The output message routine issues the output request to the ring interface. The receiving processor ring interface copies the message into a message buffer which is placed on the receiving nucleus's (or nuclei's) input message queue. The input message routine then places the message on the message queue for the destination process. When the process requests a message and the message is at the front of its message queue, it is copied into the receive message buffer in the process address space. Figure 4 shows two examples of message transmission: an interprocessor transmission and

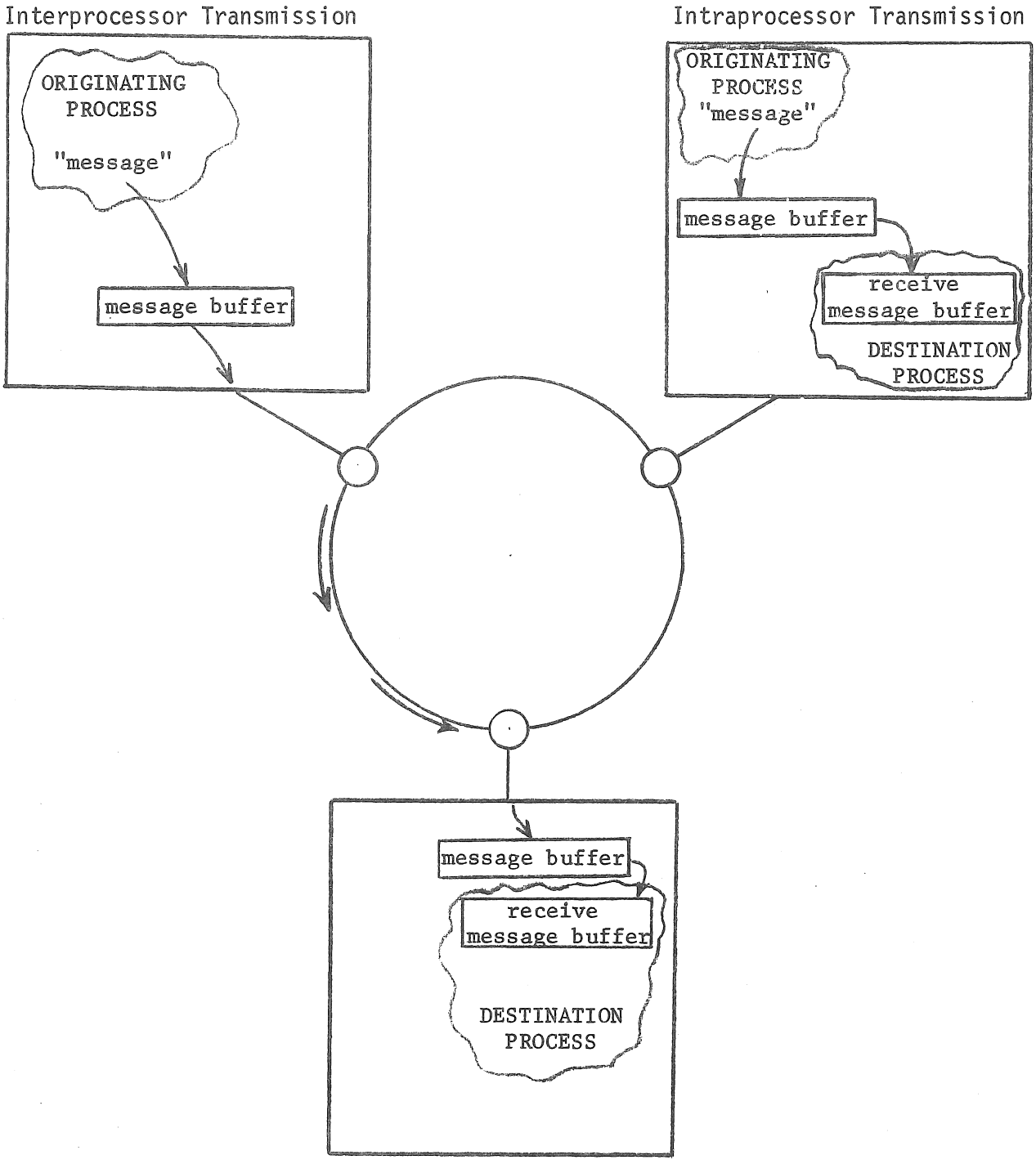


Figure 4: Examples of Message Transmission

an intraprocessor transmission.

This section describes how the message is actually transmitted from the originating process to the destination process. It describes message packeting, detecting and recovering from transmission errors, and managing the logical communication paths between processes. These communication system mechanisms are transparent to a process sending and receiving messages. This is followed by a brief discussion of higher level communication protocols.

Message Packeting

In order that certain system resources, namely message buffers and ring transmission capacity, can be equitably shared by all processes, messages are transmitted in packets if they are longer than a fixed parameter (determined at system creation time). Since packeting and reassembly are performed by the nucleus, the processes involved are unaware that it happens. In the discussions that follow, the term "message" is used instead of "message packet", because it makes the explanations easier to understand.

Detection and Recovery from Transmission Errors

There are three types of abnormal transmission conditions, categorized by the way they are detected: (1) inability to transmit, (2) transmit overrun and cyclic

redundancy check (CRC), and (3) match/accept errors.

An inability to transmit arises when the ring level protocol is breached, violated, or disrupted. To insure that only one ring interface places a message on the ring at a given time, a control passing scheme is used [LOO73]. A "control token" is passed around the ring signifying which RI has control. A ring interface may transmit a message only when it possesses the control token. After it places one message on the ring, the RI passes the control token to the next RI. (Notice that this guarantees that one ring interface does not hold control of the ring transmitting one message after another for extended periods of time.) Upon being requested to send a message, a ring interface waits for the control token. The RI could wait for the token indefinitely. However, the nucleus detects abnormally long waits by establishing a time after which, if the message has not been sent (or some other error has not been reported), the control token is presumed lost. At this point the nucleus forces another token onto the ring. If the ring protocol cannot be reestablished due to ring failure, the processor continues as best as it can (probably at a considerably reduced functional capability). Notice that any processor on the ring can restore the token. Thus, control is distributed as opposed to being centralized. A

centralized control would result in a more vulnerable system. To insure that only one processor at a time forces a control token, the detection time out plus the time needed to regenerate a token is significantly different on each processor.

The second type of transmission abnormality is a transmit overrun or cyclic redundancy check. A transmit overrun condition is signaled by a ring interface when it is unable to fetch words from memory fast enough to maintain the ring transmission rate (approximately two million bits per second). A cyclic redundancy check signal by a ring interface means that the message received after being passed around the ring is different than the message sent.

The third type of transmission abnormality is a match/accept error. This situation is signaled by the ring interface in the response status bits returned to the processor after the message has traveled around the ring and has been removed by the sending RI.

For each of these last two types of transmission abnormalities, the error is counted and the transmission is attempted again. This continues until a fixed number of consecutive unsuccessful retries (threshold number of retries) is surpassed after which the recovery mechanisms described below are invoked. Each type of error is counted

separately and for each there is a different retry threshold.

Transmit overrun or cyclic redundancy check is probably a transient condition so the retransmission is attempted immediately. If the retry threshold is violated, the sending process is returned a no-match-no-accept.

For a no-match-no-accept (i.e., process name does not exist), the message is also immediately retransmitted. For match-no-accept and match-accept (i.e., at least one RI matched the name but could not copy the message), the processor that could not copy is probably saturated. That is, either the attached RI is not initialized for input because there are no message buffers available or the RI is unable to store into memory fast enough. Because we expect that this situation may persist for a short period, the message is not immediately retransmitted. It is placed on the end of the output message queue. However, if the retry threshold is violated the sending process is returned the response status bits.

Managing Logical Communication Paths

In any communication system, if a message is sent and the response signal returned by the receiver is either not received by the sender or unintelligible to the sender, the sender can not determine whether the message was received.

To resolve this problem, message communication systems typically retransmit the message (along with some sequencing information so that the receiver can ignore copies of previously received messages) until an acceptable response is received by the sender or the transmission is abandoned.

In DCS for example, as shown in figure 5, when transmitting a message from processor 1 to processor 2, a transmission error could occur either before the message has arrived at processor 2 or after the message has passed processor 2. Suppose the error occurs after the message has passed processor 2 (labeled error 2 in the figure). In this case the message may already have been copied into processor 2. Because processor 1 cannot distinguish between the two errors, it retransmits the message with the sequence bit indicating that this is a copy of the previous message. When processor 2 receives the second message (a copy), the nucleus knows whether to ignore it by comparing the sequence bit in the message with the sequence bit from the previous message. On the other hand, suppose the error occurs before the message has passed processor 2 (labeled error 1 in the figure). In this case the message has not been copied into processor 2. (Actually, the RI copies the message and signals to the nucleus the arrival of a message with a cyclic redundancy check error, so the nucleus ignores it.)

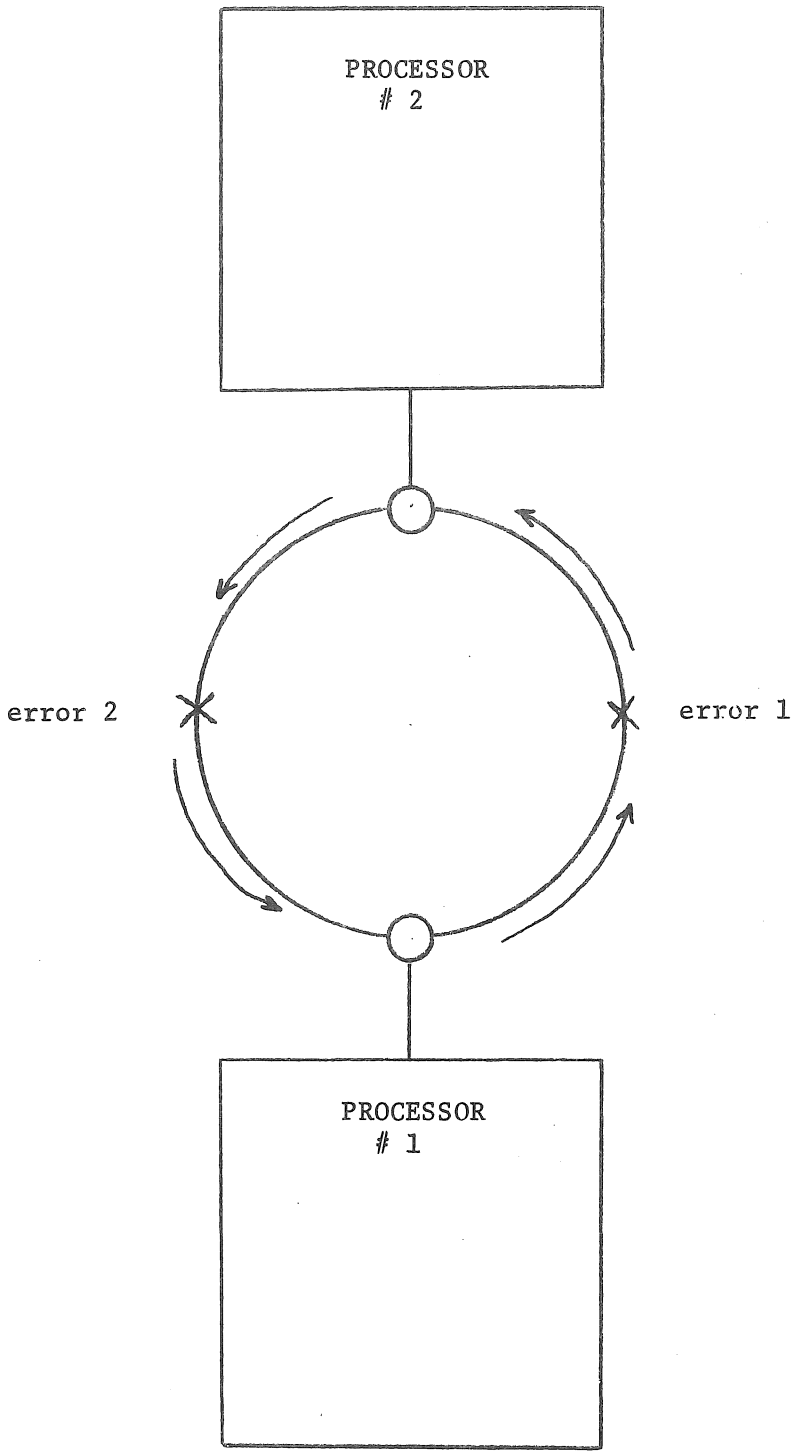


Figure 5: Possible Locations of Transmission Errors

When processor 2 receives the second message (not a copy), the nucleus knows to accept it by checking the sequence bits.

Sequencing of messages must be done at the process level rather than the processor level. This is because DCS uses location independent naming and a process might be moved (from one processor to another) between when the first message is sent and the retransmission is sent.

The communication system can be thought of as providing logical communication paths between processes (either a process-to-process or a broadcast path). The kernel maintains tables describing the status of each logical communication path for each process executing in its processor. A path is a one-directional communication link. Sequencing information for a path is maintained in a sequence bit table both at the sending and receiving end of the path. There are two sequence bit tables for each process: a send sequence bit table and a receive sequence bit table. Figure 6 shows two processes and their associated sequence bit tables. Notice that in process A's send sequence bit table, the sequence bit for process B is Ø and that in process B's receive sequence bit table, the sequence bit for process A is 1. When process A sends a message to process B, the send sequence bit for B (Ø) is

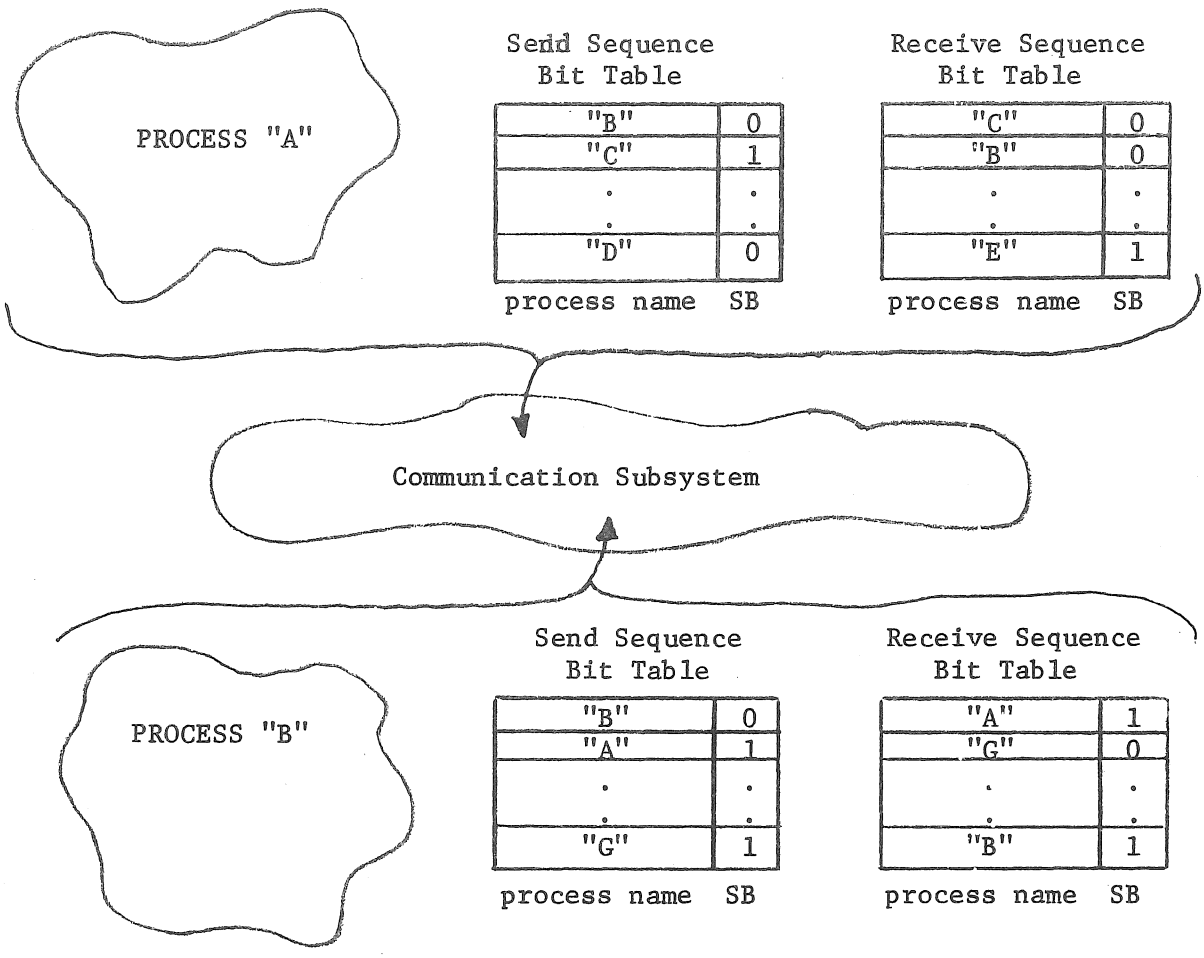


Figure 6: Sequence Bit Tables

placed in the message. After the message is received at process B, the sequence bit in the message (Ø) is compared with the receive sequence bit for A (1). Because they are not the same (Ø ≠ 1), the message is accepted and the receive sequence bit table entry is replaced by the sequence bit in the message. If a message arrives in which the sequence bit in the message matches the seguence bit in the table, the message is ignored because it is a copy of the previous message received. The sequence bit in process A's send sequence bit table is flipped (Ø -> 1 and 1 -> Ø) after the message has been successfully transmitted.

The process name in a sequence bit table must be exact, either a process name or a general name. These two cases must be kept distinct because the communication path between a process and a collection of processes (broadcast name) is distinct from the path between a process and a specific member of the collection.

Initialization of sequence bit table entries is straightforward. On the sending end, a new entry is created and the send sequence bit is set to Ø or 1 (the choice is arbitrary). On the receiving end, if the originating process name in the received message is not found in the receive sequence bit table, the message is accepted, a new entry is created, and the receive sequence bit is set to the

sequence bit in the message in the normal way. (This is not exactly correct. Actually, a synchronization message is sent before the first message. The need for this synchronization message and how it is accomplished is described in a later section on initiation and recovery.)

A process executing in DCS may communicate with an arbitrarily large number of other processes which implies that there may be many communication paths active at one time. Because sequencing information is maintained for each distinct path and because there are physical resource (i.e., memory) constraints on the total number of paths on which information can be maintained at one time, there must be a mechanism for closing communication paths. This problem is analogous to that of mapping virtual memory into physical memory in a virtual memory system and many of the same difficulties are encountered.

A close path function can be initiated from either the sending or receiving end of a communication path. First, consider a close path function initiated at the the sending end of a path. To close the path, the receiving end process name must be removed from the sender's send sequence bit table and the sending end process name must be removed from the receiver's receive sequence bit table. The nucleus at the receiving end of the path must be notified by the

sending end nucleus to close the receive end of the path. This is accomplished by sending a special type of message, called a "sequence message", to the process on the receiving end which is intercepted and acted upon by the receiving end nucleus. After the sequence message is received, the sending end nucleus can remove the receiver's name from the sender's send sequence bit table. Second, consider a close path function initiated at the receiving end of a path. As in the previous case, to close the path, the sender's name and receiver's name must be removed from, respectively, the receiver's receive sequence bit table and the sender's send sequence bit table. The nucleus at the receiving end of the path notifies the nucleus at the sending end of the path to remove the receiver's name from the sender's send sequence bit table by sending a sequence message. In this case, the receiving end nucleus can not remove the sender's name from the receiver's receive sequence bit table after the sequence message is received. This is because a message could be sent along the path by the sending end process between when the sequence message is received by the sending end nucleus and when it is acted upon. For this reason, the receiver's receive sequence bit table entry is not removed until an acknowledgment sequence message is sent by the sending end nucleus indicating that the other end of the path is closed.

The remaining paragraphs in this subsection describe in more detail exactly how these close path functions work. Readers not interested in these details should skip to the next subsection on higher level protocols.

A close path function is initiated at the sending end when: (1) a process requests that a message be sent, (2) a communication path does not exist between this process and the destination process (i.e., the destination process name is not in the sender's send sequence bit table), and (3) all communication paths for the sending process are in use (i.e., the send sequence bit table is full). A close path function initiated at the sending end consists of:

(1) blocking the process requesting the send,

(2) locking the process's send sequence bit table,

(3) selecting a path to close,

(4) sending a sequence message to close the other end of the path,

(5) freeing the entry in the send sequence bit table,

(6) unlocking the send sequence bit table, and

(7) unblocking the process requesting the send.

When a nucleus receives a sequence message request to close a particular process's receive end of a communication path and the path is defined (i.e., the sending process's name is in the receiving process's receive sequence bit table), the

path is closed by freeing the receive sequence bit table entry. If the path is not defined, no action is taken by the nucleus.

A close path function is initiated at the receiving end when: (1) a message is received for a process (i.e., the input message routine processes a message on the input message queue), (2) a communication path does not exist between the originating process and the destination process (i.e., the originating process name is not in the receiver's receive sequence bit table), and (3) all communication paths for the destination process are in use (i.e., the receive sequence bit table is full). A close path function initiated at the receiving end consists of:

(1) locking the destination process's receive sequence bit table,

(2) selecting a path to close,

(3) sending a sequence message to close the other end of the path,

(4) waiting for an acknowledgment sequence message indicating that the other end of the path is closed, and

(5) unlocking the receive sequence bit table.

When a nucleus receives a sequence message requesting that the sending end of a particular path be closed, the path is closed unless the sending end process has a message

transmission pending or its send sequence bit table is locked. After closing the path (i.e., removing the receiver's name from the sender's send sequence bit table), an acknowledgment sequence message is sent by the sending end nucleus to the receiving end nucleus indicating that the receiver's end of the path may be closed. This acknowledgment message is sent even in those cases when the sending end process does not exist (it may have terminated) or the path is not defined (the sequence message may be a copy), so that the receiving end table can be unlocked. In both cases, when the close path function is completed, the nucleus can proceed with processing, either a send system call or the message on the nucleus input message queue.

Locking the sequence bit table prohibits the nucleus from initiating another close path function on a table. So, in the case of a receive sequence bit table, messages on the input message queue processed by the input message routine which have a sequence bit table entry are accepted and, if appropriate, passed to the process. Furthermore, a process continues to execute if the function is a receive end close path. Notice also, that a receive end close path function and a send end close path function may be performed simultaneously on a process.

Sequence messages are similar to control messages in

that they are addressed to a process but are acted on by the nucleus of the processor on which the destination process resides. They are different than control messages in that they are not sequence checked. In other words, process messages and control messages have sequence bits which are checked to insure that a copy of a message is not processed. By contrast, if sequence messages were sequence checked, then sequence bit tables must be maintained for sending and receiving sequence messages. This implies that a close path function may have to be performed in order to send sequence messages thus resulting in an infinite recursion. Because sequence messages are not sequence ckecked, multiple copies of one may be received by the destination nucleus. The close path function protocols are designed so that receiving multiple copies of the sequence messages does not disturb the integrity of the communication system. For the same reason that sequence messages are not sequence checked, a special process, called the "sequence bit" process, is included in every nucleus to perform the communication path management functions. This process is the only one which sends and receives (indirectly) sequence messages.

Both close path functions require that a communication path be selected for closing. It would be desirable, for efficiency reasons, to select a path over which

communication is finished or will not be resumed for some time. This problem is analogous to that of selecting a page to remove from physical memory in a paging system. Like the page removal problem, there is no good algorithm for making the removal selection. The page removal problem is handled by using a heuristic which performs better than random selection. In the path closing problem, several heuristics are suggested: least number of messages sent, least recently used path, oldest path (first in-first out), or newest path (last in-first out). There does not appear to be a particularly good a priori reason for selecting one of these heuristics over another so an answer to this problem is unknown. This issue is discussed further in a later section.

Suppose the nucleus at the opposite end of the communication path selected for closing cannot be communicated with via a sequence message, e.g., no-match-no-accept, match-no-accept, or match-accept (the path being closed may be a broadcast path in which case it has a single sender and multiple receivers). Because a no-match-no-accept means the destination process does not exist, the other end of the path is presumed closed (i.e., the destination process has terminated and thus cannot be involved in future communication) and the close path

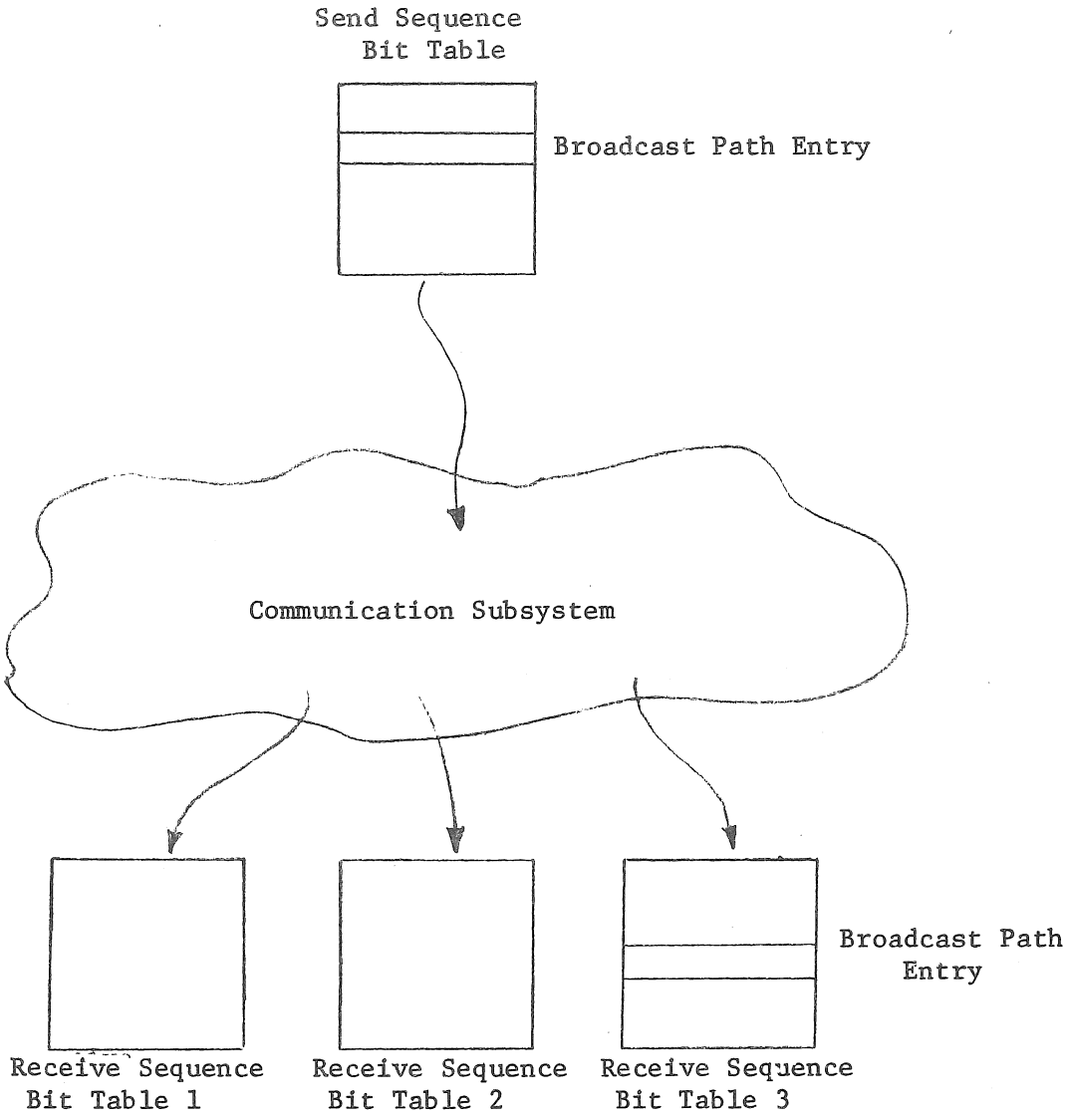


Figure 7: Sequence Bit Table State after Broadcast Sequence Message

function continued. For a match-no-accept, no nucleus received the sequence message, so another path is selected for closing. (Note, a path is not closed until the function frees the sequence bit table entry.) Match-accept causes problems because some nuclei received messages and some did not. In this case the message is retransmitted and, if the situation continues after a small number of retries, no further communication is attempted. Figure 7 shows the state of the sequence bit tables after the broadcast sequence message has been sent and acted upon by the receiving nuclei. Notice that the broadcast path entry has been freed in receive tables 1 and 2. The question is whether to free the entry in the send table. If it was a receive initiated close path function, the send table entry can be retained and the next time communication along this path is attempted all processes will receive the message correctly (processes without receive table entries will create a correct one and processes with an entry are already synchronized with the sender). (Notice that the receive sequence bit table would be locked indefinitely if the nucleus requesting the close path function is the one that does not receive the close path acknowledgment, the process with receive sequence bit table 3 in the example. Thus, a time out is set for the sequence bit table after which the

sending end is presumed closed and the close path function continued.) If, on the other hand, it was a send initiated close path function, there are two options: the entry can be freed or another entry selected for closing. If the entry is freed, the next time communication along the path is attempted the synchronization message (described in the next section) will cause the remaining receive table entries to be freed.

Because it results in less complexity, both from the user's and the system's viewpoint, communication path closings are invoked implicitly as opposed to explicitly, either by the processes involved or by the system when a process terminates. Both alternatives lead to problems when a message is sent or received and there are no sequence bit table entries available. Also, in the case of user invoked closings, more detailed implementation knowledge is required on the part of users. Implicitly invoked close path functions were chosen because they would be needed even if explicit functions were implemented.

Higher Level Protocols

DCOS does not presume any protocol or structure on the text portion of messages. Processes communicating among themselves may establish any conventions they desire. Nevertheless, some processes use a standardized protocol in

which a message requesting a function and arguments has the form "identification, function, argument,..., argument" and a message responding with the results of a requested function has the form "identification, response, result..., result". The "identification" is a symbol (supplied in some cases by the requestor and in others by the responder) useful for maintaining the distinction between simultaneous events being acted upon by two processes. For example, a requestor may supply an identification in a request message so that the response message (with the same identification returned by the process providing the function) can be distinguished as the one for the particular function request. More details and examples of the use of this higher level protocol are available in the "DCOS Programming Guide" [ROW74] (specifically section 3.4 on the input/output handler).

INITIATION AND RECOVERY

This section describes how a DCOS is initiated and how detection of and recovery from nucleus or process failure are handled. Four design goals were established for the initiation and recovery procedures. They should:

(1) not require a centralized control or source of information,

(2) maximize similarity of nuclei and common system processes (such as input/output handlers and resource allocators),

(3) allow dynamic reloading of nuclei and processes with minimal disruption of the system, and

(4) minimize resident memory space of initiation and recovery procedures.

Much of the material on nucleus and process failure detection and recovery is taken from a previous paper [ROW73].

System Initiation

Processors are divided into two classes for system (nucleus and system processes) loading purposes, those with a local information source (any physically connected device which allows access to a copy of the nucleus, such as disk, magnetic tape, card reader, or paper tape) and those without a local information source. Processors with a local source may load a nucleus directly or may be loaded across the ring. Those without a local source must be loaded across the ring.

This is accomplished by executing a ring loader which clears the ring interface name table (may first have to initiate ring) and communicates with a file system (probably a special ring load process) to load a copy of the nucleus. In keeping with the goal of minimizing resident memory space of the initiation and recovery procedures, the ring loader is bootstrapped across the ring into the processor as shown in figure 8. Thus, the ring bootstrap is kept small to make loading it easy and to minimize the probability of it being destroyed. (In a production version of DCS, the bootstrap could be "wired-in", either in the ring interface or in the processor.) Obviously, processors cannot be loaded remotely until at least one processor has been loaded directly. Otherwise, no restrictions are placed on the order in which processors are loaded.

All nuclei, for a particular type processor, are identical except for the machine identification number, interrupt device tables, and the input/output handler (if present). The machine identification number is passed to the nucleus by the nucleus load function (either a direct or ring load). The interrupt device tables bind physical device numbers to device type. The problem is how to initialize the device tables without maintaining special instances of the software. The same problem occurs with input/output

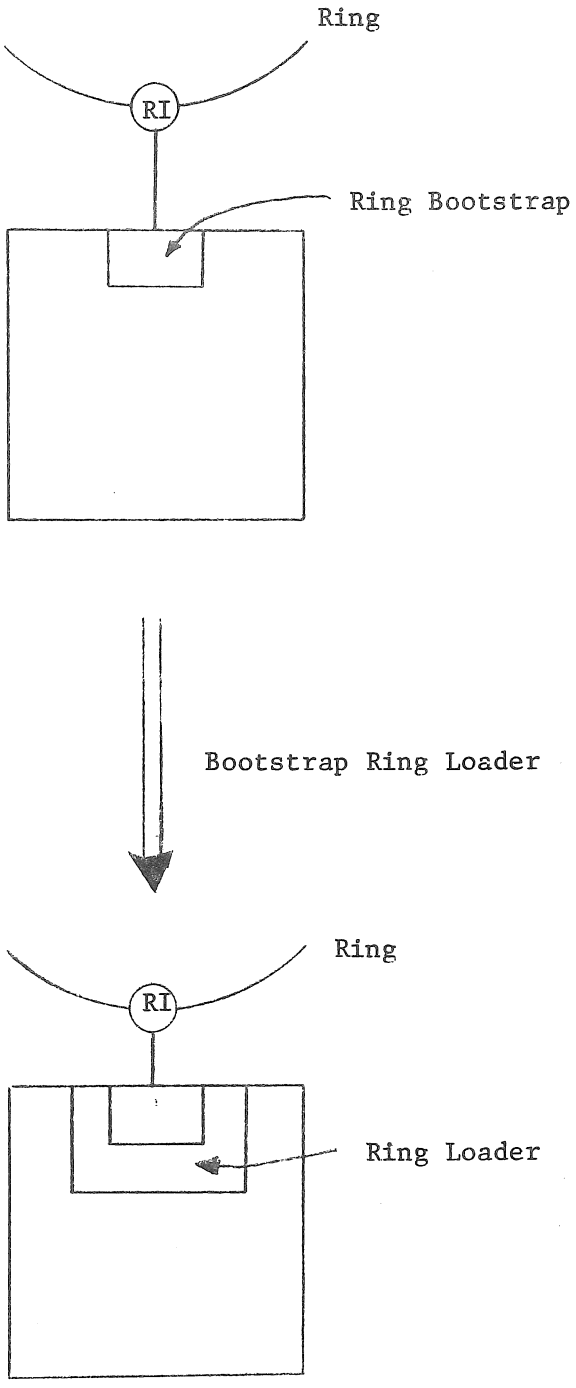


Figure 8: Bootstrapping the Primitive Ring Loader

handlers. Several solutions are possible, such as predefining all machines of a particular type to use the same physical device numbers for similar devices, compiling nuclei separately with appropriate device table definitions, compiling all possible device tables (for a set of processors) into one copy of the nucleus and selecting the appropriate table at initiation time, or establishing a more restricted initiation ordering so that the tables can be read from a file. Each of these solutions has advantages and disadvantages and the one chosen depends on the particular implementation environment. Another solution requiring special hardware, available on some third generation processors, is to provide facilities for the system to interrogate its environment to determine what physical devices exist and what their physical addresses are.

Each nucleus is loaded with a start process which initializes local resources and data structures. The start process may also load some additional system processes (e.g., command process, resource allocator, or status checker) depending on the DCS configuration. (Notice that the information location problem discussed in the previous paragraph arises here also.) The process configuration (the number of instances of each type of system process and their distribution) strongly influences the performance and

reliability of a system. This theme is elaborated on in a later section on experiences with the prototype. There certainly must be one (and probably more than one so as not to centralize control) operator or configuration control process for externally monitoring and changing the distribution and number of system processes. This is distinct from a status checker whose function is to monitor the maintenance of a minimal configuration, to do simple load balancing, and to insure that nuclei are still functioning.

Ring initialization is handled by the same detection and recovery mechanism described previously, i.e., some process attempts to send a message and, if the ring times out, a new token is generated.

Nucleus Failure Detection and Recovery

Status checkers test periodically for processor failure by sending to each nucleus a message requiring a response. If a nucleus fails to respond to several consecutive status check messages, the status checker hypothesizes that the nucleus has failed. A single status checker cannot establish that a particular nucleus has failed. However, if a given percentage of checkers decide that a nucleus has failed, recovery procedures are initiated.

Recovering from a nucleus failure is accomplished in the same way the nucleus was initially loaded, either directly

from a local information source or indirectly from a remote source. The only difference between recovery and initiation is that recovery is invoked by a status checker while initiation might have resulted from an external impetus (for example, an operator). Nevertheless, particular instances of a DCS may elect to require human intervention in recovery procedures.

Nucleus failure and, in very limited cases, process failure, lead to a peculiar problem concerning sequence bit synchronization. Suppose two processes, A and B, are communicating, and process A is in a processor that fails then process A fails. When the nucleus is reloaded, if process A was a necessary system process it will be restarted with the same name (e.g. input/output handler, nucleus process or resource allocator). Suppose the new copy of process A sends a message to process B. Because B does not know A failed and was restarted, their communication path may not be synchronized and the message may be discarded. To resolve this problem, when the first message is sent to a process, a close path function is initiated at the sending end to synchronize the logical communication path. Notice that this also solves the problem described in the previous section wherein a broadcast path is not completely closed (i.e., one or more of the receive end processes do not close

the path but the send end is closed).

Status checkers also periodically check that processes bound to resources (defined in the input/output handler device reservation tables) still exist. This prevents resources from being lost when a nucleus fails.

Process Failure Detection and Recovery

Software failures in a process are detected by traditional hardware failure indicators, e.g., an attempt to reference an undefined or protected address or an attempt to execute an undefined or protected instruction. There are several actions the system can take when a process failure is detected:

(1) save a copy of the process environment,

(2) initiate a test process,

(3) initiate a new copy of the failed process, or

(4) take no action until directed to do so by an external source.

The particular action taken depends on what process failed and the circumstances causing the failure.

PROTOTYPE DETAILS

This section describes details (as of February 1975) of the prototype DCS developed at U. C. Irvine and discusses the directions being pursued.

Hardware

The present system is composed of three processors, Lockheed SUE minicomputers, connected by ring interfaces to a data ring operating at 2.3 megabits. There are also two Varian 620/i's with IBM 2314 class disk drives attached; these 620/i's are currently connected to SUE's providing a rudimentary file system capability for the DCS. There is a modest complement of peripherals including: several terminals (teletypes and alphanumeric and graphic displays), an Addmaster and a Remex paper tape reader, a Calcomp plotter, a Kennedy magnetic tape, a Tektronix 611 storage scope, a low speed Centronics line printer, and a text preparation facility (Diablo Hytype printer and a high speed upper/lower case Data Products printer). In the near future the Varian 620/i's will be connected directly to the ring resulting in the configuration shown in figure 9. (A Computer Automation ALPHA-LSI-2 controls the text preparation facility.)

The ring interfaces are constructed using TTL circuitry. Each interface provides 16 names, each 16 bits long, in an

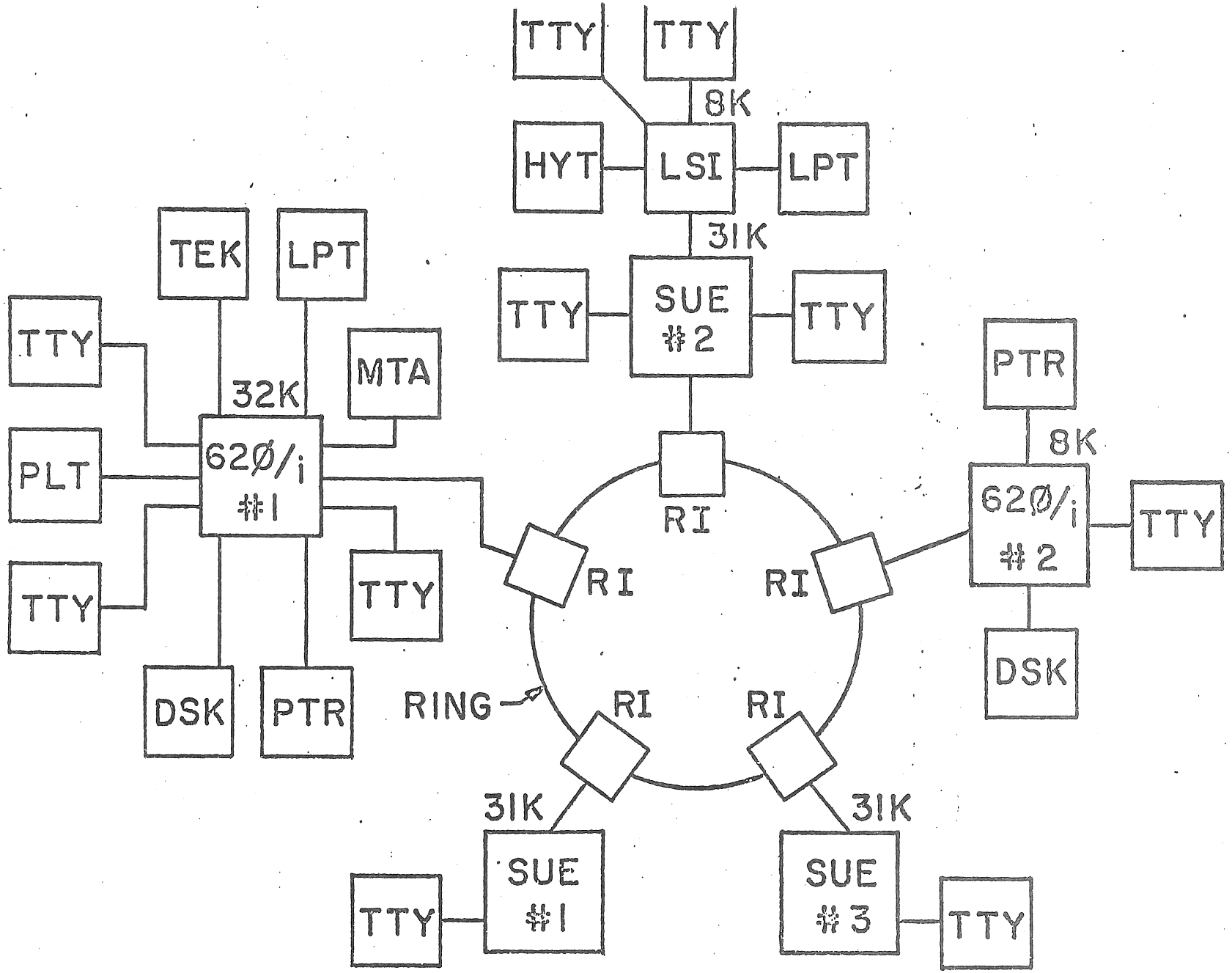


Figure 9: DCS Configuration

associative store. The associative store is implemented by a bit serial associative memory. The hardware does not support broadcast name matching so the software system uses 2 names in the associative store for those processes which may be broadcast to, one for the process name and one for the broadcast name. The fabrication cost of a ring interface is $1000, in quantities of 1. This cost would be cut in half if they were produced in larger quantities (approximately 10). It is estimated that they could be produced for $100 each using large scale integration (LSI) technology [FAR75b].

Software

The current version of the operating system consists of nuclei for each processor, sign-on processes, command processes, input/output handlers, and system status and statistics collection processes. User or application level software available includes an assembler, machine oriented language compilers (MOL620 [HOP71] and MOLSUE [HOP75]), linkage editors, a file transfer process, a file directory listing process, a line oriented text editor (QED), a text outputting system (RUNOFF), a debugger for distributed processes [SOW74], and other utility and diagnostic processes. Status checkers, complete resource allocators, and the fully distributed file system [FAR72c] have not been implemented.

A hardware name is obtained from a process name when it

is created. This is because the RI hardware only allows 16 bit names. The format of a hardware name and how it is encoded from a process name are shown in figure 10. Certain system processes have predefined names:

1.i.1 nucleus process

1.i.2 input/output handler

1.i.3 command process

1.i.4 sequence bit process

where i is the processor number. The existence of reserved system process names for input/output handlers and command processes does not imply that all of these processes exist on every possible processor. Different software configurations of the system may include some but not all of them.

When transmitting messages between processes, the system includes a message control word. The control word contains three fields, as shown in figure 11: a packet definition field, a sequence bit field, and a message definition field. The packet definition field indicates whether this packet is the first, neither first nor last, last, or first and last packet of the message. The sequence bit is used to determine if this packet is a copy of the one previously sent. The message definition field describes whether the message is a process, control, or sequence message. The control word is

CLASS.NAME.PROCESSOR.ID.SEQUENCE

CLASS.ID.SEQUENCE

where:

$0 \leq$ CLASS ≤ 15	Process class:	
	0	not used
	1	system processes
	2	reserved for resource allocators
	3-8	reserved for system process classes
	9	user processes
	10-15	reserved for user process classes
$0 \leq$ ID ≤ 15	Processor number	
$0 \leq$ SEQUENCE ≤ 255	Sequence number	

Figure 10: Hardware Names

CTLW contains:

PDF Packet definition field:

last packet bit	first packet bit	
Ø	Ø	Neither first nor last packet
Ø	1	First packet but not last
1	Ø	Last packet but not first
1	1	First and last packet

SB Sequence bit field

MDF Message definition field:

1 Process message
2 Control message
8 Sequence message

Figure 11: Message with Control Word

put into a message by the nucleus when it is copied from the sender's address space into a message buffer and removed when it is copied from a message buffer into the receiver's address space. Thus, like packeting, the sending and receiving processes are not aware of the existence of the message control word.

Control message functions provided by each nucleus are listed in figure 12. Most of these functions are performed by the nucleus process because messages must be sent to carry them out.

The packet size in the current system is chosen so that a disk file record (as defined in the file system) can be transmitted without packeting; a record is approximately 130 characters.

The control token lost wait, the length of time a nucleus waits after requesting that a message be sent until it presumes the ring control token has been lost, is different for each machine and ranges from 1/2 second to 1 minute.

Status check
: Does process exist? Nucleus ignores message; sender notified that process exists by response status bits.

Suspend process
: Suspend process execution, do not release resources. Notify and connect terminal to process named in NOTIF field of the suspended processes' context block.

Terminate process
: Terminate process execution, release resources.

Start process
: Start process execution at start address specified in load module.

Restart process
: Start process execution at address where process suspended.

Interrupt process
: Suspend process execution (initiated by user at terminal). Perform same actions specified in suspend process function.

Read directory
: Read file directory name which process is logged on to.

Examine memory
: Read specified memory locations in process.

Deposit memory
: Write specified memory locations in process.

Figure 12: Control Message Functions

The message transmission thresholds are:

	# attempts to send
transmit overrun or cyclic redundancy check	5Ø
match-accept	1
match-no-accept	15
no-match-no-accept	1

After a single match-accept or no-match-no-accept, the response is returned to the sender. The no-match-no-accept threshold is one because a considerable number of messages are sent to non-existent processes, which, if the threshold were higher, results in a noticeable reduction in transmission rate.

Currently, each process is allocated eight sending and eight receiving logical communication paths. The algorithm for selecting a path to close is least number of messages sent or received. This algorithm has proven unacceptable. Certain critical processes, in particular input/output handlers, tend to get their sequence bit tables filled with four or five processes with large numbers of messages sent (typically processes reading or writing large files). With the current algorithm it becomes nearly impossible to close these communication paths even though the processes may no longer exist. Once this happens, the overhead from sending sequence

messages gets larger as processes contend for scarcer resources, that is, the remaining sequence bit table entries. A least recently used algorithm (LRU) is being implemented to resolve this problem.

In order to minimize the time a ring interface is disabled while processing a message received signal, a linked ring of message buffers is maintained. After a message is received, the interrupt routine marks the buffer full and reinitializes the ring interface input with the next buffer. The input message routine periodically tests whether a buffer is full. If so, a system message buffer is allocated, the message is copied from the ring interface buffer to the system buffer, and the system buffer is placed on the input message queue. Thus, messages are copied four times if the message is transmitted around the ring, and only twice if it is not.

Future Directions

Plans for the near future include connecting the Varian 620/i's directly to the ring, constructing a new ring interface [FAR75b], continuing efforts to improve the reliability of the system, and assessing the performance and suitability of ring computer networks. The DCS prototype is being used as a tool in an undergraduate class on systems programming and as a resource for departmental research in resource allocation in distributed systems [EAR74], in the

design of specialized text handling processors [ARV75], and in the exploration of secure network communication protocols and internetwork security [FAR75c]. A link between DCS and the ARPANET is also being planned [EAR73, FAR73b].

Longer range plans include using the new ring interface to connect devices other than processors (e.g., terminals) directly to the ring and interfacing an existing operating system to DCOS.

EXPERIENCES WITH THE PROTOTYPE

Experiences with the DCS prototype, including available performance statistics, information distribution by broadcast messages, suggested changes to the ring interface, relationship between memory space and reliability, and the problem of how a process discovers a desired service's name are discussed in this section

Performance Statistics

There is no analysis of the DCS prototype available although work in this area is underway. Nevertheless, some subjective results were obtained by gathering data while the system was executing and developing a composite description of the samples. As such, the results should not be interpreted as precise descriptions of controlled experiments, rather they should be thought of as suggestive of the observed performance. In what follows, data on the distribution of message lengths and the frequencies of transmission errors are presented.

The message length probability density and cumulative distribution functions for a five minute period executing a three machine DCS with three interactive users are shown in figures 13 and 14. During the sampling period, arbitrary programs, such as QED and the assembler, were loaded and

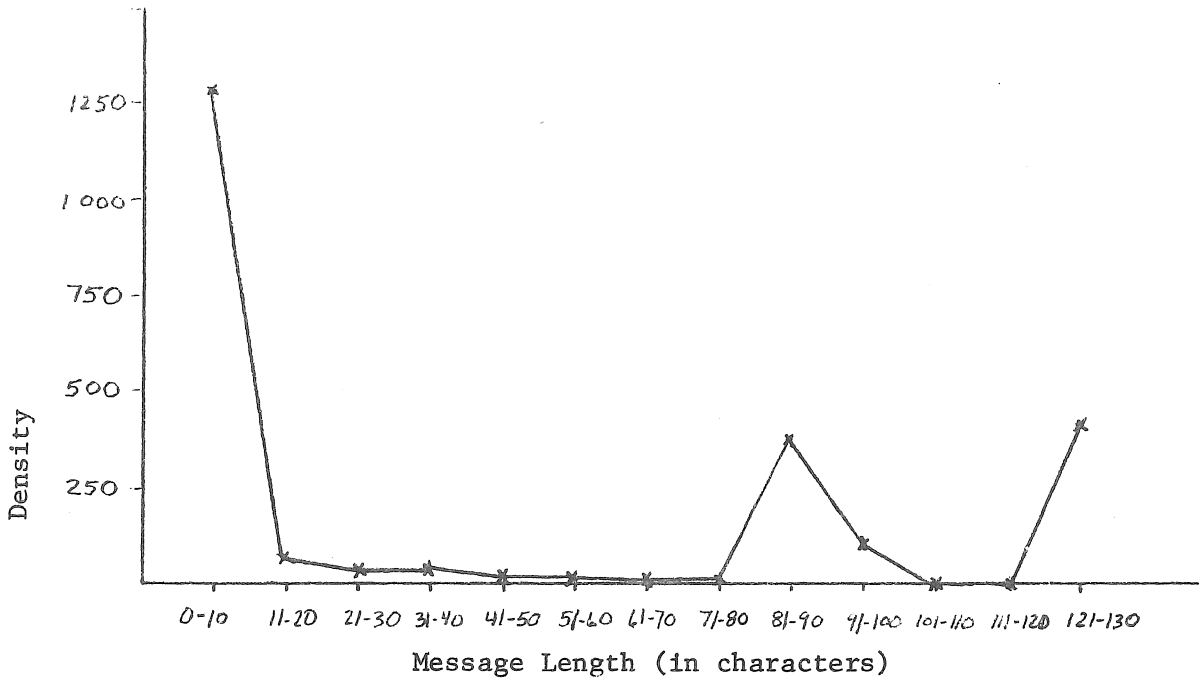


Figure 13: Message Length Probability Density Function

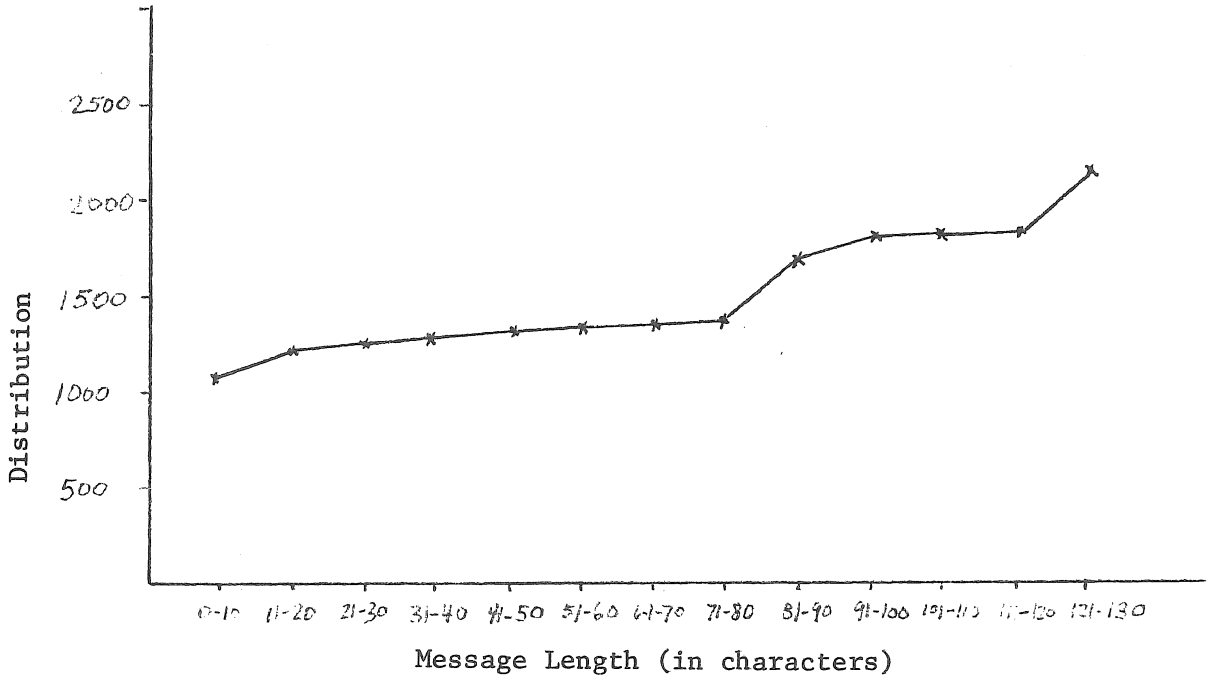


Figure 14: Message Length Probability Distribution Function

executed. There are three peaks in the density function. The first peak, messages with length between 1 and 10 characters, contains all input request and output response messages (4 and 2 characters long respectively) and other request and response messages. The peak between 81 and 90 character messages corresponds to sending binary records produced by the assembler. The last peak, messages between 121 and 130 characters in length, are reads and writes of the maximum length disk record (128 characters plus identification and request/response fields). Thus, frequently invoked mechanisms with constraints established in other parts of the system, such as the file system, significantly affect the observed distribution of message lengths.

During the 15 minute experiment, from which the 5 minute message length distribution sample was taken, the following error frequencies were observed:

frequency of retries on send	.1%
frequency of transmit overrun or CRC error	.006%
frequency of receive overrun on input	.1%
inability to transmit (token lost)	.01%
ring not initialized for input because buffer not available	.04%

The errors observed occurred in bursts, i.e. many retries

were required on a few messages as opposed to a single retry for many messages.

The frequency of transmission errors, at present, appears to be directly related to the transmission load. For example, as more processes are executed or messages sent more frequently by existing processes, the frequency of errors increases. This may result from errors not previously encountered in the hardware and software. Considerable effort is being made to understand and correct this problem.

Although we have not conducted any quantitative experiments on process and processor failure detection and recovery, the mechanisms for handling these errors are operational. The detection and recovery mechanisms in the prototype are not automatic (because status checkers have not been implemented); however, reloading the nucleus in the failed processor is straightforward. In most cases, processes executing on other processors not depending on processes executing in the failed processor have continued uninterrupted. This demonstrates the fail-soft reliability characteristic which was a predominant goal in this research project. Where this characteristic has not been achieved is at the process level. The problem is that an arbitrary executing process may depend on a resource provided by a specific process and, if the resource providing process fails

it is impossible for the system to unbind the using process from the failed process and, in those cases where meaningful, substitute another resource providing process. More work needs to be done in this area. In particular, investigating the possibilities of describing resource needs and usages in a way which would allow an algorithm, when notified of the usage needs of those processes affected by a failure, to substitute other processes (perhaps restarting failed processes) for the preempted resource. Some preliminary theoretical work on this problem has been completed recently by Merlin [MER74].

Information Distribution by Broadcast Messages

The DCS broadcast facility conveniently supports two ways of distributing information. The first is to distribute the information into disjoint subsets. The second is to distribute complete copies of the information. In the first case, a process accesses the information by broadcasting a request to all disjoint subsets, expecting an answer from the one who has the requested information. An example of this kind of distribution is resource allocation information. Each RA maintains a disjoint subset of the resources available in the system. In the second case, a process accesses the information by broadcasting to all copies of the information, using the first answer received. File, catalog, or directory information [FAR72c] is an example of this second kind of

distribution. In both cases, the problem of updating redundantly stored information is not made easier by the existence of the broadcast facility.

The broadcast facility is inadequate in cases where it is imperative that all processes which should get the message do get it. Notice that in the presently available facilities, if a process broadcasts a message and gets a match-accept response (some but not all got the message), resending the message may result in some destinations receiving a second copy and some a first copy. Furthermore, if the response to the retransmission is match-accept again, it is still indeterminant whether all processes which should get the message did. An example of this situation is when a nucleus broadcasts to all input/output handlers a command to free resources bound to a terminated process. The solution to this difficulty would be a broadcast facility which guaranteed that messages were received by all who should receive it.

There is another problem with match-accept responses to a broadcast message involving message sequencing. Suppose that a process broadcasts a message and the response status bits are match-accept. The processes which did not receive the message (because of the match no copy) may not receive the next message broadcast to them because their receive sequence bit is not synchronized with the sender's.

Enhancing the Ring Interface

Based on our experience with the prototype, several improvements to the ring interface are possible. The order in which the improvements are discussed is not related to their relative importance. Several of these improvements are being incorporated into a new ring interface [FAR75b].

The first improvement is that the names in the ring interface should be longer and there should be more of them. Another related improvement is to include hardware matching of broadcast names. The absence of this mechanism in the prototype has severely reduced the effective number of available names.

Including a machine identification number in the ring interface is a third possible improvement. This number could be used by the nucleus after it is loaded to initialize local variables pertaining to process name generation, interrupt device tables, and, where appropriate, input/output handler device reservation tables. The identification number should be manually resetable as is currently done with magnetic tape drives.

A local clock for detecting control token loss should be added to the ring interface. This would reduce nucleus overhead. It may also be desirable to put a token regeneration function in the hardware.

A fifth improvement is to include a mechanism in the ring interface wherein a message sent to it causes the processor to interrupt and initiate the ring load function. This would allow complete automation of the nucleus load function. Other external control functions should be investigated, such as sample processor status, power on processor, or power off processor, although these functions may be considerably more complex. Mechanisms similar to these and their effects on operating systems have been discussed with respect to the ARPANET by Metcalfe [MET72].

Some sort of transmission error check of the match/accept bits is needed. The current ring interface does not include the match/accept bits in the cyclic redundancy check (CRC). Other error detection schemes to supplement the CRC [HEI73] should be investigated.

Another possible improvement is to allow selective name shutoff at the ring. If one process is deluged with messages, currently the only way to shut off receipt of messages for that process is to disable receipt of all messages at the ring interface. This is an unacceptable alternative because other processes executing in a given processor should not be penalized by the behavior of the deluged process. Furthermore, selective shutoff is desirable because it signals to the sending process that the destination process is busy.

The next suggested improvement concerns ring level protection. If a network composed of processors with varying levels of hardware supported protection is to be protected from intrusions caused by requesting sensitive services in a more protected environment from a less protected environment, there must be a way of insuring the legitimacy of requests. One possible mechanism for accomplishing this is to have each ring interface include in each message a field describing the protection hardware support level of its attached processor. By using this, along with a table describing the lowest protection level from which a request for a given service can be made, ring level protection can be accomplished.

The last improvement is to include logical communication path management (sequence bit tables) in the hardware. Good design principles suggest similar functions should be grouped into modules and accesses to them rigidly controlled. In the current software, communication path management (a communication subsystem function) has percolated up to the process level. This has resulted in additional overhead and complexity in the nucleus to handle an infrequent problem (errors caused by disruptions on the transmission lines). Thus, communication path management should be incorporated into the hardware or an alternative solution to the transmission disruption problem found.

Relationship Between Memory Space and Reliability

One basic premise of this approach to providing reliable computing service is that the cost of processors and memory is decreasing. This was taken advantage of by connecting many processors together, resulting in a potentially more reliable system. One effect of this has been the use of multiple copies of information and software. At one stage of the development of the system, it was noticed that inordinate amounts of memory space were being used by the copies of various processes. Thus, it was decided to conserve memory space, which at the time had become scarce due to budget constraints, and to put more and more functions into fewer copies of software. For example, why distribute input/output devices around the network, requiring each processor to run an input/output handler? (If all devices are connected to a single processor, only one copy of an input/output handler is needed.) The reason is reliability. Clearly, if the processor to which all input/output devices are connected fails, most likely all processes executing in the system will eventually fail. This violates the goal of providing reliable, fail-soft computing service. From this one can infer that, up to some limit, distribution is directly related to reliability. Of course, this depends on how hardware and software are distributed in a particular configuration in relation to the

specific profile of demand. Notice though, that DCS provides a flexible environment for changing the software and hardware configuration.

Discovering a Desired Service's Name

This problem besets all computer networks. Suppose a programmer knows the particular service he desires (this is not very easy either), say for instance a regression analysis process. How does one find out the name of this process so a message can be sent to it requesting a regression analysis? In DCS, one sends a message to a resource allocator agent (a process which carries out the bid-request allocation algorithm) to initiate the process if a sharable copy is not presently executing. But how does one know the process name of the resource allocator agent? There are two solutions: predefine a name which all users are required to know or define a nucleus function "send to RA agent". In the first solution, the process requesting the service broadcasts to all agents who send a message giving their specific name, after which the initiate request can be sent. In the second solution, the nucleus is responsible for knowing the name of a particular agent and, if the agent known by the nucleus disappears, the nucleus finds the name of another agent.

CONCLUSIONS

Three types of conclusions can be drawn from our work on an operating system for DCS. First, many of the problems and possible solutions explored in the context of a minicomputer network are not restricted to that context. In particular, problems concerning process distribution, centralized control, and interprocess communication are found in networks composed of large processors and networks composed of both large and small processors, so solutions used in DCS might well be used in these other networks.

The second type of conclusion concerns specific details of the design which have proven useful in the prototype system. The utility of broadcast messages as a communication system mechanism for facilitating certain types of information distribution has been demonstrated. The feasibility of a system without centralized control has been demonstrated. The prototype has shown the necessity for different types of messages, i.e., process, control, and sequence. In particular, control messages are essential to being able to design a system that operates without commitment to the physical location of a process (location independence) and that has no centralized control. The suggested improvements to the ring interface should also prove valuable in a future implementation.

The last type of conclusion resulting from our work concerns possible directions for future research in computer-communication networks and in distributed computing. One area in computer-communication networks that needs investigation is alternative solutions to the problem of sequencing after transmission errors, solved in our system by the use of process level sequence bits.

An area for research in distributed computing is discovering and evaluating different forms of interprocess communication. A communication system could be designed with several forms of communication, ranging from one-process-to-one-process to many-processes-to-many-processes, where the less powerful forms cost less than the more powerful, according to some measure such as the product of transmission time and utilization of transmission capacity. Several forms are suggested by our work: one-process-to-one-process (DCS process-to-process messages), one-process-to-many-processes without guaranteed receipt by all destinations (DCS broadcast messages), one-process-to-many-processes with guaranteed receipt by all destinations, and content-based communication [GOR74]. Another area for research is investigating mechanisms for unbinding a service using process from a service providing process that has failed and substituting, where possible, an

equivalent service providing process.

ACKNOWLEDGMENTS

Many people have contributed to the DCS project. Foremost are Professors David J. Farber and Julian Feldman, principal investigators on the grants which supported the design and development of DCS, who have provided guidance to all phases of the project.

Particular contributions have been made by David J. Farber, who developed the initial system architecture [FAR72a, FAR72b]; by Frank R. Heinrich and Kenneth C. Larson, who contributed to the early design of the operating system and communications system; by Donald C. Loomis, who designed and implemented the ring interface; by Allan D. Foodym who contributed to the design and implementation of various parts of the system; and by William J. Earl and Paul V. Mockapetris, who are conducting the evaluation of the prototype system and are redesigning and reimplementing major parts of it.

Others who have made contributions include William E. Crosby, Steven K. Howell, Robert R. Ramos, Edward S. Schwartz, and Henry A. Sowizral, who implemented system and support software, and Gregory L. Hopwood and Marsha D. Hopwood, who designed and implemented the system programming languages used in the development of the software.

Finally, I wish to thank Dave Farber, Frank Heinrich, and

particularly Marsha Hopwood, whose comments on an earlier draft of this paper significantly improved its presentation.

REFERENCES

ARV75 Arvind. Personal communication.

BRI70 Brinch Hansen, P. The nucleus of a multiprogramming system. Comm. ACM 13 (April 1970), 238-241, 250.

DIJ68 Dijkstra, E. W. The structure of the T.H.E. multiprogramming system. Comm. ACM 11 (May 1968), 341-346.

EAR73 Earl, W. J. Interfacing the DCS to the ARPANET. DCS Project Memo, Department of Information and Computer Science, U. C. Irvine (March 1973).

EAR74 Earl, W. J. Resource allocation in a distributed computer network. Dissertation Proposal, Department of Information and Computer Science, U. C. Irvine (June 1974).

FAR72a Farber, D. J. and K. C. Larson. The structure of a distributed computer system -- the communication system. Proc. Symposium on Computer-Communications Networks and Teletraffic, Microwave Research Institute of Polytechnic Institute of Brooklyn (April 1972), 21-27.

FAR72b Farber, D. J. and K. C. Larson. The structure of a distributed computer system -- the software system. Proc. Symposium on Computer-Communications Networks and Teletraffic, Microwave Research Institute of Polytechnic Institute of Brooklyn (April 1972), 539-545.

FAR72c Farber, D. J. and F. R. Heinrich. The structure of a distributed computer system -- the distributed file system. Proc. International Conference on Computer Communications (October 1972), 364-370.

FAR73a Farber, D. J., et. al. The distributed computing system. Proc. Seventh Annual IEEE Computer Society International Conference (February 1973), 31-34.

FAR73b Farber, D. J. and J. J. Vittal. Extendibility considerations in the design of the distributed computer system (DCS). Proc. National

Telecommunications Conference (November 1973).

FAR75a Farber, D. J. A ring network. Datamation 21 (February 1975), 44-46.

FAR75b Farber, D. J. An unsolicited proposal for the development of a ring communications system for the intelligent terminal environment. Proposal Submitted to ARPA-IPT (June 1975).

FAR75c Farber, D. J. and K. C. Larson. Network security via dynamic process renaming. To appear in Proc. Fourth Data Communications Symposium (October 1975).

GOR74 Gord, E. P., M. D. Hopwood and L. A. Rowe. Language constructs for message handling in decentralized programs. Proc. 1974 ACM National Conference (November 1974), 526-530.

HEI73 Heinrich, F. R. Some error detection schemes to supplement the cyclic redundancy check in DCS ring transmission. DCS Project Memo, Department of Information and Computer Science, U. C. Irvine (March 1973).

HOP75 Hopwood, G. L. Notes on MOLSUE. DCS Project Memo, Department of Information and Computer Science, U. C. Irvine (January 1975).

HOP71 Hopwood, M. D. and G. L. Hopwood. MOL620 - a machine oriented language and language compiler for the Varian 620/I. Technical Report #1, Department of Information and Computer Science, U. C. Irvine (September 1971).

HOP73 Hopwood, M. D., D. C. Loomis and L. A. Rowe. Design of the distributed computing system. Technical Report #25, Department of Information and Computer Science, U. C. Irvine (June 1973).

LEV73 Levin, S. L. The distributed BASIC interpreter. Technical Report #33, Department of Information and Computer Science, U. C. Irvine (June 1973).

LOO73 Loomis, D. C. Ring communication protocols. Technical Report #26, Department of Information and Computer Science, U. C. Irvine (January 1973).

MER74 Merlin, P. M. A study of the recoverability of computing systems. Ph. D. Dissertation, Technical Report #58, Department of Information and Computer Science, U. C. Irvine (November 1974).

MET72 Metcalfe, R. M. Strategies for operating systems in computer networks. Proc. 1972 ACM National Conference (August 1972), 278-281.

ROW73 Rowe, L. A., M. D. Hopwood and D. J. Farber. Software methods for achieving fail-soft behavior in the distributed computing system. Record 1973 IEEE Symposium on Computer Software Reliability (April 1973), 7-11.

ROW74 Rowe, L. A., et. al. Distributed computer operating system programmer's guide. Technical Report #46, Department of Information and Computer Science, U. C. Irvine (April 1974).

SOW74 Sowizral, H. A. and D. J. Farber. A distributed program debugger. DCS Project Memo, Department of Information and Computer Science, U. C. Irvine (November 1974).