

UC Irvine

ICS Technical Reports

Title

Software Methods for Achieving Fail-soft Behavior in the Distributed Computing System

Permalink

<https://escholarship.org/uc/item/30m5j1xw>

Authors

Rowe, Lawrence A.
Hopwood, Marsha D.
Farber, David J.

Publication Date

1973

Peer reviewed

SOFTWARE METHODS FOR ACHIEVING
FAIL-SOFT BEHAVIOR IN THE
DISTRIBUTED COMPUTING SYSTEM

Lawrence A. Rowe, Marsha D. Hopwood,
David J. Farber

TECHNICAL REPORT #24 - 1973

Lawrence A. Rowe, Marsha D. Hopwood, David J. Farber
 Department of Information and Computer Science
 University of California, Irvine
 Irvine, Calif. 92664

Summary

The Distributed Computing System* (DCS) is a network architecture being developed at the University of California, Irvine. The main goal of the DCS is to develop an integrated hardware and software system which provides reliable, fail-soft service at low cost. This reliability is to be achieved by distributing hardware, software, and control of the system over the network. The system may be located at a single site or geographically distributed. Connections between major hardware components have been designed so that a failure in one component does not necessarily make other components unavailable. The software system complements this hardware design. A software nucleus is resident in each processor in the network. Other parts of the software are modular and several processors are capable of hosting the various modules. Because software control is distributed rather than centralized, a failure in a software module does not affect all the processes that are in execution in the network. The only processes affected are those using the particular software module which failed. This system design allows partial failure while being relatively immune to total failure.

The DCS design recognizes that one can never be certain that software is completely debugged. Thus, the system has facilities for detecting and recovering from software failures. A failure may be limited to a process or it may extend to all processes within a processor. When a failure in a single process is detected and that process is a system process (e.g. a source allocation process or an agent for process initiation), the rest of the system continues to function. Other software modules take on the additional work load or a new copy of the failed process is initiated. Non-system processes are terminated with appropriate notification. When a failure involving an entire processor is detected, it is assumed that the resident nucleus has failed and a new copy is loaded into this processor by another processor in the network.

This paper describes techniques which will be used in the DCS software to obtain reliable, fail-soft behavior, with emphasis on the detection of and recovery from software error.

Introduction

The Distributed Computing System [6] is a computer network with an integrated hardware and software design. The main design goal is to provide reliable, fail-soft service at low cost. The software system will achieve this design goal by distributing software control over the network and by including failure detection and recovery facilities throughout the system.

The DCS makes use of standard hardware components. Because very little specialized hardware is used, many of the redundancy techniques described in

the fault-tolerant computing literature [2,3,9] cannot be employed. In particular, masking redundancy techniques are not generally applicable. Standby redundancy techniques are used only to the extent that the detection of a failure can cause a system reorganization, yielding, at least temporarily, a different and somewhat degraded system. There are no standby spares. Modularity is employed, both in the hardware and software of the DCS.

The DCS Environment

The hardware system is a collection of computing system components connected to a digital communication ring by ring interfaces. In the prototype system mini-computers are used as the computing components. The communication ring serves as a unidirectional information path and the ring interfaces assist in information routing. Figure 1 shows a DCS configuration with six processors.

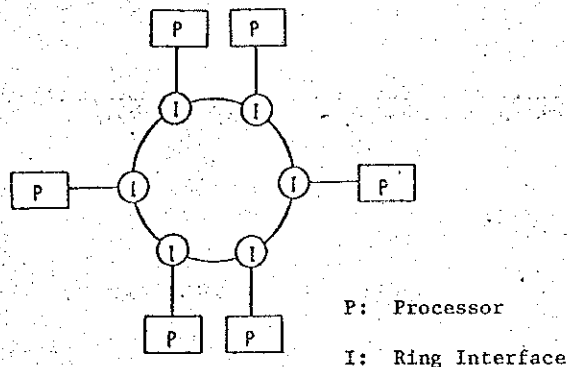


Figure 1: A DCS with Six Processors.

The DCS software system is process oriented, that is, all activities are carried out by processes. These processes, or tasks, interact by sending and receiving messages. A process places a message addressed to another process onto the ring. As the message arrives at each ring interface, the interface checks its table of active processes to determine if the destination process is present in the attached processor. If the process is present, the interface attempts to copy the message into the processor memory. Whether the process is present or not, the interface allows the message to travel on to the next interface on the ring. The message continues around the ring until it arrives at the interface for the processor in which the sending process resides. This interface removes the message from the ring.

Our approach to communications protocols is different from that used in most other computer networks [5,8,11]. In the DCS, messages are addressed to processes, rather than processors, with routing accomplished through a combination of hardware and software facilities. This approach has the advantage that each process is unaware of the network topology or the physical location of any other process.

This work is supported by the National Science Foundation under grant GJ-1045.

The system software consists of a nucleus for each processor and a collection of processes. A nucleus is resident in each processor at all times. This nucleus schedules activities within the processor and provides message communication services. The other system processes provide services such as process management, resource allocation, and error detection and recovery. System services, except those provided by the nucleus, are requested by sending messages. Since messages are routed to the addressed process no matter where in the DCS the process resides, software services can be distributed throughout the network. In fact, the only system software required in a particular machine is the nucleus.

File systems [7] and language processors also communicate by sending messages. The component processes of these can likewise be distributed throughout the network.

Protection

Protection in the DCS is achieved by insuring that the sending process name in a message is actually the name of the process sending the message. (Brinch Hansen [1] and Denning [4] describe the advantages and disadvantages of this forgery prevention technique.) In the DCS the originating process name, or sender, is placed in the message by the resident nucleus. This means that the integrity of the system is only as good as the least protected nucleus. Traditionally, a system is protected using a combination of memory protection and privileged instruction access. The difficulty of effectively utilizing these protection tools combined with possible vulnerability of the entire network make it desirable to have additional protection at the network level. The cost of traditional protection hardware as well as the desirability of being able to utilize processors for which no protection hardware is available also make network level protection desirable. Thus we have chosen to pursue protection in terms of software without precluding the introduction of hardware protection both at the processor and network level at some later date. One simple software protection scheme is to provide only interpretive systems and limit usage to these systems only. A more general solution, which we are investigating, is to group machines into protection equivalence classes and to control the sending of messages across equivalence classes.

Load Sharing

One advantage of a network architecture is the possibility of load sharing, dynamically distributing the load so that each processor receives some portion of the total work. Dynamic load sharing can have many beneficial consequences. In particular, when a processor becomes unavailable, its work load may be divided among the other processors. However, most current networks do not attempt dynamic load sharing for two reasons:

1. Transferring program code from one processor to another, dissimilar processor is usually impossible.
2. Transferring a process from one processor to another processor, whether or not the processors have the same order structure, requires that communication paths to the process be adjusted.

The DCS design, in which messages are addressed to a process rather than a physical location, eliminates the second difficulty, making load sharing among similar processors possible. Name addressing also makes it easier to replace a process running on one processor by a process of the same type running on another, dissimilar processor. This can be used to effect load sharing among dissimilar processors without solving the program transferability problem.

Distribution of Control

System control functions, such as scheduling and resource allocation, are distributed over the processors in the network. The nucleus in each processor is responsible for scheduling the processes active in that processor. In this way a failure in any given processor will affect only a subset of all processes in execution.

Resources in the DCS are treated as processes. Each resource is associated with a process and all use of that resource is through the associated process. Resource allocation can then be achieved through the process management mechanism. We call this mechanism a request for quotation scheme. To initiate a process, a user sends a request for quotation message addressed to all resource allocators. Each processor in the network has a resource allocator which allocates resources (e.g. memory and peripheral devices) connected with that processor. Note that the resource allocator need not reside on the processor for which it does allocation. The resource allocators reply with bids to perform the requested service. The requestor receives these bids and selects the best according to its own criterion. A message accepting the bid is then returned to the offering resource allocator, but this does not yet complete the agreement. Bids are not binding and the resource allocator may have accepted other work between the time the allocator sent the bid and the requestor sent the acceptance. Thus the allocator may not be able to satisfy the request, in which case the requestor is notified and he must renegotiate a contract. If, on the other hand, the resource allocator can still honor the bid, he registers relevant management information (record of use, time of use, cost of use, etc.) and returns a message to the requestor acknowledging the contract. It is only at this point that the agreement is complete and the requestor can assume that the resource has been assigned to him.

There are three points to notice about this scheme: 1. control is distributed, 2. load balancing is carried out implicitly by the relationships between the different bids, and 3. expansion or contraction of available resources requires little or no programming effort.

Failure Detection and Recovery

Software failures in the DCS are divided into two classes, nucleus failure and process failure. The failure of a nucleus is tantamount to a processor failure, since without the nucleus a processor cannot function in the DCS. The failure of a process, whether a system or user process, is somewhat less serious than a nucleus failure, since its scope is likely to be smaller. In this section we describe how software failures are detected, what conditions might have caused the failure, and how the DCS achieves a fail-soft environment in light of the continuing possibility of such failures.

Nucleus Failure

The failure of a nucleus running on a processor in the DCS is detected when communication involving a process executing on that processor becomes faulty. These communication failures are recognized when a process fails to accept a message.

The failure detection mechanisms come into play every time a process sends a message. The process must know whether the message it transmitted has been received. When the message is absorbed at the originating ring interface, there are two additional bits of information present, the message status bits, which indicate the status of the transmission attempt. The first of these bits, MSB1, is set when any ring interface matches the destination process name in the message to a process name in the attached processor, but the message cannot be copied into the processor memory. This can occur, for example, if all the buffers available for receiving messages are full. The second bit, MSB2, is set when a message is successfully copied into the processor memory. Thus, a successful transmission is signaled by MSB2 being set. Figure 2 describes the meanings of the possible values of the message status bits.

<u>MSB1</u>	<u>MSB2</u>	<u>Meaning</u>
0	0	The destination process name was not matched by any ring interface.
0	1	The message has been copied by at least one ring interface.
1	0	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 2: Possible Values of Message Status Bits.

The failure of a process to receive a message is recognized when a message is sent and the status returned is neither status bit set or MSB1 set. In the first case, neither status bit set, the process name is not known by any ring interface in the network. This means that a ring interface has failed or the process name has been removed from the ring interface. A test for ring interface hardware failure is periodically performed by other error detection mechanisms [10]. A process name is removed from the ring interface when the task has been terminated, so the sender should assume the process no longer exists.

The more difficult case is the second, where MSB1 is set. MSB1 is set when the ring interface has not been initialized to receive a message because the input message buffers are full or the software nucleus in the attached processor has failed. Whether the buffers are full or the software nucleus has failed is distinguished by error detection processes called status checkers which execute in some of the processors in the DCS. Each time a message is returned with MSB1 on, the nucleus, besides returning the status to the program, sends a message to all status checkers indicating the error along with the originating and destination process names. The status checkers maintain a scoreboard of communication failures.

The nucleus of each processor is aware of how frequently the ring interface is unable to copy messages because the input buffers are full. When this frequency reaches some threshold, the nucleus sends a copy of its process name table (the same table that is in the ring interface) to all status checkers. The status checkers then match the name table against the scoreboard of communication failures. If a name in the table matches a name in the scoreboard, then the status checker concludes that the message buffers are full. Upon recognition of this situation, the status checker may elect to initiate a load balancing process which might assist the busy processor by moving some of its processes to other machines.

The second possibility when MSB1 is set is that a processor has failed. The status checkers test for processor failure by periodically sending status check messages to all nuclei. If a nucleus fails to respond to several consecutive status check messages, the status checker assumes the nucleus has failed. A single status checker cannot certify a processor failure, so the status checker sends messages to the other status checkers and waits until a certain percentage agree that the processor has failed. If the failure is verified, the resource allocator for the processor is inhibited and a new copy of the nucleus is loaded into the machine.

Nucleus Recovery

Recovery begins with the loading of a new copy of the nucleus into the processor which failed. A nucleus restart process in some other processor will cause a bootstrap to be loaded and executed. The bootstrap clears the ring interface and initializes the hardware to receive messages from the nucleus restart process. The nucleus restart process sends messages to the bootstrap which are in turn placed in memory creating a new copy of the nucleus. This restart process does not change anything else in the processor. In particular, system tables which contain information about processes executing in the processor are not reinitialized.

As a result of the nucleus failure, there can be no certainty about the processes which were executing in the machine. It would be possible to continue these processes under control of the new nucleus. However, the difficulty of detecting process errors related to the nucleus failure and the possibility of compounding such errors during continued execution strongly suggest that the appropriate default action is to terminate these processes. Thus, the first task of the new nucleus is to inform the processes which initiated each of the terminated processes. This presents a problem since the information needed to do this is maintained in system tables within the failed processor and must be considered as possibly erroneous. Still, it is desirable to take advantage of correct information in the tables if possible. That then is our goal in restarting a nucleus. Correct information should be acted on without adversely affecting other processes in the ring should the information be wrong.

The nucleus iterates through the system tables sending a message to each process in the system which initiated a process previously in execution in the processor. These messages indicate that the initiated process has been terminated. Before sending this message, however, the nucleus looks to see whether the initiator process was also executing in the failed processor. If so, the notification is unnecessary and the nucleus moves to the next initiator-initiatee

process name pair in the table. (It should be remarked that while all these events are taking place the occurrence and time of each event are being recorded in an information repository. This provides an external record of the internal system actions.) If the initiating process was not executing in the failed processor, the nucleus sends a message to it indicating that a process it has initiated has been terminated. The nucleus then waits for some length of time for an acknowledgment from the initiator. When a process receives a message indicating that a process it initiated has been terminated, it takes the following actions:

1. If it did not initiate the process, a message is returned to the sender indicating that it did not initiate the process which was terminated. The process might also send status check messages to all of the processes it has initiated. This provides a check for an incorrect initiated process name in the system tables.
2. If it did initiate the process, a status check message is sent to that process. If the status check reveals that the process is executing correctly, the nucleus which sent the message is notified. If the status check shows the process to be terminated, an acknowledgment of the process failure is returned.

These procedures insure that correct information in system tables causes the nucleus to notify a process that a process it initiated has been terminated. If the information in the tables is incorrect we would use these procedures to guard against failures induced by erroneous information. There are three cases to consider.

The first case is an error in the initiator name. The incorrect name does not exist as a process, an error is detected and recorded. On the other hand, if the incorrect name does exist as a process, the nucleus can determine that it did not initiate the process which was terminated. This is guaranteed by the uniqueness of process names.

The second case is an error in the initiated process name. The initiating process determines if it initiated the process named in the message from the nucleus. If the erroneous name is not the same as one of a process it has initiated, the error is detected. If the name is the same, the initiating process sends a status check to the process, which responds that it is still functioning, and the error is detected.

The last case is an error in both the initiating process name and in the initiated process name. If the initiating process name does not exist, the error is detected. Suppose the initiating process name does exist. If the initiated process name does not exist, an error is detected. If the initiated process name exists, the initiator sends a status check message to the process and the check reveals that the process is still executing.

These procedures elicit correct actions when the information is correct and do not cause incorrect actions when the information is wrong. The possibility remains, however, that in the event of errors in system tables, an initiating process may not get notification of the termination of a process it has initiated. This places on each initiator responsibility for periodically sending a message to each process it has initiated. Acceptance of a message is an indication that the initiated process still exists.

The last action required by the nucleus, before being logically returned to the system, is to reset the resource allocator for the processor. The resource allocator was inhibited by the status checker when it recognized the nucleus failure. After this is done, the processor has been successfully reintroduced into the system.

Process Failure

Software failures in a single process, either a user process or a system process, are detected when an interrupt indicating an error occurs or when a regular, observable process action fails to occur. An interrupt indicating an error is usually the result of an attempt to reference a protected or undefined memory location or an attempt to execute a protected or undefined instruction. These are traditional hardware indicators of failure. The failure of a regular, observable process action to occur can be used as a software indicator of failure. Unlike hardware indicators, software indicators can be inhibited.

One software mechanism for detecting failure is a heartbeat. At regular intervals a process increments a word in its data space, the heartbeat location. The nucleus in the processor in which the process is executing checks this same location at intervals to see that the process has incremented the location since the last check and is thus still functioning in at least a minimal fashion. If the nucleus finds that the process has not changed the location over several heartbeat intervals, it assumes that the process has failed.

Process Recovery

After recognizing a process failure signaled by either hardware or software indicators, the nucleus establishes communication with a status checker and sends a message indicating that a process has failed. The status checker has five actions which it can take when notified of such a failure. The particular actions taken depend on what type of process has failed. The possible actions are:

1. Save a copy of the process environment;
2. Initiate a test process;
3. Initiate a new copy of the failed process;
4. Take no explicit action until told to do so by an external source; and
5. Terminate the process.

A copy of the process environment may prove useful in determining the cause of the error condition. A status checker can initiate a test process which monitors message transmissions to and from the failed process. By monitoring message traffic and initiating actions in the subject process by sending it messages, the test process may be able to determine the nature of the failure and decide whether the process should be terminated or allowed to continue. These test processes are not general procedures for examining any process; they are special tests written for a particular process which might fail. This mechanism will primarily be used as a means for testing and recovering from system process failures.

Another action which a status checker can take is to initiate a new copy of the failed process. This action may be taken either because the status checker was requested to do so by a test procedure or because

the type of process which has failed is essential to the functioning of some aspect of the DCS. When initiating a new copy of the process, the status checker may also cause message transmissions involving the process to be monitored. Initiation of a new copy will probably occur only with system processes although the mechanisms are available to the user.

A status checker may choose to take no explicit action when notified of a process failure. Upon receiving a message that a process has failed, the status checker sends a message to the information repository announcing the situation. The status checker will then take no further action involving the process unless it receives a message to do so.

The last possible action is to terminate the process. The default action taken for a process is to save the process environment, notify the process which initiated this failed process, and terminate the task.

We have described how a process failure is recognized and what actions a status checker may take when notified that a failure has occurred. Because we feel all possible sources of information should be used, another mechanism included in the DCS is to allow messages to be sent to the status checker by a user indicating that some failure has occurred. Although immediate action will probably not be initiated, the information provided by such messages is saved in a repository. This information may prove useful in repairing and maintaining the DCS.

Conclusion

In the preceding discussion we have shown how fail-soft software behavior can be attained in a computer network by distributing hardware, software, and control and by using relatively simple error detection and recovery mechanisms. Failures, such as those signaled by the failure of a process to accept a message, by the occurrence of an error interrupt, or by the absence of a software indicator (heartbeat), can lead to a wide range of recovery procedures. These include taking no action, saving the failure environment for future analysis, terminating the process, initiating a new copy of the process, and combinations of these actions. These software error detection and recovery features will help to make the DCS a reliable fail-soft system.

Acknowledgments

We wish to acknowledge the assistance of our colleagues on the DCS Project for the many hours of interaction which have contributed to the ideas presented here.

References

1. Brinch Hansen, P. "The Nucleus of a Multiprogramming System." Comm. ACM 13, (April 1970), 238-241, 250.
2. Carter, W. C. and W. G. Bouricius. "A Survey of Fault Tolerant Computer Architecture and its Evaluation." Computer 4, (Jan./Feb. 1971), 9-16.
3. Carter, W. C.; D. C. Jessup, A. B. Wadia, P. R. Schneider, and W. G. Bouricius. "Logic Design for Dynamic and Interactive Recovery." IEEE Trans. Comput. C-20, (November 1971), 1300-1305.
4. Denning, P. J. "Third Generation Computer Systems." Computing Surveys 3, (Dec. 1971), 175-216.
5. Farber, D. J. "A Survey of Computer Networks." Datamation 18, 4 (April 1972), 36-39.
6. Farber, D. J., J. Feldman, F. R. Heinrich, M. D. Hopwood, K. C. Larson, D. C. Loomis, and L. A. Rowe. "The Distributed Computing System." Proc. Seventh Annual IEEE Computer Society International Conference, (Feb. 1973), 31-34.
7. Farber, D. J. and F. R. Heinrich. "The Structure of a Distributed Computer System - The Distributed File System." Proc. International Conference on Computer Communications, (Oct. 1972), 364-370.
8. Farber, D. J. and K. Larson. "The Structure of a Distributed Computer System - The Communications System." Proc. Symposium on Computer-Communications Networks and Teletraffic, (Microwave Research Institute of Polytechnic Institute of Brooklyn, April 1972), 21-27.
9. Hopkins, A. L. "A Fault-Tolerant Information Processing Concept for Space Vehicles." 1971 International Symposium on Fault-Tolerant Computing, (March 1971), 97-100.
10. Loomis, D. C. "Ring Communication Protocols." Technical Report #26, Department of Information and Computer Science, University of California, Irvine, California, (Jan. 1973).
11. Roberts, L. G. and B. D. Wessler. "Computer Network Development to Achieve Resource Sharing." Proc. AFIPS 1970 SJCC, Vol. 36, AFIPS Press, Montvale, N. J., 543-549.