PUBLISHING:
A Reliable Broadcast Communication Mechanism

Michael L. Powell
David L. Presotto

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

ABSTRACT

Publishing is a model and mechanism for crash recovery in a distributed computing environment. Published communication works for systems connected via a broadcast medium by recording messages transmitted over the network. The recovery mechanism can be completely transparent to the failed process and all processes interacting with it. Although published communication is intended for a broadcast network such as a bus, a ring, or an Ethernet, it can be used in other environments.

A *recorder* reliably stores all messages that are transmitted, as well as checkpoint and recovery information. When it detects a failure, the recorder may restart affected processes from checkpoints. The recorder subsequently resends to each process all messages which were sent to it since the time its checkpoint was taken, while ignoring duplicate messages sent by it.

Message-based systems without shared memory can use published communications to recover groups of processes. Simulations show that at least 5 multi-user minicomputers can be supported on a standard Ethernet using a single recorder. The prototype version implemented in DEMOS/MP demonstrates that an error recovery can be transparent to user processes and can be centralized in the network.

This research was supported by National Science Foundation grant MCS-8010686, the State of California MICRO program, and the Defense Advance Research Projects Agency (DoD) Arpa Order No. 4031 monitored by the Naval Electronic System Command under Contract No. N00039-82-C-0235.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1983 ACM 0-89791-115-6/83/010/0100 $00.75

1. Motivation

To death and taxes we can add another certainty of life – errors. In a computing system, errors are caused by many things and often result in the failure of activities performed by the system. As a computer system becomes more distributed and contains more autonomous components, not only does the frequency of errors increase, but also the number of conditions that are classified as errors. One of the promises of distributed computing is a more available computing system. To achieve this goal, it is necessary to continue running despite the presence of errors.

Recovering from failures in a monolithic computer system has been thoroughly studied. A failure usually manifests itself as (or requires) the halting of the complete system. Therefore, a single, system-wide, consistent state is all that is needed to restart. Transaction mechanisms pioneered in database systems[Verhofstad 78, Gray 78], coupled with checkpointing of system and user program states, can allow the system to be restored to some state it had before the failure.

In a distributed system, complete failures are infrequent. Moreover, it is rarely preferable to force the whole distributed system to fail in order to recover from a partial failure. Thus, in recovering from errors, it is necessary to weave a restart state for part of the system into the current state of the rest of the system.

Because the system is distributed, it is more difficult to get a completely consistent picture of the system state. Since the system continues to run as the image is being formed, special care must be taken to ensure that the snapshot represents a complete and consistent state for the system.

The difficulty of recovery in distributed systems is due to the interactions between the processes. Unlike a single-processor system in which the interactions are strictly ordered, it may be difficult from any particular perspective to know in what order a set of interactions occurred. Published communications provides a way to recover from failures by using the broadcast medium as the viewpoint from which to obtain a properly ordered and consistent view of the system.

2. Published Communications

In this section, we define our model of processes and failures, and describe published communications in those terms.

2.1. Model of Processing and Failures

We define a *process* as an instance of a program that has begun to execute. The *state* of such a process includes:

- the instructions and variables used in the program
- information related to the sequencing of the program such as the program counter and the execution stack
- information managed by the system for the process such as messages not yet received or device buffers

Processes *interact* with one another by sharing or passing a subset of this state. Since the processors are assumed to be deterministic, the information contained in an instantaneous process state is determined by its initial state and its interactions with other processes.

Processes can *fail* for a number of reasons and in a number of ways. For the purposes of our study we can classify failures according to two characteristics: whether or not the failure is detected, and whether or not it is deterministic. *Undetected* failures are those that are not noticed by the process. For instance, if the adder produces a wrong answer and the program continues running with the bad result, there may be no way to know that there is a problem. A failure is also considered undetected if its effects are allowed to propagate to other processes before detection. *Deterministic* failures will occur whenever the process attempts the same operation or sequence. Without detailed knowledge of the circumstances, deterministic failures cannot be avoided. To be *recoverable*, a failure must be detected and must not be deterministic.

Included in the group of recoverable failures are hardware errors, transmission errors, resource and load dependent errors. Generally, a recoverable error would not have occurred if the processes involved had been running on different processors or at a different time. The essential characteristic of recoverable failures is that there is a (preferably good) chance they will not occur if the process does the same thing over again.

This paper treats only recoverable failures. A deterministic failure can be avoided only by eliminating some activity. The decision not to do something that had been requested is beyond the scope of a general recovery mechanism. A completely undetected failure cannot, of course, be recovered. However, since we include in undetected failures those that are detected too late to avoid propagation to other processes, it is possible to change some undetected failures to recoverable ones by increasing error checking in processes.

A *crash* is defined as the halting of a process on the detection of a recoverable failure. Since a crash is defined in terms of processes, the failure of a *processor* can be thought of as the crash of all processes in that processor. In fact, where convenient, the system is permitted to "round up" any system failure to a crash of all the processes affected by the failure.

Recovery is the act, following a crash, of returning the system to a consistent state from which it can proceed as if the crash had not occurred. Recovery requires two things: the ability to preserve information across a crash, and the ability to construct a consistent state using the information so preserved.

Information is preserved across a crash in a non-volatile storage facility, that is, one that has low probability of being altered by the crash. This is usually achieved by storing the information on devices whose failure modes are decoupled in some way from those of the other elements of the system. Often the information is also duplicated to insure against single failures of the storage facility. A number of solutions to this problem have been developed, including MIT's Swallow system [Svobodova 80, Arens 81] and Lampson's and Sturgis's stable storage[Lampson and Sturgis 79]. We assume that a reliable storage facility can be provided for use in publishing messages.

To allow the reconstruction of consistent states of processes, it is common to occasionally make copies of part or all of the process state. In this paper, we call the information necessary to reconstruct a complete process state at some point in time a *checkpoint*. The entire state of a process may be large, and techniques exist for recording only the parts of the process state necessary to reconstruct the complete state. To reduce the cost of making repeated copies as the process state changes, the system will make copies of the complete state only infrequently, and will usually make a copy of just that part of the state that has changed since the previous checkpoint.

Doing recovery in a multiprocess environment is more difficult for two reasons: the checkpoints must provide enough information to create a consistent state among several processes, and the recovered processes must be brought back to a consistent state with processes that did not fail.

2.2. Consistent States

For isolated processes, determining a consistent state is no problem – any complete state is consistent. However, sets of processes that interact must be checkpointed in such a way that all the separate checkpoints are consistent with one another in light of the interactions. Consider, for example, the three processes with the interactions shown in Figure 2.1 (adapted from [Randell 78]). The horizontal axis represents time (increasing left to right). The dashed vertical lines represent interactions between two processes in which both processes may communicate information to each other. The square brackets represent the checkpoints of individual processes. Since processes are deterministic except for their interactions, a set of checkpoints is consistent so long as there are no interactions which occur before some of the checkpoints and after other ones.

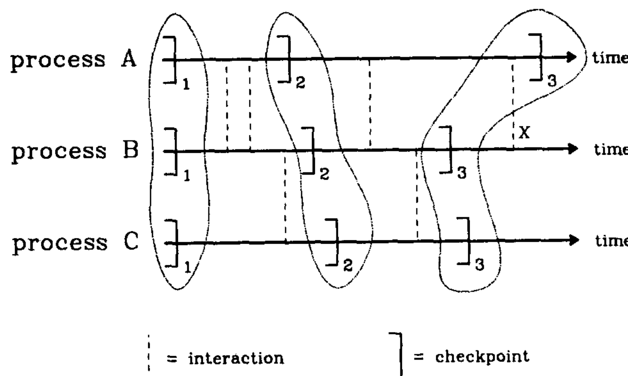

FIGURE 2.1: Checkpoint sets 1 & 2 are consistent. Checkpoint set 3 is not.

Represented graphically, if a line connecting a set of checkpoints intersects no interaction lines, then those checkpoints are consistent.

Figure 2.1 shows two sets of consistent checkpoints. The checkpoints labeled 1 represent the starting state of all three processes and are therefore consistent. The checkpoints labeled 2 are consistent since no interactions separate them. However, checkpoint set 3 represents an inconsistent view. If the processes are restarted from these checkpoints, process A will see the results of the interaction labeled X, but process B will not. Faced with these three checkpoints for each of the three processes, it would be necessary to go to checkpoints older than the most recent set.

The problem of obtaining consistent views of states has been addressed in distributed database systems. The most widely used solution has been that of transaction processing[Gray 78, Skeen and Stonebraker 81]. A transaction always takes the system from one consistent state to another. The interacting processes declare when a state is consistent, and the system prevents updates from having effect until another consistent state is reached. Transactions fit well in data base applications where secondary storage is considered to be the only important state. Applications are designed so that the state of a process between transactions is unimportant and need not be checkpointed. An application must also be prepared to redo work done for a transaction that does not complete.

We wish to place as little structure as possible on the processes that can be recovered. In recovering general distributed computation, we wish to have the following properties:

1) Programmers are not required to know about the checkpoint or recovery mechanism.
2) Checkpointing does not require global actions.
3) Recovery should require the minimum possible perturbation to non-failing parts of the system.

Property 1 means that the mechanism cannot require actions to be taken by the processes involved. Property 2 means that checkpointing will be done for individual processes. Property 3 means that individual processes must be recoverable, despite interactions with other processes.

One way to obtain these properties is to provide a way that an individual process may be checkpointed so that its state can be restored to that at the time of the failure. This may be done by saving the original state of the process, plus all of its interactions. The state may be recovered by restarting the execution of the program and providing it with the same interactions it had when it originally ran. If we constrain ourselves to message-based systems, then the interactions are messages and can be easily identified. The checkpoint information must be augmented on each interaction (message). We call the recording of these messages *publishing*. In Figure 2.1, process B could be restarted with checkpoint 3 and subsequently be presented with interaction X in order to recover it.

Since we are interested only in the most recent state and not any previous ones, we can often reduce the amount of information saved for a process by occasionally saving its complete state. Once the complete state has been saved, any older interactions can be discarded. We need only save those interactions that occurred after the most recent complete process state. We can state it as a rule:

> A checkpoint for a communicating process taken at time t_0 is valid at time $t > t_0$, if all the interactions of the process between time t_0 and time t are also saved.

2.3. Recovering from Crashes

To recreate the state of a process at time t from a checkpoint taken at time t_0, it is necessary to cause the process to redo the computation done between t_0 and t. Since processes are assumed to be deterministic between interactions, it is merely necessary to recreate the same interactions in the same sequence in order to cause the same computation to take place. It is of course necessary to prohibit a recovering process from affecting other processes until it reaches the state it had at the time of the failure. Otherwise, for example, an operation that should have been done once may be done twice.

In the above discussion, the reader might have assumed that time t was the time of the failure. Certainly, the above statements are true for that value of t. However, a more interesting t is the time that recovery for the process is completed. Since other processes continue while recovery is taking place, interactions between the time of the failure and the time recovery is complete must also be accounted for.

The recovery of a process thus contains several aspects:

1) The process is restarted from a checkpoint.
2) The process runs and is presented with all interactions that happened after the checkpoint.
3) Messages that were sent by the process before the failure occurred are discarded.

3. A Published Communications System

In this section, we describe the design of a practical published communications system. We have implemented published communications in DEMOS/MP, a message-based distributed operating system. DEMOS/MP is an experimental system, and does not actually support a broadcast medium as required by published communications. Thus, we have emulated an acceptable network.

Figure 3.1 shows a system in normal operation. A recording node is attached to the network via a special interface. The node is in charge of recording all messages on the network and of initiating and directing all recovery operations.

The necessary components of a published communication system are:

- Broadcasting messages
- Storing messages and checkpoints
- Detecting crashes
- Recovering processes

We will discuss our design of these components in turn.

3.1. Broadcasting Messages

In order to centralize the recovery function in a network, it must be possible for some node to see all communications that occur on the network, in the order in which they were received. On many local area networks (LANs), not only may any node overhear the messages destined for another node, but it may do it passively, that is, without the knowledge of the communicating parties. Such networks include Ethernet[Metcalfe and Boggs 76], rings[Farber et al 73, Wolf and Liu 78], and Datakit[Fraser 79].

These networks were not designed with publishing in mind. Therefore, they contain some characteristics that, though avoidable, were not considered harmful in the current implementations. For example, current Ethernet connections may miss messages because they cannot transfer data to the host computer fast enough. It would be necessary to build a fast enough connection with enough buffering to be guaranteed never to miss a message.

It is important that the communicating parties and the message recorder agree on which messages were correctly transmitted and which were not. Since errors may occur in the connection between the network and the receiver of a message, we must rely on a lower level (link level) communication protocol to correct for these errors. The recorder must understand the protocol so that it may determine whether a message was successfully transmitted or not.

Although the link level protocol can take care of messages the recorder accepted but the receiver did not, it cannot take care of messages the receiver accepted but the recorder did not. We would normally expect the recorder to be more reliable than a receiver, but nonetheless, it is possible for the latter case to arise. In this case, it is necessary for the recorder to interfere to cause the message to be rejected by the receiver (and retransmitted by the sender).

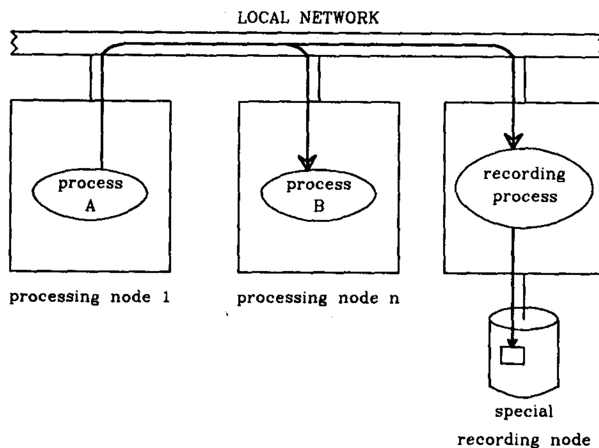

FIGURE 3.1: All messages are received by both the intended receiver and the publishing process which saves them on disk.

Possible solutions to this problem depend on the type of network. With an Ethernet, an "acknowledging Ethernet"[Tokoro and Tamaru 77] may be used, in which a space for an acknowledgement is inserted after each packet . This space would be for the recorder to acknowledge the recording of the message; if no acknowledgement is present, the receiver discards the packet. In a ring, it is possible to route the message so that it must pass the recorder before reaching the receiver (perhaps requiring an extra trip around the ring). The recorder could mark the message to be ignored if it could not record it. In a star network, all messages pass through a central point. A recorder attached to this point can refuse to pass on any messages it cannot record.

With a combination of slightly re-engineered media, a standard link level protocol, and a special feature to allow the recorder to destroy messages it cannot record, it is possible to guarantee that a message is received by the destination only if it is recorded, and that the recorder can determine which messages have been successfully received by the destination. This guarantee does not hold in the light of network partitioning or unrecoverable recorder failure. However, the probability of such failures can be made acceptably low with conventional hardware reliability techniques.

3.2. Storing Messages and Checkpoints

Checkpoint information is stored according to process id. When a new process is created, the recorder is told the initial state of the process (usually, a program name and some parameters).

Messages seen by the recorder are stored in the order in which they would be received by the destination process. This is the message stream that will be transmitted to the process if it is restarted. In addition, the recorder keeps track of the highest numbered message that a process has sent. This will determine when messages generated by a recovering process should be transmitted to their destinations.

At any time, the recorder will accept a checkpoint for a process. After the checkpoint has been reliably stored, older checkpoints and messages can be discarded. Frequent checkpointing decreases the amount of storage required and the time to recover a process, but increases the execution and network cost. The correct choice of checkpointing frequency will improve performance, but will not affect the recoverability of a process or the system.

3.3. Detecting Crashes

The crash detection system has two distinct functions; the detection of a process crash and the detection of a processor crash. The latter is rounded up to the crash of all processes on the processor.

Single process crashes are characterized by process errors. Such errors cause traps to the operating system kernel, which stops the process and then sends a message to the recovery manager containing the error type and process id of the crashed process.

Processor crashes are detected via a timeout protocol. For each processor in the system, the recovery manager starts a watchdog process on the recording node. The watchdog process watches for messages from the machine being watched. If no messages have been seen in a while, the processor is considered to have crashed and is restarted. Of course, it is a good idea for each processor to send a message from time to time, even if it has nothing to say, to avoid appearing to have crashed.

3.4. Recovering Processes

The system in recovery mode looks as in Figure 3.2. The main element is the recovery manager, which resides on the recovery node and is in charge of all recovery operations. It maintains a database of all known processes, their locations, and checkpoint information.

When the recovery manager receives notification of a crash it starts up a recovery process for each crashed process. The recovery process then performs the following steps:

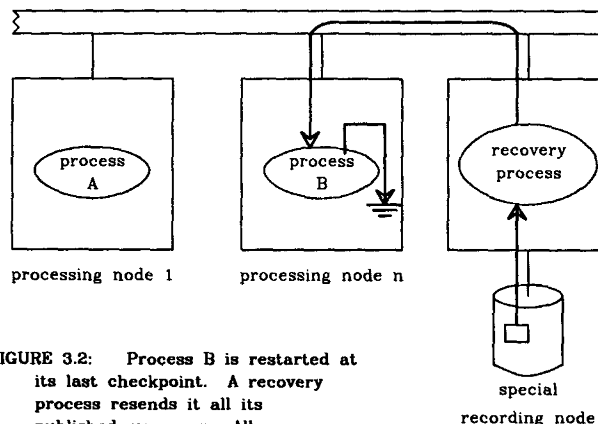

FIGURE 3.2: Process B is restarted at its last checkpoint. A recovery process resends it all its published messages. All messages resent by process B are discarded.

(1) Pick a node for the process to restart on. Unless the processor has failed, this will be the same node that the process used to be on. If the processor has failed, it would be best to have one or more spare processors on the network that could assume the identities of failed processors. Otherwise, in addition to recovering processes for a failed processor, it will be necessary to migrate them to other nodes.

(2) Send a message to the node's kernel telling it to start up a process with the specified process id and set it in the recovering state. Transmit the information from the latest checkpoint to allow the kernel to regenerate the process to the time of that checkpoint. Also, notify the kernel when to stop ignoring messages from the process. The process can then resume running.

(3) Send to the recovering process all messages that it had received between the time of its last checkpoint and the subsequent crash.

It is up to the kernel on the new processor to ignore all messages sent by the recovering process until the process sends a message it had not sent before the crash.

As stated above, it is possible that a process will have to be recovered on a different processor. This is essentially process migration combined with recovery. [Powell and Miller 83] explains in detail a mechanism for migrating processes from a source processor to a destination processor in a distributed system. Since the recorder has the requisite process state, it can mimic the actions of the source processor in order to restart the crashed process on another node. It is also the duty of the source processor to forward some messages following the actual migration of a process. Since the former location of the process is not responding to messages, the recorder can forward them itself without interference.

4. Related Work

Publishing provides a system with reliable message delivery, the guarantee that all messages will eventually be delivered despite crashes of either sender or receiver. A number of systems currently support reliable messages, including the Reliable Network[Hammer and Shipman 80], Tandem's Non-Stop system[Bartlett 81], the Auregen Computer System[Borg et al 83], and Fred Schneider's broadcast synchronization protocols[Schneider 83]. Although each of these systems has some similarity to publishing, they all differ from it in one significant way: their mechanisms are all distributed. In all these systems, the application processors must expend resources, both CPU and memory, to save the redundant information that will be used in the event of crash recovery. Publishing, by passively listening to the network, allows this work to be centralized in one recorder processor. In many cases this will decrease the amount of the system power consumed by the reliability mechanism.

The centralization can also, perhaps counter-intuitively, increase the reliability of the system. The broadcast medium is a single point of failure for local broadcast networks. Nonetheless, the medium can usually be made significantly more reliable than other parts of the system. Increasing the reliability of one special purpose processor, perhaps by adding an uninterruptable power supply or replicating the processor, can be cheaper than improving the reliability of all the processors in the system.

Centralization also means the often complex algorithms for recovery can be implemented once, and in a straight-forward way. This contrasts with the Tandem system, which requires servers to interact with the recovery mechanism, and RelNet, which requires complicated protocols and cooperation between nodes to spool messages destined for crashed processors.

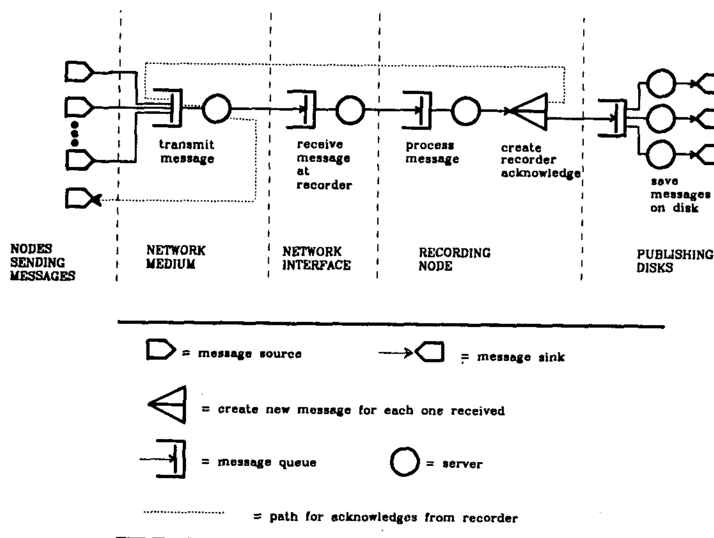

FIGURE 5.1: Queuing Model of Publishing System

To build such a recorder, we assume the ability to listen to all messages on a broadcast network. For at least one network, the Ethernet, a number of such listeners exist: In METRIC[McDaniel 77], a passive recorder was attached to the Ether to record performance information generated by programs on the network. [Shoch and Hupp 79] mentions a "passive listener set to receive every packet on the net." [Wilkinson 81] used a passive Ethernet listener to resolve concurrency conflicts for a data base system, and suggested using this listener to record recovery information in the same fashion as publishing.

5. A Queuing Model Simulation

In order to get a ball park figure for resource requirements, we used a queuing system model to simulate a system. The model was an open queuing model and was solved using IBM's RESQ2 model solver[Sauer et al 81].

The system modeled was that depicted in Figure 3.1. Its open queuing model equivalent is depicted in Figure 5.1. The processing nodes are represented as message sources. Messages are assumed to be delivered when they are broadcast, so the receiving nodes do not appear in the model. A return path was included from the recovery node to the network to take care of acknowledgments from the recording process.

Sending nodes feed three types of messages into the system: short messages (128 bytes long), long messages (1024 bytes long), and checkpointing messages (1024 bytes long). The checkpoint traffic was generated under the assumption that a process is checkpointed whenever its published message storage exceeds its checkpoint size. This policy tries to balance the cost of doing a checkpoint for a process against the disk space required for published message storage. The results were checkpoint intervals between 1 second for 4k byte processes during high message rates and 2 minutes for 64k byte processes during low message rates.

Table 5.1 shows the values of hardware parameters chosen from our computing environment at Berkeley, which consists of VAX 11/780's connected via a 3 megabit/sec Ethernet.

PARAMETER	VALUE
Ethernet interface interpacket delay	1.6ms
Network Bandwidth	10 megabit per second
Disk Latency	3 ms
Disk Transfer Rate	2 megabyte per second
Time to Process Packet	0.8 ms

TABLE 5.1: Simulation Parameters

The operating points for the model were determined by three load parameters:

1) load average - the number of processes per processor.
2) state sizes - the sizes of the changeable state of a process.
3) message traffic - the amount of network communication.

These parameters were estimated by measuring the most heavily utilized research VAX at UCB over the period of a week. The load average and state sizes were directly measurable. Figure 5.2 shows the distribution of state sizes.

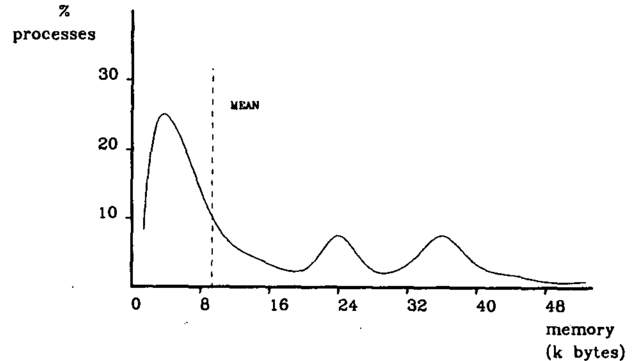

FIGURE 5.2: State Size Distribution for Unix Processes

The message traffic was not measurable, however, since no distributed system existed at UCB at the time. Instead, the following method was used to convert measurements of the single processor into a distributed equivalent. All system calls were assumed to translate to short messages sent to servers. All I/O requests were assumed to represent long messages sent to devices or other processes. The sizes of these messages were estimated to be 128 and 1024 bytes respectively.

Using these measurements, four operating points were established, one representing the mean of each parameter and the other three representing the measurements when each of the parameters was maximized. Table 5.2 shows the parameter values for those operating points.

Description	Load Average	Disk Access	System Calls
Maximum Load Average	23	19/sec	106/sec
Maximum Disk Access Rate	6	43/sec	111/sec
Maximum System Call Rate	6	5/sec	860/sec
Mean Value for All Parameters	7	13/sec	118/sec

TABLE 5.2: Simulation Operating Points

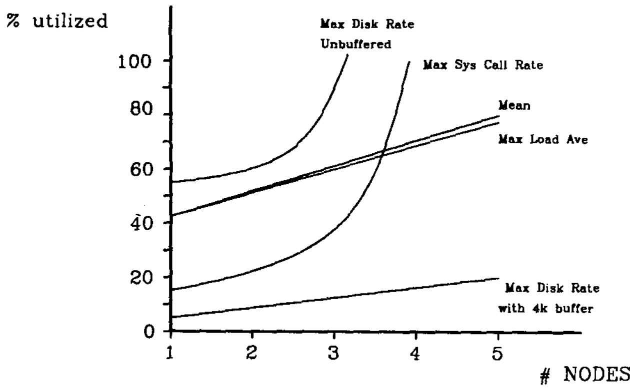

FIGURE 5.3a: Disk Utilization

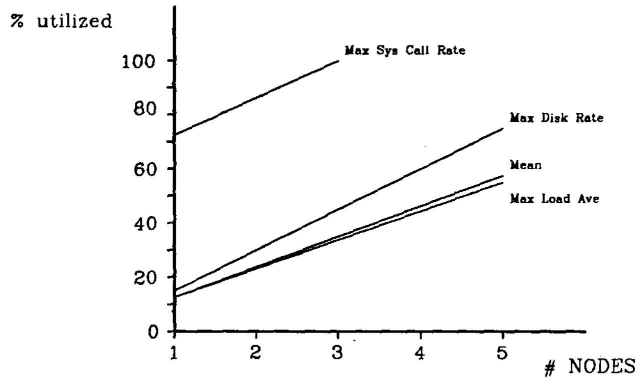

FIGURE 5.3b: Recovery Node Utilization

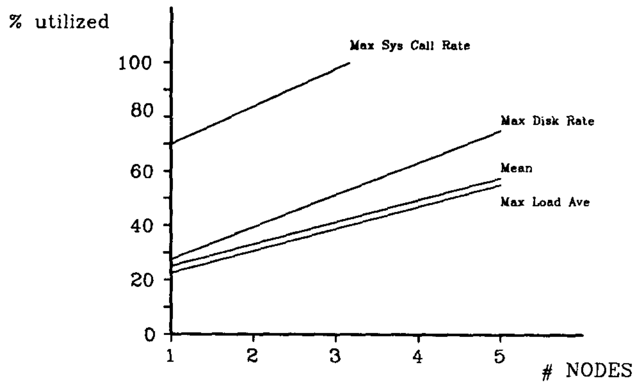

FIGURE 5.3c: Network Interface Utilization

The system was simulated for from 1 to 5 processing nodes and from 1 to 3 disks at the publishing node. Figure 5.3 shows plots of the utilization of the publishing node processor, its disk system and its network interface.

The system stayed within physical limits with two exceptions. The first was the saturation of the disk system used with the maximum long message rate. This saturation was removed by allowing messages to be written out in 4k byte buffers rather than forcing one disk write per message. The second problem occurred at the high system call rate operating point. If this rate persists for more than a few seconds, all three subsystems saturate when more than 3 processing nodes are attached to the system. This saturation cannot be removed by any simple optimizations; luckily, this operating point was not a long-lived phenomenon in the system measured. Therefore saturation at this point should offer no significant problems.

From this simulation we concluded that the simple system was viable for at least 5 nodes. We found no cases in which much buffer space was needed in the recording node (at most 28k bytes). The worst case for checkpoint and message storage was 2.76 megabytes. However, this was constrained by our choice of checkpoint intervals. Making less frequent checkpoints increases the required storage by the amount of extra message traffic in the longer intervals between checkpoints.

6. Adding Published Communications To A Distributed System

An initial implementation of published communications has been added to DEMOS/MP, a multiprocessor version of the DEMOS system originally created for the CRAY-1[Baskett et al 77,Powell 77]. Because it is an experimental system, we simulate both the hardware and the workload required to test these ideas. Since we are primarily interested in whether or not such a system could be created and how it would work, the experimental environment gave us results more easily and with less disruption of normal work than a more realistic environment would have.

6.1. Experimental Environment

DEMOS/MP runs on a number of loosely connected Z8000-based nodes, connected via point to point parallel links. The same code also runs under VAX UNIX[Ritchie and Thompson 78], where we have created a simulated multiprocessor environment. Generally, all code except low level device drivers is developed and debugged on the VAX system. The code can then be moved without change to the Z8000 systems.

Since we have no reliable broadcast network or passive network listeners, we simulate them. On the Z8000s, we accomplish this by making the recording node the hub of a star configuration. Any messages received incorrectly by the recorder are not passed on. In the version running under VAX UNIX, an Acknowledging Ethernet is simulated using a low level protocol on top of the datagram sockets provided by Berkeley's 4.1c UNIX implementation. Any messages not immediately acknowledged by the recorder are ignored by the receiver and will subsequently be resent by the sender.

6.2. Changes to the DEMOS/MP Kernel

Since the idea is to passively record recovery information, the changes to the normal nodes were few. Most significant was the simplest change, that of causing all messages (including intra-node) to be broadcast on the network. Since our processes are spread rather thinly across the nodes, most messages were already going over the network, and the effect on performance was not noticeable.

Applications that have heavy intra-node traffic could notice a significant performance loss if all messages are published. One way to reduce this problem is to treat a group of processes as a single process. Messages within the group are not published. However, all of the processes in the group must be checkpointed and recovered as a unit.

A few additions were made to allow the kernel to notify the recovery manager of significant events such as process creation and termination (normal or otherwise).

A simpler, but less flexible message forwarding mechanism was implemented. If the recorder detects an incorrectly routed message, it sends to the kernel of the sender a request to update the address field of the sending process's link.

6.3. The Recording Node

The recording node runs a modified DEMOS/MP kernel. This kernel includes:

- the checkpoint process
- the publishing process
- the recovery manager
- the recovery processes
- the garbage collector

These functions were put in the kernel to avoid interfering with message communication.

The crash detection processes run as user processes and require no change to the DEMOS system. They are exactly as described in the previous sections.

6.4. Status of the Publishing Experiment

This implementation is the same as the system described in previous sections with one exception: at present no checkpointing is done after the process has been started. All recovering processes are restarted at the beginning and all published messages are subsequently replayed to them. Checkpointing is being added and appears to present no particular problems.

A number of experiments still remain to be performed. Questions of storage management and reliability in the recorder must be addressed, including protocols for replicated recorders. In addition, mechanisms for improving the performance for intra-processor messages, such as treating all processes in a machine as one process, should be explored.

7. Conclusions

We began by looking for a mechanism that could centralize the reliability and recovery aspects of a distributed system with a broadcast network. Starting with a model for processes and their interactions, we identified the state to be recovered and the information needed to restore it. Publishing appears to fulfill the requirements for a passive recorder and a recovery mechanism that can handle any process at any time.

With the simulation and experiments described above, we have shown that published communications is a feasible and practical mechanism. Our implementation revealed that it can be added naturally to many message based systems. We have also shown, via our queuing model, that the resource requirements necessary for publishing are reasonable for a class of systems typical of many local area networks.

8. References

[Arens 81]
G. C. Arens, "Recovery of the Swallow Repository," Technical Report 252, MIT Lab for Computer Science (Jan. 81).

[Bartlett 81]
J. Bartlett, "A NonStop Kernel," *Proc. of 8th ACM Symposium on O. S. Principles,* pp. 22-29 (Dec 81).

[Baskett et al 77]
F. Baskett, J. H. Howard, and J. T. Montague, "Task Communication in DEMOS," *Proc. of 6th ACM Symposium on O. S. Principles,* pp. 23-32 (Dec 1977).

[Borg et al 83]
A. Borg, J. Baumbach, and S. Glazer, "A Message System Supporting Fault Tolerance," *Proc. of 9th ACM Symposium on O. S. Principles,* (Oct 1983).

[Farber et al 73]
D. Farber, J. Feldman, F. Heinrich, M. Hopwood, K. Larson, D. Loomis, and L. Rowe, "The Distributed Computing System," *Proc. of 7th Annual IEEE Computer Society International Conference,* pp. 31-34 (Feb 1973).

[Fraser 79]
A. G. Fraser, "Datakit - a modular network for synchronous and asynchronous traffic," *Conference Record, International Conference on Comm.,* pp. 20.1.1-20.1.3 (June 1979).

[Gray 78]
J. N. Gray, "Notes on Database Operating Systems," pp. 393-481 in *Operating Systems: An advanced course,* Vol 60 of Lecture Notes in Comp. Sci., Springer-Verlag (1978).

[Hammer and Shipman 80]
M. Hammer and D. Shipman, "Reliability Mechanisms for SDD-1: A System for Distributed Databases," *ACM TODS* **5**(4) pp. 431-466 (Dec 1980).

[Lampson and Sturgis 79]
B. Lampson and H. Sturgis, "Crash Recovery in a Distributed Data Storage System," Technical Report, XEROX PARC (1979).

[McDaniel 77]
G. McDaniel, "METRIC: a kernel instrumentation system for distributed environments," *ACM Proc. 6th Symposium on O.S. Principles*, pp. 93-99 (Dec 1977).

[Metcalfe and Boggs 76]
R. M. Metcalfe and D. R. Boggs, "Ethernet: distributed packet switching for local computer networks," *CACM* **19** pp. 395-404 (July 1976).

[Powell 77]
M. Powell, "The DEMOS File System," *Proc. of 6th ACM Symposium on O. S. Principles*, pp. 33-42 (Dec 1977).

[Powell and Miller 83]
M. Powell and B. P. Miller, "Process Migration in DEMOS/MP," *Proc. of 9th ACM Symposium on O. S. Principles*, (Oct 1983).

[Randell 78]
B. Randell, "Reliable Computing Systems," pp. 282-292 in *Operating Systems: An advanced course*, Vol 60 of Lecture Notes in Comp. Sci., Springer-Verlag (1978).

[Ritchie and Thompson 78]
D. M. Ritchie and K. Thompson, "UNIX Time-Sharing System," *Bell System Technical Journal* **57**(6) pp. 1905-1929 (1978).

[Sauer et al 81]
C. H. Sauer, E. A. MacNair, and J. F. Kurose, "Computer/Communications System Modeling with the Research Queuing Package Version 2," Technical Report RA 128 (38950), IBM Watson Research Center (Nov 1981).

[Schneider 83]
F. B. Schneider, "Synchronization in Distributed Programs," *ACM Transactions on Programming Languages and Systems* **4**(2) pp. 179-195 (1983).

[Shoch and Hupp 79]
J. F. Shoch and J. A. Hupp, "Measured Performance of an Ethernet Local Network," *Local Area Communications Network Symposium*, (May 1979).

[Skeen and Stonebraker 81]
D. Skeen and M. Stonebraker, "A Formal Model of Crash Recovery in a Distributed System," *Proc. 5th Berkeley workshop on Distributed Data and Computer Networks*, (Feb 1981).

[Svobodova 80]
L. Svobodova, "Management of Object Histories in the Swallow Repository," Technical Report 243, MIT Lab for Computer Science (July 1980).

[Tokoro and Tamaru 77]
M. Tokoro and K. Tamaru, "Acknowledging Ethernet," *Fall Compcon proceedings*, pp. 320-325 (1977).

[Verhofstad 78]
J. S. M. Verhofstad, "Recovery techniques for database systems," *ACM Computing Surveys* **10**(2) pp. 167-196 (June 1978).

[Wilkinson 81]
W. K. Wilkinson, "Database Concurrency Control and Recovery in Local Broadcast Networks," Ph.D. Thesis, University of Wisconsin at Madison (1981).

[Wolf and Liu 78]
J. Wolf and M. Liu, "A Distributed Double-Loop Computer Network (DDLCN)," *Proc. Seventh Texas Conference on Computing Systems*, pp. 6.19-6.34 (1978).