Experience with the Distributed Computer System (DCS)

Paul V. Mockapetris
and
David J. Farber

UCI Technical Report 116

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92716

February, 1978

This research was supported by ARPA, through the Information Processing Technology Office, under contract N00014-76-C-0954, and NSF under grant GJ1045.

Abstract

The Distributed Computer System (DCS) is a distributed timesharing system developed at the University of California, Irvine. The system consists of multiple minicomputers coupled by a ring communications system. The present system has been used as a vehicle for research and development of distributed computer systems, and to support a limited amount of class programming.

The design objectives of the system are coherence, high availability through failsoft behavior, and low cost. Enhanced system availability is achieved through the distribution of hardware, software, and control. The distribution discipline of DCS also makes changes in system configuration natural. The failure of a redundant hardware or software network component is isolated; the remaining components continue to function. Mechanisms for the detection and the restart of the failed components allow the construction of fault tolerant services. This paper describes the message functions and techniques that are used in the DCS and points out some possible evolutionary trends in message based distributed systems.

Motivation for the DCS

The DCS evolved out of a desire to explore construction of a coherent distributed environment. Three factors are relevant to the attractiveness of distributed computing:

1. The problems faced by today s computer systems are often distributed in nature. This can be due to geographical location, but is more subtly motivated by the distributed nature of our government businesses, and institutions.

2. The cost effectiveness of distributed systems is attractive. The multiple smaller systems of the distributed environment have profited relatively more by the development of mini and micro computers than have larger systems by maturing technology in their construction. The cost of communications bandwidth has fallen rapidly, but not as rapidly as that of memory and CPUs. The changing tradeoff has been noted in the design of packet networks such as TELENET [Roberts 77], and makes moving applications processing power towards the end user profitable.

3. By its very nature, distribution has some advantages over centralization. Multiple smaller systems do not

present the single point of failure seen in a centralized system. Hence, the reliability and availability of the total system can be superior. Incremental growth is easier. The overhead spent dividing up a larger system is eliminated. Because there are multiple processing sites, there can be variety in the processing systems and policies. The best system can be chosen for each application. One can question whether it is possible to provide a broad spectrum of services via a large central machine due to conflicting priorities and goals in the user community.

However, distribution is a technique, not a solution in itself. Several costs and problems are created by distributed processing systems.

Distribution of a large application does not simply divide the system design problem into smaller parts that can be separately solved. In addition, the user often wants the appearance of a centralized, integrated, system. In a distributed environment, data base problems related to synchronization, uniformity, and update must be solved to accomplish integration.

The centralized source still has certain economies of scale. For example, secondary storage systems exhibit large economies of scale.

While a multiplicity of machines, and machine types may be more effective, it is more difficult to support and maintain.

With these factors in mind, the original DCS design [Farber 70] aimed at producing a general purpose timesharing environment. The system that evolved has the following main features:

1. The system has 3 identical 16 bit minicomputers as its main processing elements. Other machines of different types are essential for full operation of the system, but are integrated into the system in a restricted manner. The DCS system does not completely address the problems of a heterogeneous processor population.

2. The hosts in the DCS system are connected by a unidirectional data ring running at 2.2 Mbit. The ring interface (RI) supports high level protocols such as addressing by process name and implicit acknowledgments at the hardware level.

3. The system has a simple process structure that provides a coherent environment. The system interface used by processes has been kept simple. A user

application consists of one or more user processes which communicate between themselves and access system resources via messages. Such an amalgamation is called a process net.

4. Interprocess communication is allowed only via the message system. Hence the physical location of processes is never a constraint.

5. The operating system kernel in each machine is small and simple. All machines on the system run identical kernels.

6. Most conventional operating system functions have been moved into autonomous server processes. These servers provide I/O, etc. via message protocols to the rest of the system.

The following sections describe the present DCS implementation in detail and attempt to impart the flavor of the system and the way it is used.

What is a process?

A DCS process is composed of one or more tasks, associated program storage, and one or more process names. All of the components of a process reside in one machine.

All of the present DCS hosts are multiprogrammed. A task is a locus of scheduling within a machine; a task can be blocked and run independently of other tasks. Each task has an associated context of general purpose registers, etc. Most processes have only one task; in the present system, only the nucleus process has more than one task.

The program storage is the memory space used for the code and data of the process' tasks. The program storage is composed of an impure (non-reentrant) block of code and data, and an optional pure (reentrant) code block. The pure code block may be shared between processes on the same machine. For example, if two users invoke the editor, the system will attempt to run both edit processes in the same machine. If this is possible, only one copy of the pure code is necessary. This strategy optimizes the use of our limited memory.

Shared impure segments are not allowed. The sharing of impure memory would invite the use of modes of interprocess communication other than messages. Any communication based on shared memory would either constrain the location of processes, or create the need for complex storage access mechanism.

When a process is created, it is assigned a unique process name. This process name can thus be used to

uniquely specify the destination or origin for a message. The process name is location independent; it does not imply the location of the process within the system.

In addition, the DCS software allows the creation of broadcast names. A broadcast name is associated with all processes of a given generic type. For example, all I/O Handlers have a common broadcast name. A message addressed to the I/O Handler broadcast name is delivered to all I/O handlers.

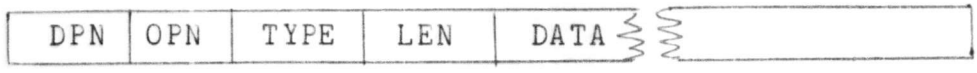
What is a message?

Messages are the standard mode of communication. When a process wishes to send a message, it supplies the name of a destination process or broadcast name and a message text. In the current implementation, process names are 16 bits in length. The names of certain system processes are forced (by software convention) to predefined values so that the user can easily communicate with them.

The system attempts to deliver the message and notifies the sending process as to whether the specified destination could be found, and whether errors were encountered in transmission. No routing data is required, and no initial connect protocols are visible to the user.

When a message arrives for a process, it is chained by the system on that process' input message queue. Processes consume messages from their input message queues via system calls. Several options are available. The process can wait a variable time for a message arrival, receive a message only if one is immediately available, test for an empty input message queue, etc.

The format of a message in the current implementation is shown in table 1.



where

DPN - destination process name (16 bits)
OPN - originating process name (16 bits)
TYPE - message type and flags (16 bits)
LEN - length of DATA segment (16 bits)
DATA - variable length data string

Table 1. - Message Format

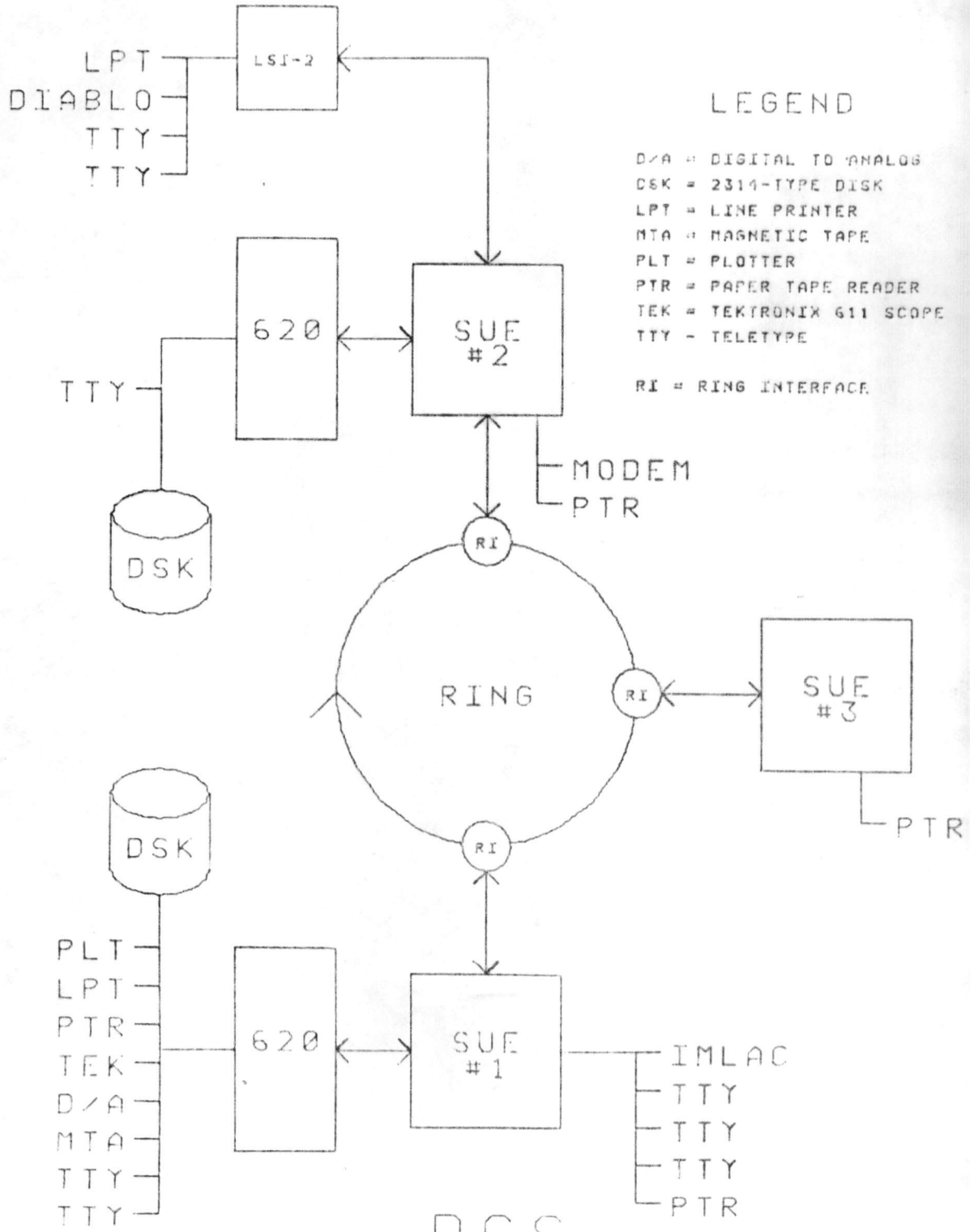
The transmitting process supplies the DPN, and the address and length of the data segment. The system fills in the OPN of the message with the unique name of the process that sent

the message. Thus a process that receives an unsolicited message can reply to the OPN in the unsolicited message. The type field is used by the system and contains sequence bits, etc. The hardware in the RI allows messages of up to 64K bits to be sent. Messages of this length would create a severe buffering problem and also lock out other machines from the ring for an extended period. Thus the system kernel packets messages whose data length is greater than 128 bytes. This packeting is transparent to the user. If a process attempts to transmit a message which would exhaust system buffer space, the message is not transmitted, and an error indication is returned to the transmitting process.

The Hardware System

The hardware configuration of the DCS is shown in figure 1. The main components of the network are 3 Lockheed SUE minicomputers connected via a communications ring. The resources of the Varian 620/i minicomputers are made available to the rest of the network by agent processes in the attached SUEs. The 620/i minicomputers currently function as intelligent channels.

Interprocess message transmission takes place over the communications ring. The ring interfaces (RIs) implement access to the communication ring. Ring data flow is unidirectional and the current implementation allows a



DCS
HARDWARE CONFIGURATION

FIGURE 1

transfer rate of over 2 MHz over a twisted pair circuit. The data on the ring can be represented as a bit string composed of concatenated messages and followed by the control token. The token is a bit pattern that denotes the end of the bit train, and that confers the right to transmit to each RI as it passes through. The train circulates around the ring continuously. RI status sensing and software timeout procedures provide for token initiation and replacement of tokens destroyed by transmission errors. Each ring interface is composed of independent input and output controllers. Figure 2 shows a functional model of the RI hardware.

The RI output controller can send a message around the ring by inserting it before the control token as the train passes. If there is no message waiting for output, the bit train is simply forwarded. If a message is output, a new token is transmitted after the message. Other RIs may also add messages in a similar fashion. Each message is deleted by the same RI that originally sent it as it completes one ring circuit. The RI can do this by simply deleting the lead message of any train if it added a message to the previous train.

The number of messages in the train varies according to the load placed on the communications medium and the length of the ring. There can be as many messages in the train as

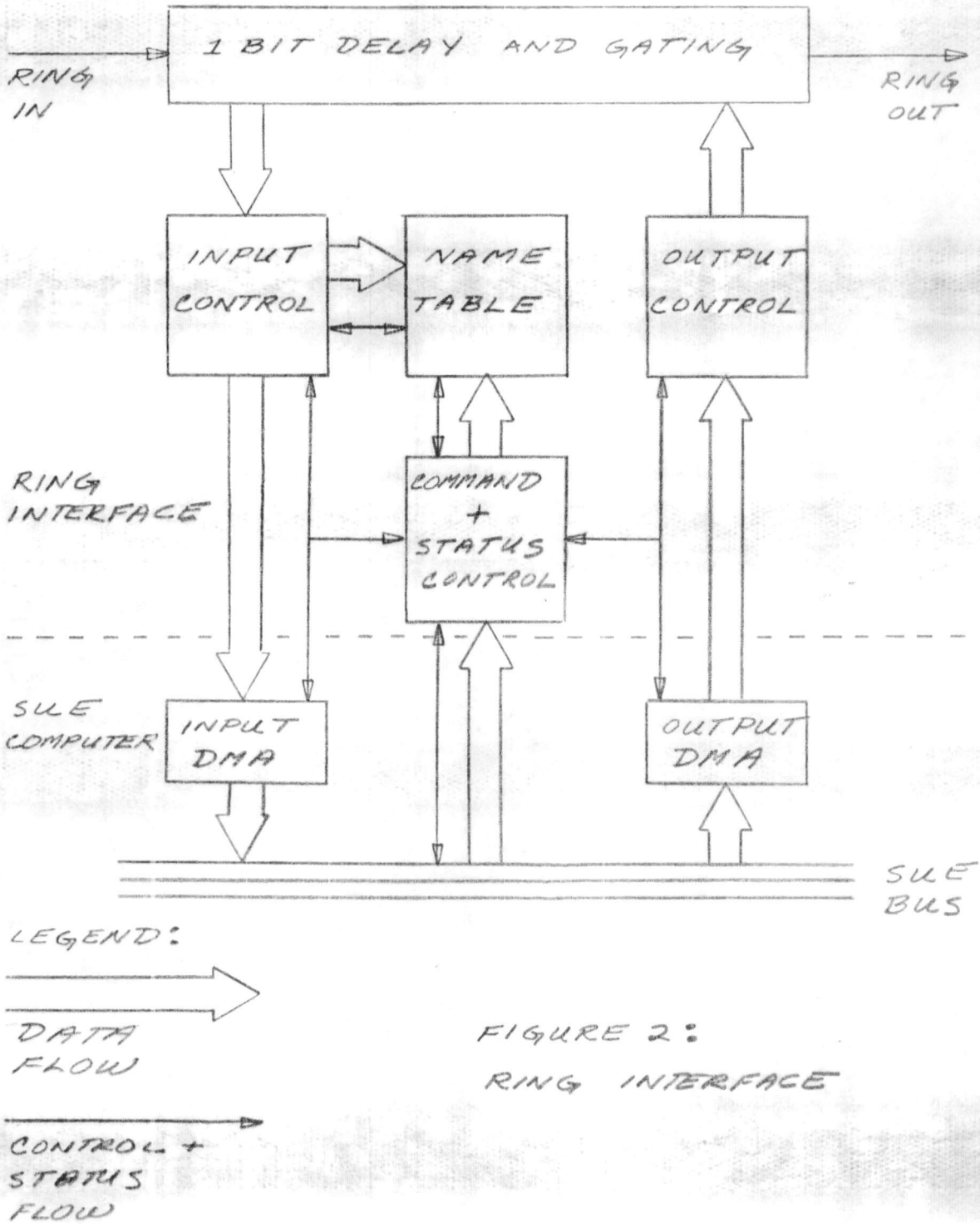


FIGURE 2:
RING INTERFACE

there are RIs on the ring, or no messages at all and a train consisting only of the token. Note that the physical length of the ring plus the one bit delay in each RI will usually be less than the length of a message. The RI may thus be deleting the head of a message train as it transmits the tail. In such a short system, the ring hold only a fraction of a message, and transmission rights circulate in a round robin manner.

The physical message transmitted by the RI includes two status bits, match and accept, at the end of the transmitted message. One of these status bits is set by the input controller of every RI that recognizes the DPN address. The match and accept bits are left unchanged otherwise. As a message passes through a RI on the ring, the match bit is set if the RI wanted to copy the message but could not. The accept bit is set if the RI copied the message correctly. The output controller that transmitted the message examines the returning message to detect transmission errors and to record the settings of the match and accept bits in the returning message.

The returned setting of the match and accept bits is used to detect certain transmission problems, and to drive the system sequencing protocols. The sequencing protocols used are similar to those used in other nets, except that sequence numbers are kept on a process name to process name

basis. The sequence bit process (SBP) is a special system process used to maintain the sequencing data, and to purge seldom used and defunct sequence data. The sequencing protocols are invisible to the user process and serve to increase the reliability of the message system.

There is some ambiguity possible due to the nature of the returned match and accept bits. This ambiguity is due to the fact that each of the bits signals that one or more RIs set that bits. The possible settings and meanings are shown in table 2.

MATCH	ACCEPT	MEANING
0	0	The message was transmitted to a process whose name was unknown to all RIs on the ring.
0	1	The message was addressed to a process whose name was known to at least one RI. All RIs who recognized the message address were able to successfully copy the message.
1	0	The message address was recognized by at least one RI. All RIs who recognized the message were unable to copy it.
1	1	The message was recognized by at least two RIs. At least one was able to copy the message, and at least one RI was unable to copy the message.

Table 2 - Match/Accept Results

at any machine, the message transfer system in the kernel routes it to the nucleus process rather than to the process to which it was addressed. This scheme has the advantage that the controlling process does not have to know the process name of the nucleus in the controlled process' machine. The nucleus validates the use of control messages and returns response messages as necessary. Control messages are used to enable the controlling process to start, kill, suspend, restart, or continue the controlled process. Other control messages allow examination of the controlled process' memory or storing of new values in the controlled process' memory. One user of the control message facilities is a remote control debugger which can precisely control the execution of a system of interacting processes under test. The process system may be spread over several machines and may even use several different types of machines.

The second type of system process is the I/O handler (IOH). Each machine with I/O devices may have an IOH to manage these devices and make them available via the message system. Note that while the kernel performs first level interrupt service and allocates the devices, neither the kernel nor the nucleus process contain any data management functions. When the nucleus wants to read a file, it does so by sending a message to an IOH.

Each IOH processes requests for data services for devices it controls. For example, an IOH that manages disk drives will contain the code to implement file system operations. Users request data management services in messages whose text adheres to the conventions set up for all IOH messages. Once a logical channel to a device or file is opened, IOH assigns a logical file number (LFN). Each LFN is unique within a given IOH. Thus a logical data channel is completely specified by an IOH/LFN pair. Open logical channels belong to the process that opens the channel. However, the owner of the channel can reassign ownership to any other process.

The command processor is a system process which provides the user a command level interface to the DCS. A terminal user talks to CP from his terminal to login to the system and to run programs. A program request causes CP to obtain a new process with the desired program code. When the program is run, the terminal (i.e. the IOH/LFN pair) may be transferred to the process. The user then can type directly to the process without any need to interface to CP. When the process terminates, the terminal IOH/LFN pair is transferred back to CP. CP then reads the next command.

Sample System Configuration

Figure 3 is an actual DCS system status display, generated by SYSTAT, a DCS command. The illustrated system consists of three machines. Each group in the display represents the contents of one machine; each line in the display is data for one process. The three machines have logical machine numbers of 1, 2, and D, as shown in the M column. The H.N. column correspond to the hardware name of the process. The DIR column specifies the home directory of the user running the process. The NAME field gives the process' symbolic name. The TTY column gives the symbolic identification of the controlling terminal for a process. The ORG and SIZE data give the process' origin address and size in bytes. The STATUS column gives the dispatching status of the process. HEXCLOCK and RUNTIME give the process' cumulative execution time in timer units and hours, minutes and seconds, respectively.

Each of the three machines has a nucleus process (symbolic name NUP/ H.N. of 1X01), and a sequence bit process (SBP/1X04). Machines 1 and 2 have attached I/O devices, hence there are I/O handler processes (IOH1/1102 and IOH2/1202) in these machines. The command processor (CP/1103) resides in machine 1. All of these processes are system processes.

The error message processor (EMP/2105) is a utility process that generates error messages and help sequences from encoded files. In this example, it was loaded by the QED/C219 process. Utility processes are created as needed, and self destruct if they are not used for a period of time. During the lifetime of a utility process, it may be used by any number of processes. Other examples of utility processes are the login process, the file finder process, and the process finder process.

Processes C10C and C219 are user class processes. C10C is the SYSTAT process itself; figure 3 is a duplicate of the display it produced. C219 is a user running the QED text editor.

```
        D C O S   S T A T U S    WED 07-JAN-76 08:51:03

M  H.N. DIR   NAME   TTY     ORG  SIZE STATUS HEXCLOCK RUNTIME

1  1101       NUP            B0D0 4408 RN     00001614 00:01:34
   1102 DCS   IOH1           01E8 311E IX     000228C3 00:39:18
   1103 DCS   CP             33BA 3F40 IT     000387CA 01:04:16
   1104       SBP            AA00 06D0 *      00000451 00:00:18
   2105 DCS   EMP            747E 0AC8 IT     00000146 00:00:05
   C10C       SYSTAT TTY100  7FFA 1730 RN     00000051 00:00:01

2  1201       NUP            B0D0 4408 RN     00001DE5 00:02:07
   1202 DCS   IOH2           01E8 32BE IX     00025088 00:42:08
   1204       SBP            AA00 06D0 *      00000229 00:00:09
   C219 DDT   QED    TTY201  47A4 2FF8 IT     000002E1 00:00:12

D  1D01       NUP            B0D0 4408 RN     00008989 00:09:46
   1D04       SBP            AA00 06D0 *      000003BB 00:00:15
```

Figure 3 - SYSTAT display

Sample Message Dialog

Table 3 shows a stylized message dialog corresponding to the initial connection of the user to the DCS and execution of a program which lists a file on the the user's console. Please note that the messages are numbered only for purposes of this discussion. The dialog is edited; message sequences and protocols for login and location of resources are omitted.

	FROM	TO	MESSAGE
1	CP	All IOH	Please assign me any free terminals.
2	IOH1	CP	I have assigned you terminal "TTY7:", refer to it as LFN 5.
3	CP	IOH1	Write prompt and read from LFN 5.
			<The user types "RUN LIST,30 ALPHA.TXT" on terminal TTY7:.>
4	IOH1	CP	I read the string "RUN LIST,30 ALPHA.TXT" on LFN 5.
5	CP	IOH2	Reserve the file "LIST.RB" for my use.
6	IOH2	CP	I have reserved the file "LIST.RB" for your use and have also assigned LFN 2 for your use.
7	CP	IOH2	Open file "LIST.RB" for reading on LFN 2.
8	IOH2	CP	O.K.
9	CP	IOH2	Transfer LFN 2 to process NUC1.
10	IOH2	CP	O.K.
11	CP	NUC1	Create a process from the file assigned to LFN 2 on IOH2. Its execution directory is "PVM", and it is a user process requiring 30K.
12	NUC1	IOH2	Read a record from LFN 2.
13	IOH2	NUC1	The record from LFN 2 is: <text of record>.
			<The previous two steps are repeated as many times as necessary.>
14	NUC1	IOH2	Free LFN 2 and the file assigned to it.
15	IOH2	NUC1	O.K.
16	NUC1	CP	Load request complete; process name is "UP".
17	CP	IOH1	Transfer LFN 5 (TTY7:) to process UP.
18	IOH1	CP	O.K.
19	CP	UP	Your controlling terminal is LFN 5 on IOH1, your option string is "ALPHA.TXT".
20	CP	CTRL(UP)	Start executing with controlling terminal LFN 5 on IOH1.

Table 3 - Message Dialog

The sequence starts with message 1 sent by the Command Processor (CP) to all I/O Handlers (IOHs). This broadcast message requests the allocation of any free terminals to CP. Such a broadcast is periodically done by CP to collect terminals which were freed by processes and/or terminals connected to IOHs which only recently came into existence.

Message 2 is a response by IOH1 to the request broadcast in message 1. The text of the message states that symbolic terminal "TTY7:" has been assigned to CP as Logical File Number (LFN) 5. Note that LFNs are unique only within a given IOH; an open I/O logical channel is specified by the IOH/LFN pair.

The third message writes a prompt on the newly acquired terminal to inform the user that CP is ready for input; it also requests the text of the line read as the confirmation from IOH. When the user types the string "RUN LIST,30 ALPHA.TXT" he is entering input data which will be interpreted by CP as a command. The text of the terminal input is returned as the response to the combined write/read operation specified in message 3. CP interprets the text of message 4 to be a RUN command for program LIST, with a 30K core minimum and an option string of "ALPHA.TXT" for the LIST program. Message 5 is a request by CP to IOH2 to allocate a LFN for use in reading the load module of the program LIST (file LIST.RB). IOH2 responds that the file is

available and assigns LFN 2 for use by CP in using the file. Message 7 is the open request for the file on the preallocated LFN. Messages 6 and 8 are the IOH2 acknowledgments for the reserve and open requests respectively.

CP desires to load the process into the machine with a nucleus named NUC1. The load file and LFN are assigned to CP at this point. IOHs only honor data transfer requests for the process which reserved the LFN. To allow NUC1 to load the process, it transfers ownership of the LFN assigned to the file to NUC1. It requests the transfer in message 9. IOH2 acknowledges the transfer in message 10.

CP then requests NUC1 to create a process using a supplied IOH/LFN pair. The load request message also specifies that the core assigned to the process is to be at least 30K. Additional options that are specified in this message are the run time directory to be used with the process, privilege codes, etc.

If NUC1 is able to process the request, it allocates program core, and begins to load the process. Messages 12 and 13 portray the reading of one load record. These two steps are repeated as many times as necessary to load the file. When the load is complete, NUC1 signals IOH2 that it is done with the file and LFN so that they can be reallocated. IOH2 acknowledges the freeing in message 15.

NUC1 signals the completion of process creation in message 16. This message is a response to the load request in message 11 and carries the name of the newly created process (UP). The user process (UP) is in a non-executable state when created; in particular, it is blocked waiting for the arrival of a start control message. UP can accumulate messages on its input message queue, but cannot execute.

When CP gets message 16 it transfers the user's terminal to UP. The transfer is accomplished in message 17. IOH1 acknowledges the transfer in message 18.

Two remaining things must be done to start the execution of UP. It must be informed of its option string and must be unblocked. Message 19 is a "start-up" message from CP to UP containing the option string "ALPHA.TXT", and a specification of the controlling terminal's IOH/LFN pair. This message is a data message and does not change the blocked status of UP. The start up message is a DCOS software convention that allows a terminal user to pass arguments to a process via the command line. This message does not affect the blocked status of UP.

Message 20 is a control message from CP to the environment of UP. It sets the controlling terminal for the process and causes the process to be unblocked if the process is waiting for a start message. Thus, UP is now

able to run. Note that the start up message (19) precedes the start control message (20) so that the process is guaranteed that its start up message, if any, is available when execution begins.

At this point UP will begin execution. When it issues its first receive message SVC, it gets message 19. By sending messages to the IOHs, it can locate file ALPHA.TXT and print it on the users terminal. When UP terminates, the terminal controlling UP (TTY7:) is transferred back to CP by the nucleus as it destroys UP.

Conclusions and Suggestions for Future Work

The DCS is neither the "coming structure" for all computer networks, nor even the "best" structure for local nets. No network could do this due to the wide range of goals and available technologies. The DCS does not address many problems that must be faced in current networks such as long haul communications over low bandwidth links, and software "inertia" from already existing systems. The DCS does provide an operational example that furnishes practical insights and suggests directions for future work. Two areas that are explored in the DCS system, and that we feel merit future work, are the communications system and the process environment.

The DCS system architecture requires a good communications system at its heart. We used a ring; other packet transport systems, such as the Ethernet [Metcalfe 77], could do the job. All of the available technologies have tradeoffs due to the "singularities" of transmission techniques and the level of intelligence designed into the lowest level communication system. Our bias is to provide a fairly high level of support in the communications hardware rather than attempting to minimize functionality in the device. This decision made our software simpler while increasing the hardware cost.

Our RI has two fairly unique properties: the implicit acknowledgment system, and process addressing of messages. Because our system uses only a few percent of the raw ring bandwidth, message transfer rates are determined by the software overhead. The implicit acknowledgment system is worthwhile not because of ring traffic, but because of the queueing, scheduling, and other software overhead it eliminates. The process addressing scheme also saves overhead, but we feel that its real worth is the way it supports our process structure.

We are presently engaged in the final checkout of the Local Network Interface (LNI) as a successor to the present RI. [Mockapetris 77] The LNI extends the addressing mechanism and uses a better transmission technique. The internal structure of the LNI is partitioned to separate the transmission system interface, computer interface, message address recognition system, and the send and receive message processing and formatting functions. Each of the sections is programmable. We plan to use this design to experiment with varying transmission techniques, addressing methods, and higher level protocols.

The present LNI is implemented in TTL logic, but in a structure analogous to a planned LSI implementation. The high level of the LNI, coupled with a low cost LSI implementation could offer opportunities that are presently

prohibited by the cost of the TTL LNI. It makes little sense to couple a $1000 microprocessor or terminal, or even a $5000 disk to a system with a $2000 interface. Even a small LSI production would drop the cost of the LNI to the $100-$200 range. We feel that dispersing the functionality of a DCS like system via the dumb host route is promising and hope that the LSI version will allow this.

The DCS process context is the result of a concerted effort to provide a coherent system without impairing the power of the system. The DCS shows that messages can be reasonably used to transfer all data and control, and that the process net structure is viable. Future advances in this type of homogeneous system will depend on improvements to the basic system structure and a better understanding of process interaction.

The DCS process control mechanism is adequate to control the execution of a process net. The system debugger is the most sophisticated example of the use of the process control mechanism. The debugger's main shortcoming is that it doesn't "understand" the semantics of message dialogs, hence it describes the individual process states rather than the state of the process net. Two reasons for this exist.

First, the programming languages and support routines that the user employs to construct processes do not create a

representation that reflects the control interactions caused by messages. The debugger may know that a process is waiting for a message, but it doesn't know if a reply to a previous request is outstanding, and if so, from what process the reply should come. A programming language or control flow modelling system that allows the user to represent a process' internal control flow by means of a token flow model or transition map may be the answer. [Pickens 76]

The second part of the difficulty is the proliferation of interprocess server protocols that seems to be endemic to all networks. This proliferation complicates the user's programming task and makes observations based on message content difficult. This problem may be tractable in a homogeneous environment such as the DCS.

A solution to these problems would be applicable beyond the debugging stage. [Farber 76] To construct a failsoft environment, failures must be detected. Observation of message flow in a process net can detect many failures. Security violations can also be detected. Our present approaches all rely on incorporating timeouts and other checks in each process. This causes duplication of effort and makes security verification of any sort difficult.

We believe that future generations of DCS like systems will solve these problems by a combination of more sophisticated communications services, better process environments, and process net design tools. The present communications hardware can be extended to deal with higher level problems such as sequencing, etc. At the same time, it makes sense to look at new facilities in addition to dispersing the present message software into the communications hardware. The message address names can be used to reflect structural information about the process net, and to partition systems for security purposes. Because message arrivals determine the vast majority of process scheduling decisions, it may be desirable to incorporate process scheduling into the communications hardware. The use of broadcast messages needs exploration, and better methods to guarantee delivery can be implemented. Interprocess synchronization and resource allocation facilities can be assisted by the communications system. But before these communications tools can be effective, the basic process structure must be enhanced; we don't need new server processes and protocols, structures and facilities until we develop a basic environment that is coherent and can be dealt with by the user.

Acknowledgements

Many people have contributed to the DCS project. This paper is a summary of work done by these people. Professors David J. Farber and Julian Feldman served as principal investigators and provided valuable guidance. Students who have made contributions include William Crosby, William Earl, Allan Foodym, Elaine Gord, Frank Heinrich, Steve Howell, Greg and Marsha Hopwood, Kenneth Larson, Donald Loomis, Eric Olsen, Robert Ramos, Lawrence Rowe, Edward Schwartz, Henry Sowizral, and others.

Bibliography and References

[Farber 70] David J. Farber, "A Distributed Computer System", UCI TR 4, Sept 1970

[Farber 76] David J. Farber and John Pickens, "The Overseer", Proceedings of the International Computer Communications Conference 1976

[Metcalfe 76] Robert M. Metcalfe and David J. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", Communications of the ACM vol 19, no 7, July 1976

[Mockapetris 77] Paul V. Mockapetris, Michael R. Lyle, and David J. Farber, "On the Design of Local Network Interfaces", Information Processing 77, Proceedings of the IFIP 77 Congress

[Pickens 76] John Pickens, "Debugging and Monitoring Distributed Control Structures", PhD thesis, University of California, Santa Barbara, 1976

[Roberts 77] Lawrence G. Roberts, "Packet Network Design - The Third Generation", Information Processing 77, Proceedings of the IFIP 77 Congress