# PUBLISHING:
## A Reliable Broadcast Communication Mechanism

by

*David Leo Presotto*

# ACKNOWLEDGEMENTS

A number of people, either directly or indirectly, helped in this research and influenced its direction. The quality of this thesis is mostly due to the helpful criticisms and suggestions of my committee. Michael Powell, my advisor, originally suggested this thesis. As my advisor, he has given me many useful insights into Computer Science and has taught me the meaning of research. During the implementation, as one of the original authors of DEMOS, he provided much needed knowledge and experience. Elwyn Berlekamp, besides his helpful editorial criticisms, was responsible for giving me a new viewpoint from which to consider system reliability, that of algebraic coding theory. Lucien LeCam provided many useful suggestions concerning the style and form of this thesis.

In addition to my committee, two other people were directly involved in this research. Bart Miller, my co-researcher in the DEMOS/MP project, helped in many of the kernel changes necessary for the implementation of publishing and provided the metering system used to obtain the DEMOS/MP measurements in Chapter 5. He also acted as a sounding board for many of the ideas used in this thesis. Domenico Ferrari supervised the queuing simulations in Chapter 5 and offered many suggestions concerning their evaluation.

This research could not have been performed without a conducive and stimulating environment. For this, I thank all the members of the Computer Science Division at Berkeley. In particular, I wish to thank the members of the OSMOSIS and CSR groups for their participation in thought provoking seminars. I also wish to thank Bill Joy and Sam Leffler for the interprocess communications in the 4.2 Berkeley Software Distribution of UNIX.

A thesis is more than a research contribution, the proposal and evaluation of new idea. It is a rite of passage that gets exponentially more difficult as one approaches the end. I could not have endured it without the help and support of my girl friend, Caryl Carr.

This research was supported by National Science Foundation grant MCS-8010686, the State of California, 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.

# TABLE OF CONTENTS

# LIST OF FIGURES

# PUBLISHING: A Reliable Broadcast Communication Mechanism

*David Leo Presotto*

Ph.D.

Computer Science Division
Department of Electrical Engineering
and Computer Sciences

*Sponsors*
National Science Foundation
State of California
Defense Advanced Research Projects Agency

Michael L. Powell
Committee Chairman

## ABSTRACT

Today's computing environment is becoming increasingly more distributed. Due to their flexibility and inherent parallelism, distributed systems can be both more personalized and more powerful than centralized computers. However, with their qualitative and quantitative increases in complexity, distributed systems are more susceptible to failure. One way of increasing the reliability of these systems is to recover from faults before they lead to failures.

A number of methods have already been developed to perform fault recovery in distributed systems: recovery lines, recoverable transactions, and shadow processes. In order to effect time-bounded recovery, each of these methods requires interaction with the user application. This interaction may sometimes fit naturally into the application program. However, in many instances, the lack of transparency of the recovery system may significantly restrict the application programmer's style. Also, existing programs need to be rewritten to make use of these methods.

Making recovery transparent to the program being recovered is, in the most general case, a difficult and, perhaps, unsolvable problem. However, by considering only message-based systems, the problem can be greatly simplified. Message-based systems, especially those connected by low cost broadcast media, represent the most common type of distributed system. We have developed a new communications model for such systems called published communications. In this model, a passive recorder reliably stores all messages broadcast onto the network. Coupled with the idea of deterministic programs, published communications allows the transparent recovery of processes in a

distributed systems.

In order to evaluate the consistency of the model with message-based systems, an initial implementation has been added to an existing message-based system, DEMOS/MP. A number of minor changes were necessary to conform DEMOS/MP to the model. However, it was not necessary to change any programs already running on the system.

The performance of published communications was determined both by evaluating a queuing model of the system under different loads and by measuring the DEMOS/MP implementations. The simulation shows that recorder, constructed from current technology, can support a system of up to 115 users. The measurements show that the steady state costs of publishing messages is low.

# CHAPTER 1

## The Recovery Problem

Ever lowering costs and rapid advances in technology have made computer science one of the fastest changing areas in science. Today's computing environment is becoming increasingly more decentralized. Low cost processors and peripherals allow users to have personalized systems suited to their particular needs. At the same time, high bandwidth, low latency computer communications, especially those for local area networks, allow these users to access jointly held resources.

A popular use for these decentralized systems is the automated office. In an automated office each person uses a small, usually low cost, personal computer configured to his/her particular needs. For example, secretaries may have inexpensive processors to perform word processing functions such as letter preparation. Engineers, on the other hand, may need processors outfitted with mice and high quality displays for computer aided design. More expensive resources such as high quality printers and telex machines would be accessible by all of these computers via a network. Figure 1.1 shows a typical configuration for the XEROX STAR, an office system marketed by the XEROX Corporation. (The diagram is from a XEROX publicity brochure. It was generated on a XEROX STAR workstation.)

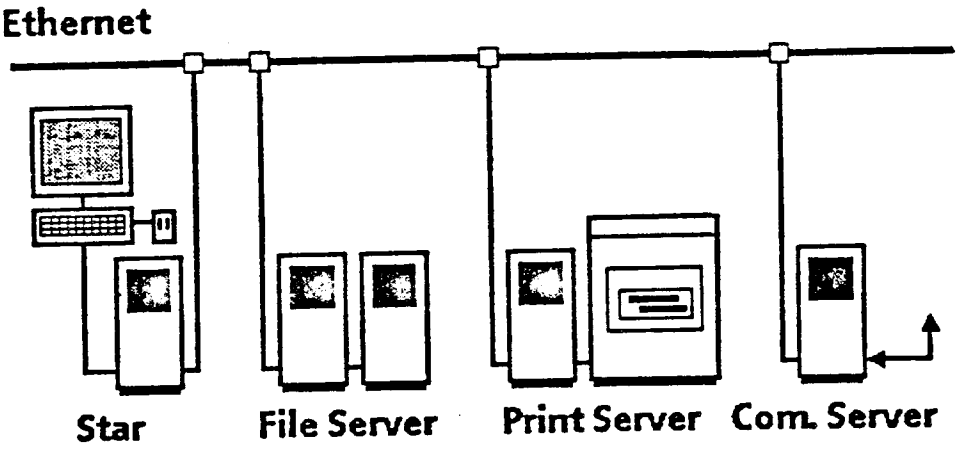



**Figure 1.1: A XEROX STAR Configuration**

Decentralized systems are also used to increase the speed of computation. Often, computations can be decomposed into a number of smaller computations, each of which may execute in parallel on a separate computer. These computations may take much less time than the equivalent sequential computation on a single computer, if the time taken up by management of and communications between the parts is low enough. Recently, such parallel computations have become important in the field of cryptoanalysis. Many codes that were once thought to be secure have been shown to be susceptible to exhaustive analysis attacks by large numbers of computers. Diffie and Hellman[Diffie and Hellman 77], for example, have shown how to break the National Bureau of Standards data encryption standard (NBS/DES) [NBS 75a,NBS 75b] using a network of one million computers. A controlling computer partitions the search space into one million parts and notifies each of the others which part it must search. The computers then exhaustively search their partitions of the space. When one finds a solution, it informs the controller and all are stopped and given a new problem. With parallel computation, the NBS/DES code might be broken in less than a day.

These two examples are illustrative of a type of computing, *distributed computing*. Specifically, a distributed computation is one in which a number of concurrently executing programs cooperate to perform the computation. In the decentralized systems just described, distributed computations are used to take advantage of resource sharing and overlapped execution. Programmers

also distribute computations for the same reasons they use subroutines, to modularize their programs. Separating a computation into a number of cooperating programs, each running in its own address space, can often lead to better understanding and easier debugging than would a monolithic organization. The DEMOS operating system, described in Chapter 4, is an example of a system organized as a distributed computation for just this reason.

For distributed systems, as for other types of systems, reliability is an important issue. Brian Randell provides an excellent description of reliability[Randell et al 78]:

> The ***reliability*** of a system is taken to be a measure of the success with which the system conforms to some authoritative specification of its behavior. Without such a specification, nothing can be said about the reliability of the system. When the behavior of the system deviates from that which is specified for it, this is called a ***failure***. A failure is thus an event, with the reliability of the system being inversely related to the frequency of such events. Various formal measures related to a system's reliability can be based on the actual (or predicted) incidence of failure (see, for example, [Shooman 68] ). These measures include Mean Time Between Failure (MTBF), Mean Time Between Repair (MTBR), and ***availability***, that is, the fraction of the time that a system meets its specification.

In some distributed systems, increasing reliability is a critical problem. For example, Hellman and Diffie expect that their system would normally have a mean time between failure of 6 minutes. Since they expect the system to take a full day to crack one code, this reliability is unacceptable and must be increased.

To understand how reliability can be increased, one must understand the difference between a failure and a fault. We have defined a failure as a deviation from an authoritative description of the system. A *fault* is the immediate mechanical or algorithmic cause of the failure. For example, an alpha particle may cause an illegal state change in a processor register or memory location. If a program then uses that register in a calculation, the result of the calculation would be incorrect. Here, the register state change is the fault and the incorrect calculation is the failure.

Obviously, a system's reliability can be increased by decreasing the fault rate. For instance, lead shielding will decrease the number of alpha particles striking the register. However, in trying to reduce the the rate at which a particular fault occurs, we must take into account all potential causes of the fault. Different causes may be related such that avoiding one may exacerbate another. For instance, if the lead shielding made it difficult to cool the processor, the resultant overheating could cause more faults, canceling the advantage of the shielding.

Assuming that some faults are unavoidable, we can improve reliability by preventing the fault from leading to deviation from the "authoritative definition", that is, we can isolate the fault and undo its effects before it causes the system to malfunction. We call this action *recovering* from the fault. Fault recovery can occur at many different levels of a system. For instance, the illegal register state can be detected when the register is accessed by

storing a checksum of the register in the processor state. The fault could then be corrected immediately using the redundant information. If this is not possible, the entire calculation would need to be aborted and restarted to make sure no erroneous results are propagated.

As we show later in this chapter, recovery in a distributed system is considerably more difficult than in a centralized system. It is the subject of this thesis to study this problem and to present a solution.

## 1.1. Formal models and definitions

Before developing a solution to the problem, we need a more precise definition of what the problem is. In this section, we define our domain of interest, distributed systems, the problem, faults, and the solution, fault recovery.

### 1.1.1. A model of a distributed system

We have already defined what distinguishes a distributed system. It is any system which makes use of or supports the use of distributed computations. In this section, we present a model of a distributed system consistent with this definition.

A distributed computation is a set of concurrently executing programs which cooperate to perform a computation. These programs communicate with each other and with peripheral and storage devices, such as clocks, terminals, printers, and disks. Both the executing programs and the devices can be modeled by what we call *processes*. A process is an object that contains state information, that can change its state information, and that can interact with other processes by changing their states. Processes are capable of concurrent, independent, and possibly unsynchronized execution. Execution is the act of changing the state of a process.

For processes representing executing programs, the state consists of:

- the writable address space of the process, normally the variables used by the program
- information related to the sequencing of the program such as the program counter and the execution stack
- information managed by other parts of the system for the process, such as unread messages or device buffers

The process state can change either as a result of execution of the next instruction of the program that it represents or via interactions with other processes.

A process representing a device is made up of both the device and any software used to control the device (often called a device driver). A disk process, for example, is the disk itself, its controller, and the disk driver running on the processor to which the disk is attached. Its state is made up of the variables used by the driver, the internal registers of the controller, and the information written on the disk. As another example, a line printer is a process whose state consists only of the variables used by its driver and the registers in the printer.

If a process is considered a "black box", we can characterize its behavior by observing its interactions with other processes. A process is said to be *deterministic* upon its input interactions if, each time it is started and receives the same input interactions, it will produce the same output interactions.

Most processes are, by this definition, deterministic. The reasons for this are twofold. First, deterministic processes are much easier to debug since problems occurring in these processes can be easily reproduced. Second, traditional processors and programming languages are designed to support sequential deterministic execution. Therefore, it is natural to write deterministic programs for these processors and with these languages.

Nevertheless, non-deterministic processes do exist. In the case of device processes, the cause is the non-determinism of the devices themselves. For example, a disk process may schedule requests according to the position of the movable disk arms in order to improve throughput. The ordering of output interactions from the disk may thus depend not only on the input interactions but also on disk arm position and varying speeds of disk arm movement. In the case of processes representing programs, non-determinism is caused by generating output dependent on the amount of execution time allowed the program between inputs. For example, in UNIX, a typical way of measuring the idle time of the system is to create a low priority process that will execute whenever no other processes can execute. The program consists of a tight loop, whose execution time can be accurately determined. The program can, therefore, determine its running time from the number of iterations through the loop. On request, it returns its execution time. Such a program is non-deterministic since its outputs depend on the timing of requests to it and not upon the requests themselves.

We claim that non-deterministic processes are few in number. This is partially substantiated by our examination of UNIX and DEMOS processes. In both cases non-determinism existed only in a few device drivers and in programs which were easily changed to be deterministic. Therefore, in this thesis, we present a general recovery mechanism only for deterministic processes. We assume that other mechanisms can be used for the few non-deterministic

processes that exist or that those processes can be transformed into deterministic ones.

### 1.1.2. Fault classification and crashes

For the purposes of our study, we can classify a fault according to two characteristics: whether or not the fault is detected, and whether or not it is deterministic.

A fault is considered *undetected* if it is not detected by the affected process or by the system before it can alter the state of some other process. Undetected faults correspond to what Lampson and Sturgis call "unexpected undesired events"[Lampson and Sturgis 79]. Since they do not cause the recovery mechanism to be initiated, these faults will become failures.

A *deterministic* fault is one that is algorithmic in nature and thus a property of the process itself. Since we are assuming deterministic processes, such faults will recur whenever the process is recovered following detection of the fault. Deterministic faults can be recovered from only if the programmer supplies alternate algorithms to be used should a fault be detected in the program. Thus, deterministic faults can be avoided only by a non-deterministic program. Since, as we state later in this chapter, one of our goals is transparent recovery, we consider only non-deterministic faults in this thesis.

A *crash* is defined as the halting of a process on the detection of a fault. 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 fault to a crash of all the processes affected by the fault.

### 1.1.3. Process Recovery

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

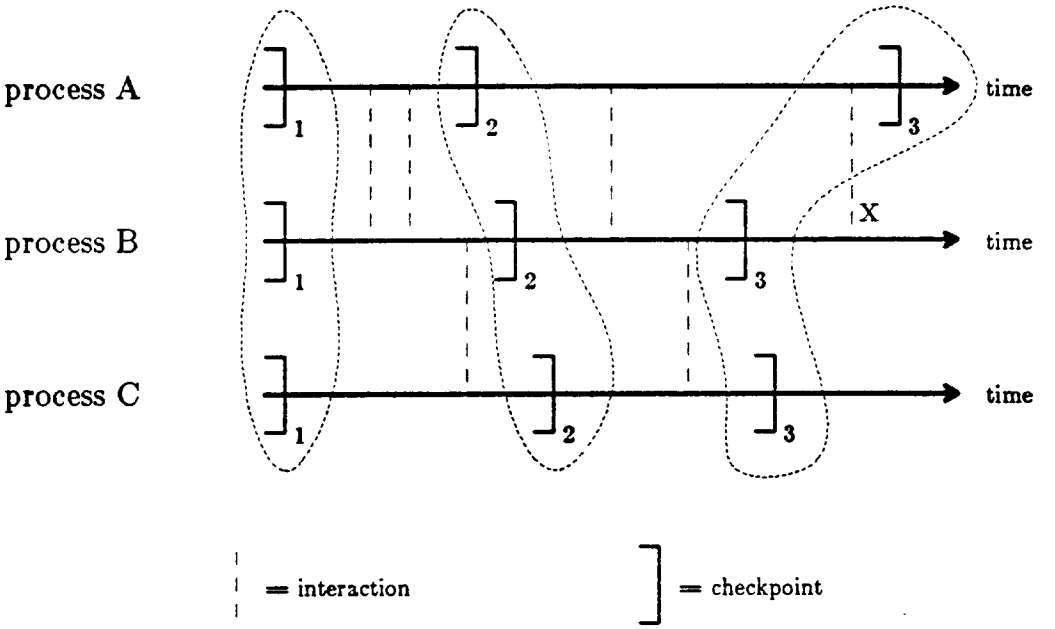
Information is preserved across a crash in a non-volatile storage facility, that is, one that has low probability of being altered by the crash. This is usually achieved by storing the information on devices whose failure modes are decoupled in some way from those of the other elements of the system.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

To allow the reconstruction of consistent states of processes, it is common to occasionally make copies of part or all of the process state. Making copies more often can reduce the time to perform recovery. In this thesis, 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 interacting processes, and the recovered processes must be brought back to a consistent state with processes that did not fail.

### 1.1.4. The problem of distributed recovery

For an isolated process, 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 1.2, adapted from[Randell 78].




*Checkpoint sets 1 and 2 are consistent.*
*Checkpoint set 3 is not.*

**Figure 1.2: Process Checkpointing**

The horizontal axis represents time (increasing left to right). The dashed vertical lines represent interactions between two processes. During an interaction, both processes may communicate information to each other. The square brackets represent the checkpoints of individual processes. The following rule can be used to determine consistent checkpoints:

> **Rule 1:** Since processes are deterministic upon their interactions, checkpoints for any two processes are consistent as long as the processes do not interact with each other between the times the checkpoints are taken.

Represented graphically, if a line connecting a set of checkpoints intersects no interaction lines, then those checkpoints are consistent. We call this line (or set of checkpoints) a *recovery line.*

Figure 1.2 shows two sets of consistent checkpoints. The checkpoints labeled 1 represent the starting states 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 have seen the results of the interaction labeled X, but process B will not. Thus, checkpoint set 3 could not be used; instead, it would be necessary to go to checkpoints older than the most recent set.

Any recovery method for distributed computing must therefore determine consistent states for all recovered processes. It must also ensure that those states are consistent with the current states of any non-recovered processes. An inconsistent checkpoint can be used only if the interaction accounting for the inconsistency can be reproduced in order to eliminate it.

### 1.1.5. An ideal recovery mechanism

Having defined the problem, we can now describe the properties an ideal recovery mechanism should have. The environment we are aiming for is the general purpose distributed system supporting many users. Special purpose or single user environments are less interesting to us since, in this type of system, custom mechanisms designed for the specific application are usually better than any general purpose mechanism.

In a general purpose system, a recovery mechanism should exhibit the following properties:

- **Independent process recovery** - Recovery should require the minimum possible perturbation to non-failing parts of the system. This means that processes should be individually recoverable, despite interactions with other processes. Recovery may slow down non-failing processes, but it should not cause them to be restarted.
- **Transparency of recovery mechanism to programs** - Programs should not need to be aware of or interact with the recovery mechanism. It should not be necessary for the programmer to change his programming style to please the system. This allows naive users to write recoverable distributed computations without having to learn how to use the recovery system. It also allows existing programs to be made recoverable without being changed.
- **Arbitrarily bounded recovery time** - It should be possible for the programmer to specify maximum recovery times for individual processes. This means that processes should be checkpointed independently. This allows checkpoint frequencies to be specified on a per process basis, allowing individual recovery time bounds to be placed on the processes. This also ensures that the steady state actions of the recovery system are simple and efficient when no processes are being recovered.
- **Low cost** - Though the cost of storing the recovery information will be noticeable, it should not be excessive. No recovery mechanism will be widely accepted if it requires too sizable a portion of the system's resources.

These properties represent the ideals that this thesis hopes to approach.

## 1.2. Thesis plan

Chapter 2 reviews a number of current recovery methods. It discusses their advantages and weaknesses in the light of the ideal system.

Chapter 3 introduces our solution to recovery, published communications. A design of a published communications system is outlined.

Chapter 4 reports our work in adding published communications to an existing system, DEMOS/MP. Necessary changes to the system structure are discussed.

Chapter 5 discusses the performance of published communications in a distributed environment. The chapter is divided into two parts. The first presents a queuing model for an Ethernet-based system with up to five processors. The model is solved numerically. The second part presents measurements of the DEMOS/MP system both before and after publishing is added.

In Chapter 6, we offer network specific solutions for ensuring that all messages are published before being used. We also present some variations on publishing.

Chapter 7 concludes the thesis by summarizing what we have accomplished and by suggesting future research that can be built upon our work.

# CHAPTER 2

## Survey of Distributed Recovery

Chapter 1 described the main problem of recovery, that of returning the faulted system to a consistent state from which it may continue. A number of recovery mechanisms have been built to solve this problem. In general, they can be classified as belonging to one of four classes:

1) recovery lines

2) recoverable transactions

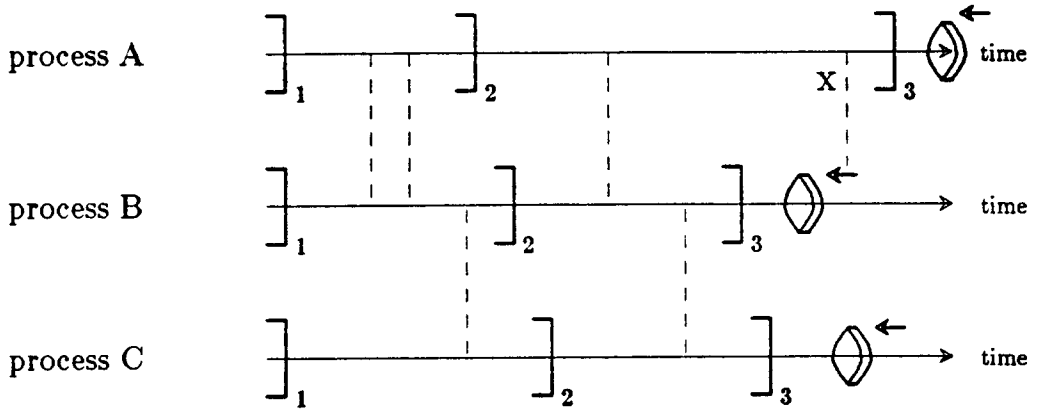
3) shadow processes

4) reliable messages

This chapter presents the first three and explains what their advantages and failings are, in light of the ideal system of Chapter 1. Discussion of reliable messages is left to Chapter 3 since it relates directly to the work presented there.

### 2.1. Recovery lines

One way of recovering processes is to independently generate checkpoints for them and, following a crash, to look through the stored checkpoints for a set of consistent ones at which to restart processes. The set of consistent checkpoints is termed a recovery line[Randell 75]. As we stated in Chapter 1, checkpoints for two processes are consistent if, between the the times the checkpoints are taken, no interactions occur between the processes. A recovery system can detect this, either by monitoring the interactions and checkpoints as they happen or by scanning a history of interactions and checkpoints after a crash has occurred.




*Rings slide left to next checkpoint.*

**Figure 2.1: Finding recovery lines**

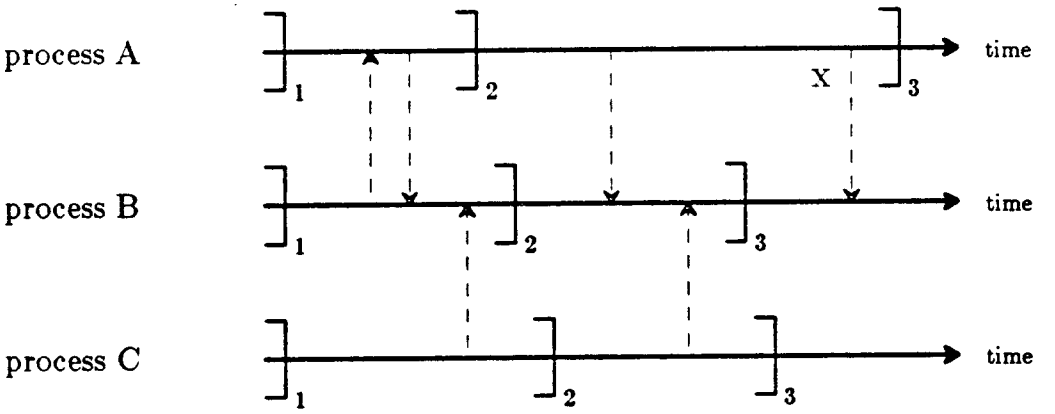
To see how consistent checkpoints can be found, consider Figure 2.1. It represents the history of interactions and checkpoints for three processes. Following a crash, we place a ring on the time axis for each process. Note that, for processes that have not crashed, we can consider the current state of the process a checkpoint. On each iteration of the search algorithm, we let the rings slide back in time (to the left) to the nearest checkpoint on that axis. Whenever a ring slides through an interaction line, all checkpoints to the right of the interaction line for the interacting processes are invalidated, thus allowing rings on those lines to slip further. The algorithm terminates when an iteration occurs without any slips. At this point the rings should be at a consistent set of checkpoints from which we may restart the processes.

As an example, assume that in Figure 2.1 process B has crashed. Its ring will slip to checkpoint 3. In passing interaction X, it will cause process A's ring, on the subsequent iteration, to slip to its checkpoint 2. The algorithm will continue until all three rings stop at the checkpoints labeled 2.

For the recovery system to find recovery lines, it must "see" all interactions and checkpoints. In systems with a formal communications mechanism, the communications mechanism itself can inform the recovery system, making the building of interaction histories transparent to the process. In shared memory systems, the programmer must do the informing since interaction between processes cannot be intercepted by the system. Providing this information does not constrain the programming style in any significant way, so this requirement is not a major violation of the ideal of transparency.

Researchers at the University of Newcastle[Randell 75] have used recovery lines and independent checkpointing in their recovery block systems to recover communicating processes. Although their systems have been centralized, there is nothing to indicate that the same techniques would not also work for decentralized systems. However, they point out one major disadvantage: there is no guarantee, with independent checkpoints, that a recovery line, other than the starting state of all processes, will exist. Therefore, the amount of work to be redone, following recovery, can be arbitrarily long.

Attempts have been made to remedy this situation. The normal recovery line model assumes that any interaction that occurs between checkpoints makes those checkpoints inconsistent. This is because we know nothing about the information flow occurring in the interaction. However, Russell points out that, in the case of message based systems, all interactions are actually directional[Russell 77]. He proposes saving messages and then replaying them to processes after they restart at a checkpoint.




*All interactions are messages.*

**Figure 2.2: Directional interactions**

To understand what Russell means, consider the example in Figure 2.2. Suppose the interaction X were a message from process A to process B. After the crash of process B, we could restart it at checkpoint 3 and replay message X to it without affecting any other processes. However, if the message were from process B to process A we could not do so, since process B would resend the message and process A would receive it twice. Also, since Russell assumes processes to be non-deterministic, there is no guarantee that the process will resend that message after restart.

In this case we can change the consistency requirement for checkpoints (Rule 1) to be:

**Rule 2:** Checkpoints for two processes are consistent if and only if, between the the times the checkpoints are taken, no message is sent from the process with the earlier checkpoint to the process with the later checkpoint.

Unfortunately, Russell's approach lessens but does not solve the problem. We will refer back to this solution later on since it comes very close to the concept of published communications.

## 2.2. Transaction processing

Another way of obtaining a consistent state, which has been widely used in distributed database systems, is transaction processing[Gray 78, Skeen and Stonebraker 81]. Rather than have the system search for recovery lines, all inconsistent states are enclosed within programmer specified transactions. 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 taking effect until another consistent state can be reached. This is done using stable storage. Until reaching the commit point, when all processors taking part in the transaction will have stored away all partial results in stable storage, all updates are considered tentative. If any process fails before that point, all processes abort and the transaction has no net affect. After all processes taking part agree that they have reached the commit point, they are committed to finish even if they crash and are restarted. Applications are designed so that the state of a process between transactions is unimportant and need not be checkpointed. This is equivalent to saying that the the initial state of a process is its checkpoint.

For data base applications, transactions are natural constructs. Even if they were not used for recovery, they would probably be used for concurrency control. In such applications, expressing all computations as transactions, in order to allow them to be recovered, will not cause seriously change his style of programming. This is not true of all applications. In distributed computations where data is pipelined from one process to another, such as graphics applications, the transaction is not a natural unit. In such situations, transactions violate our ideal of programmer transparency.

Transactions also violate the goal of independent process recovery. If a crash occurs in the middle of a transaction, all of the processes involved may have to start over. The result is that much work has to be redone. Liba Svobodova[Svobodova 81] tries to reduce the amount of work redone by hierarchically nesting transactions. In her model, complex transactions can perform their work by decomposing themselves into a number of sub-transactions. The sub-transactions can, in turn, be decomposed into other

sub-transactions and so on. When a fault occurs, instead of restarting the whole transaction, we can restart just the sub-transactions that were in effect at the time of the fault. Unfortunately, this is done at the expense of making the recovery mechanism even more restricting of programming style.

## 2.3. Shadow processes

A recovery method for message-based distributed systems is the concept of shadow processes, introduced in the Tandem Corporation's Non-Stop systems[Bartlett 81]. With shadow processes, we assume that each process (the primary) has another another process (the shadow) that runs in tandem with it. Should the primary process fail, the shadow process is ready to take on any tasks normally performed by the primary. Obviously, the shadow process should run on equipment that is not likely to fail along with that of the primary process. In order for it to take over, the shadow process's state must be kept up to date with that of the primary. This can be done by having the shadow receive all inputs that the primary does and duplicating all its actions. However, in practice, this is not done for two reasons. The first is that some method would have to be found to make sure the shadow did not interfere with the primary. For instance, if the primary outputs to a terminal, we would not want the shadow to repeat the action. Second, by having both processes perform the same actions, to achieve similar throughput, a shadow system would need twice as much computing power as a system without shadow processes. Many people are reluctant to double the cost of their equipment for reliability.

Instead, the shadow process is periodically updated to reflect the state of the primary. In the Tandem system, the shadow's state is updated by messages sent to it from the primary. It is the programmer's responsibility to make sure the shadow is correctly updated. Should the primary crash, all messages that would normally be received by it are rerouted to the shadow. Thus, there is no need for recovery lines or transactions.

Shadow processes achieve a number of the goals of the ideal system. The equivalent of checkpointing, bringing the shadow up to date, is performed independently for all processes. Processes are also individually recoverable. However, the mechanism is not transparent to the user since it is left to the user to update the shadow process.

## 2.4. Summary

In this chapter we have introduced three recovery methods. Each embodies some of the properties of our ideal system at the expense of discarding others. Recovery lines are designed to allow independent checkpointing and transparent recovery. The only requirement is that the recovery system must

be made aware of interactions. In general, this can be done without the active participation of the processes. However, recovery line systems are forced to give up time-bounded recovery for independent checkpointing. There is no guarantee that a recovery line will exist, other than the start state of all processes.

Recoverable transactions give up user transparency and independent checkpoints in the attempt to provide time-bounded recovery. Often, this concession is not very harmful since, in applications like distributed data bases, the transaction is a natural unit of work and programs can easily and efficiently be composed of recoverable transactions.

Finally, shadow processes offer independent process recovery and time-bounded recovery. In return for this, programmer transparency is lost. Communication protocols must be designed and written for the primary to keep the shadow consistent with it.

# CHAPTER 3

## A Solution - Published Communications

In the last chapter we reviewed three classes of recovery systems, each of which failed to satisfy some property or properties of the ideal system. We now present a method belonging to a third class of recovery system, reliable message systems.

In the case of recovery lines and recoverable transactions, often, a crashed process cannot be brought to a state consistent with non-failing processes. Therefore, not only the crashed process but also some non-failing processes must be restarted to rebuild a consistent state. Shadow processes avoid this problem by providing an always up to date replacement for the failed process. The shadow process is consistent with the other non-failing processes so no backup is required. Unfortunately, keeping the shadow process up to date requires either complete duplication of the primary or explicit interaction between primary and shadow.

Reliable message systems are much like shadow process systems. They assume that a failed process can be replaced in a way that is consistent with the state of non-failing parts of the system. However, instead of providing an up to date shadow, they provide a way of independently recovering a failed process to its immediate pre-failure state. Any message sent to a failed process in a reliable message system is guaranteed to arrive once the process has recovered.

This chapter presents a reliable message mechanism called published communications, shows how it meets the goals of the ideal system, and explains how such a system can be designed.

### 3.1. Published Communications

A published communications system is one in which a reliable recorder saves, or *publishes*, in stable storage all process checkpoints and all messages sent to processes. When a process crashes it is recovered by:

1) restarting the process at a previous state, such as its initial state or some subsequent checkpoint.

2) sending to it all messages that had been sent to the original process and not read by the process before the checkpoint was taken. These messages must be resent in the order in which they were received by the original

process.

3) ignoring any messages sent by the recovering process that had been sent by the original process.

As we shall show, publishing is most appropriate in networks using a central communications medium such as a broadcast medium or ring. In such networks, messages can be published by a centralized recorder that passively copies messages transmitted on the network. Such media are important because they are currently the most popular means of interconnecting processors and other resources to form a distributed system.

## 3.2. Meeting the goals of the ideal system

This section looks at each goal of the ideal system and shows how that goal is met by published communications.

### 3.2.1. Independent process recovery

To restart a process without affecting non-failed processes, we must be able to recover the process to the state it had immediately preceding the fault that resulted in its crash. The process can then resume its normal execution. Assuming that processes are deterministic upon their input interactions, we can recreate the state by restarting the process at any checkpoint and recreating any interactions it experienced between the checkpoint and the detection of the fault. Therefore, an encoding that includes a previous state of the process and all interactions since then is sufficient for regenerating the pre-crash state. We can state it as a rule:

> **Rule 3:** 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.

Comparing this to Rule 2 from Chapter 2, we see that we have eliminated dependencies between the checkpoints of two different processes. The validity of a checkpoint now depends only upon the process checkpointed and messages sent to that process. Using this rule we obtain both independent process recovery and independent checkpointing, since no process need synchronize checkpoints with any other.

If we constrain ourselves to message-based systems, then the interactions are messages and can be easily recorded. This is precisely the publishing system which we have described above.

In Rule 3, 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. The system continues publishing messages after a process crashes. A process sending messages to a recovering process does not have to wait for the recovery to complete. It can continue sending messages to the process. The recorder will save these messages and replay them to the recovering process when it is ready to receive them. The recovering process will eventually catch up and be able to accept its messages directly. At that point the recorder will stop acting as a buffer between it and the rest of the world.

### 3.2.2. Transparency of recovery mechanism to programs

For published communications to be invisible to the processes, it is necessary that the steps outlined above be performed without the process's knowledge. In message-based systems, processes send and receive messages via calls to the operating system kernel. If we perform all the publishing actions in or below the system kernel, then the actions will be transparent to the process.

### 3.2.3. Arbitrarily bounded recovery time

The real time necessary to restart a process depends on the time needed to perform the following three steps:

- loading the process with the checkpoint state
- replaying the published messages to the process
- allowing the process to execute from its checkpoint state to its pre-crash state.

In general, the three steps will be occurring in parallel. However, for the sake of finding an upper bound for recovery, we will assume that the three steps are serially executed. Therefore, the recovery time will be bounded by the sum of the times necessary to perform each of these steps or:

$$t_{\max}=t_{reload}+t_{replay}+t_{compute}$$

These times are all elapsed real time. Each of these times can be expressed as a function of load dependent parameters in the following ways:

- $t_{reload}$ has two main components, a fixed time necessary to build system table entries for the process, $t_{cfix}$, and a variable part which is the length of the checkpoint in pages, $l_{check}$, times the time to load a page of the checkpoint, $t_{page}$.
- $t_{replay}$ is the sum of the time needed to lookup and replay each message. A message's lookup and replay time is also made up two parts. The first is a fixed time per message, $t_{mfix}$, for looking up the message and initiating the transfer. The second is the time to transmit the message: the

length of the message in bytes, $l_{msg}$, times the time to transmit a byte of the message, $t_{byte}$.

- $t_{compute}$ is a the real time needed by the process to recompute its precrash state from the checkpoint state. It is the process's execution time since the last checkpoint, $t_{since}$, divided by the fraction of the CPU it can obtain during recovery, $f_{cpu}$.

Thus, at time $\tau$, a process that was checkpointed at time $\tau_0$ and that has received $n_\tau$ messages at time $\tau$ will have a maximum recovery time of

$$\begin{aligned} t_{\max} = \quad & t_{cfix} + t_{page} l_{check} \\ & + t_{mfix}(n_\tau - n_{\tau_0}) \\ & + t_{byte} \sum_{i=n_{\tau_0}+1}^{n_\tau} l_{msg_i} \\ & + (\tau - \tau_0) \frac{1}{f_{cpu}} \end{aligned}$$

The values of the load dependent parameters ($t_{cfix}$, $t_{page}$, $t_{mfix}$, $t_{byte}$, and $f_{cpu}$) can all be be determined empirically by measuring the system under various loads and with varying numbers of simultaneously recovering processes.

The process specific values ( $l_{check}$, $t_{mfix}(n_\tau - n_{\tau_0})$, $\sum_{i=n_{\tau_0}+1}^{n_\tau} l_{msg_i}$, and $\tau - \tau_0$) can be accumulated each time a process is checkpointed or receives a message.

With all these values in hand, the system can dynamically determine $t_{\max}$ for each process. Each time a process receives a message or expends its time slice, the operating system can calculate its new process dependent parameters. It can then use these and the load dependent ones, corresponding to the current load, to determine the process's $t_{\max}$. If the system checkpoints a process whenever its $t_{\max}$ exceeds its specified recovery time, the process can always be recovered in that amount of time.

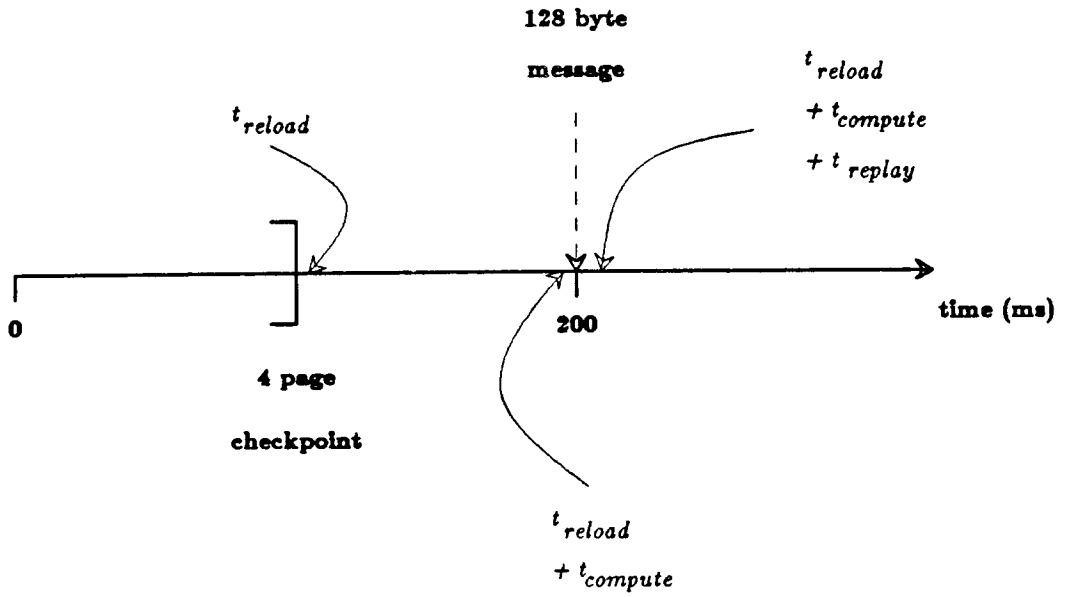



**Figure 3.1: Calculating recovery times**

As a simple example, consider the process with the history shown in Figure 3.1. Let us assume a constant load on the system with

$$t_{cfix} = 100ms$$
$$t_{mfix} = 2ms$$
$$t_{page} = 10ms/page$$
$$t_{byte} = 0.01ms/byte$$
$$f_{cpu} = 0.5$$

Immediately following the checkpoint, the recovery time is just the time to reload the checkpoint or

$$\begin{aligned} t_{\max} &= t_{reload} \\ &= t_{cfix} + t_{page} l_{check} \\ &= 100ms + 4pages * 10ms/page \\ &= 140ms \end{aligned}$$

At time 200 ms, the recovery time depends both on the reload time and the time it takes to redo 100 ms of work ($100ms \frac{1}{f_{cpu}}$) or

$$
\begin{aligned}
t_{\max} &= t_{reload} + t_{compute} \\
&= t_{reload} + t_{since} \frac{1}{f_{cpu}} \\
&= 140ms + \frac{100ms}{.5} \\
&= 340ms
\end{aligned}
$$

Finally, immediately following the message, the recovery time is dependent on reload time, execution time, and the time to replay the message or

$$
\begin{aligned}
t_{\max} &= t_{reload} + t_{compute} + t_{replay} \\
&= t_{reload} + t_{compute} + t_{mfix} + l_{msg}\, t_{byte} \\
&= 140ms + 200ms + 2ms + 128bytes * .01ms/byte \\
&= 343.28ms
\end{aligned}
$$

### 3.2.4. Low cost

In published communications, there are two steady state functions: the generation of checkpoints and the publishing of messages. Checkpoints are generated by the processing nodes. The effect on performance should not be more or less than other systems that use checkpointing. However, since publishing allows independent checkpoints, we are free to provide checkpoint intervals on a per process basis.

A first order approximation to an optimum per process checkpoint interval was determined by John Young[Young 74]. Young's model contains three parameters: the compute time between checkpoints ($T_c$), the time needed to save a checkpoint ($T_s$), and the mean time between failure ($T_f$). Young defines the cost of checkpointing ($t_l$) to be the sum of the time spent checkpointing between failures and the time lost to recomputing following a failure. Assuming that failures arrive exponentially, Young found that, as a first order approximation, $t_l$ can be minimized by choosing $T_f = \sqrt{2T_s T_c}$.

The key to keeping the publishing overhead low is the centralization of the publishing process. On many local area networks (LANs), not only may any node overhear the messages destined for another node, but it may do so 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]. Using this property, we can perform all publishing using a special purpose processor attached to the network. Since this processor performs its function passively, it should not affect the performance of the system in any significant way. This assumes that the recorder can record messages as fast as the processes can inject them onto the network. Chapter 5 presents a queuing simulation that indicates that this is indeed

possible. We also assume that some spare capacity is left both in the recorder and in the network to handle recovery of processes.

### 3.3. Published Communications

A published communications system is one that incorporates the features of the previous section: centralized message publishing and independent recovery by message play back. In this section, we present the elements necessary in a published communications system. Chapter 4 will show how these elements have been added to an existing system, DEMOS/MP.

Figure 3.2 shows published communications system in normal operation.

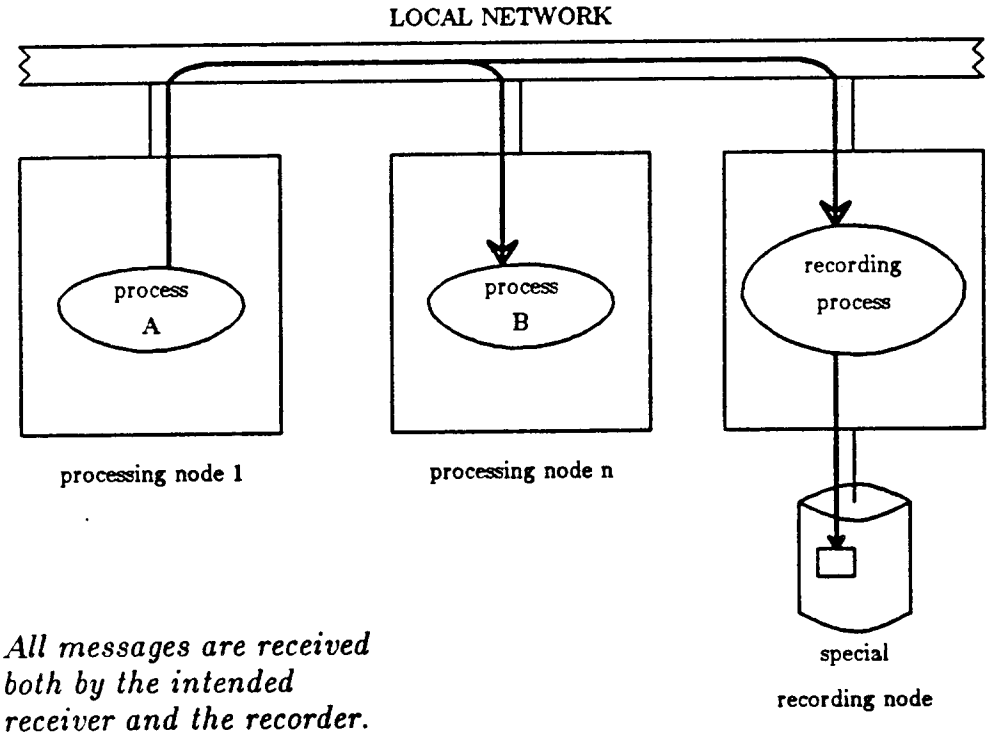



**Figure 3.2: Publishing System Before Failure**

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 main functions of a published communications system are:

- Storing messages and checkpoints
- Detecting crashes
- Recovering processes

- Recovering from recorder crashes

In the rest of this section, we examine how the system performs each of these.

### 3.3.1. Storing Messages and Checkpoints

The first checkpoint for a process is the binary image from which the process is created. When a new process is created, the recorder is told the initial state of the process, usually the name of this binary image and any other parameters associated with the process creation. If a crash occurs before the process is checkpointed, this binary file is used to restart the process.

Messages seen by the recorder are stored in the order in which they would be received by the destination process. These messages constitute a 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. A suboptimum choice of checkpointing frequency will yield less than optimum performance, but it will not affect the recoverability of a process or the system.

### 3.3.2. 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 treated as the crash of all processes on the processor.

Single process crashes are caused by sporadic processor errors. Such errors cause traps to the operating system kernel, which stops the process and 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.

Faults occurring within the kernel are handled as crashes of the whole processor.

### 3.3.3. Recovering processes

The system in recovery mode looks as in Figure 3.3.

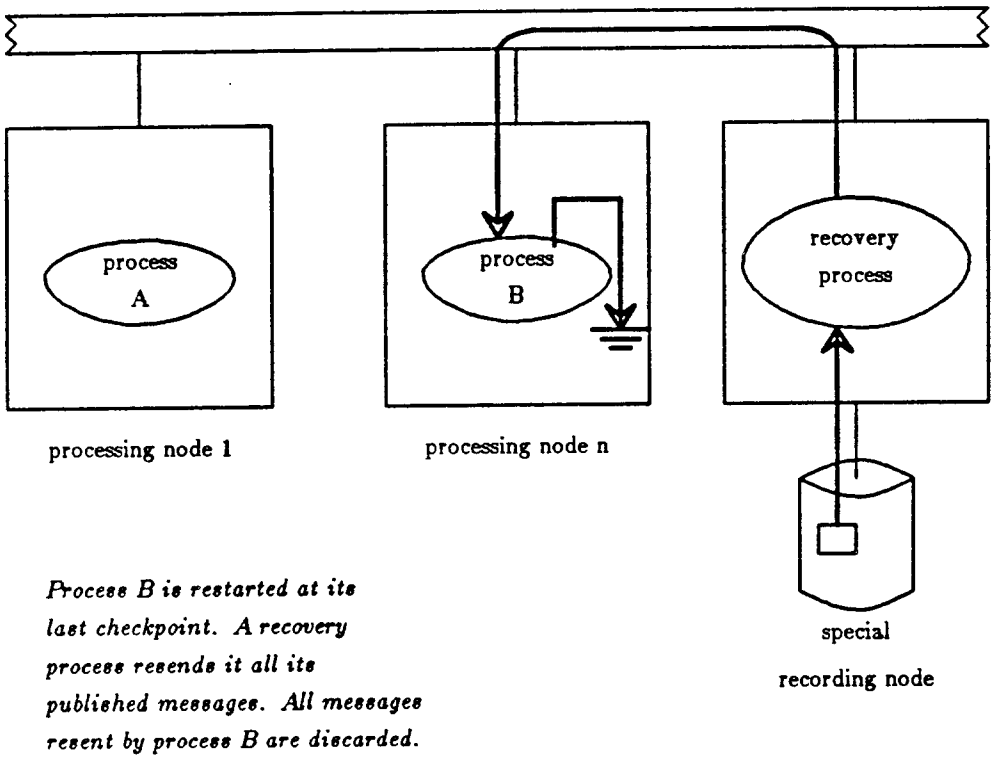



**Figure 3.3: Recovering a Process**

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:

(1) Picks 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) Sends 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) Sends 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. Messages arriving from processes other than the recovery process are discarded until the last recovery message is sent.

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.

### 3.3.4. Recorder recovery

In order to guarantee correct functioning of the system, all message traffic to processes must be suspended whenever the recorder goes down. Since this is a major disruption to any system, this should be a much lower probability event than other parts of the system failing. To insure this, techniques such as triple modular redundancy (TMR) for the recorder's components and battery backup for its power supply should be employed. TMR is a technique, originally proposed by Von Neuman[Von Neuman 56]. In TMR, each component in a system is triplicated. Outputs from the the three parts are passed through a voting circuit which selects the majority output. Thus any single component fault is automatically recovered. If no two outputs are the same, an error condition is flagged. Thus TMR increases both the reliability of a system and its error detection capability. The battery backup is necessary to protect against power fluctuations and to power any stable memory that is implemented using solid state memories. Such memories lose their contents if power is removed, but can be powered for hours using inexpensive batteries.

However, it is still possible for the recorder to fail. When this happens, three properties must be guaranteed for the correct functioning of the system:

1) no messages or checkpoints can be lost
2) any processes being recovered when the crash occurs must be recovered subsequent to the restart
3) any processes that crashed while the recorder was down will be recovered

The first property is provided by requiring a positive acknowledgement from the recorder for each message. The acknowledgement is given only after the message has been reliably stored. The message cannot be used by the receiver until this acknowledgement is received. In Chapter 6, we explain how this can be done without serious performance degradation of the system in many local area networks.

The second and third properties are provided by the recorder's restart actions. When the recorder restarts, it first reads the checkpoint and message information on its stable storage to determine which processes should exist. It then sends queries to all processors requesting the state of these processes. Upon receiving responses from these processors it then takes the following actions depending on the reported state of the process:

- **the process is functioning** - Nothing has happened so no action is taken.
- **the process has crashed** - This would occur if the process had crashed immediately prior to or while the recorder was down. Recovery is started for the process.
- **the process is being recovered** - The recorder had started to recover the process before crashing. The process is destroyed and recovery is restarted.
- **the process is unknown** - This might occur if the processor the process was on crashed while the recorder was down. Recovery is started for the process.

If any processor doesn't answer, the fact will eventually be detected by the crash detection system and recovery of processes on that processor will proceed in the usual way.

### 3.4. Recursive crash of the recorder

A recursive crash is a crash that occurs while the recovery is in progress. The only difference between a recursive crash of the recorder and its original crash concerns the process state requests sent by the recorder during restart. After the recursive restart, responses for old state requests may be received.

In order to ignore these out of date responses, we need to uniquely identify the responses belonging to any particular restart. This can be done by providing, in stable storage, a counter that is incremented each time a restart begins. All state request and response messages must contain that restart number. All state responses containing different numbers are ignored.

### 3.5. Recursive crash of a process

The recorder's kernel contains a table of all recovering processes and the recovery processes assigned to them. Whenever a recovering process crashes, the recorder's kernel is notified of it by the crash detection system. It then terminates the recovery process, terminates the recovering process, and creates a new recovery process to reinitiate recovery. It is assumed that when the process is terminated, all messages queued for it are also discarded.

### 3.6. Limitations

Any recovery system can only recover from detected errors. Therefore, the effectiveness of published communications depends on the fault detection capabilities of the underlying system.

With a single recorder, network partitioning can not be handled. If the network splits, the part with the recorder will attempt to restart on its part of the network all processes that were running on the now inaccessible part of the network. Should the network once again join, chaos would result.

With multiple recorders, a network partition may be recovered from. To do this, we must ensure that a recorder exists on each part of the network that is likely to become a partition. For example, a network made up of a number of Ethernets connected by transceivers must have a recorder on each of the separate Ethernets. It is the responsibility of each recorder to record messages for, and recover from crashes of, only processes on its part of the network. Should the network partition, no duplicate processes will be created. Processes communicating with processes on other partitions will just wait until the network is once again joined. Other processes will continue as if nothing happened.

### 3.7. Related systems

Publishing provides a system with reliable message delivery: it guarantees that all messages will eventually be delivered despite crashes of either sender or receiver. A number of other 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.

To build such a recorder, we assume the ability to listen to all messages on a broadcast network. Much precedent exists for this technique. 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.

The reader should also note the similarity between published communications and the system proposed by Russell, described in Chapter 2. Published communications can be seen as an extension of his ideas under the assumptions of deterministic processes and centralized communications media.

# CHAPTER 4

## An Implementation

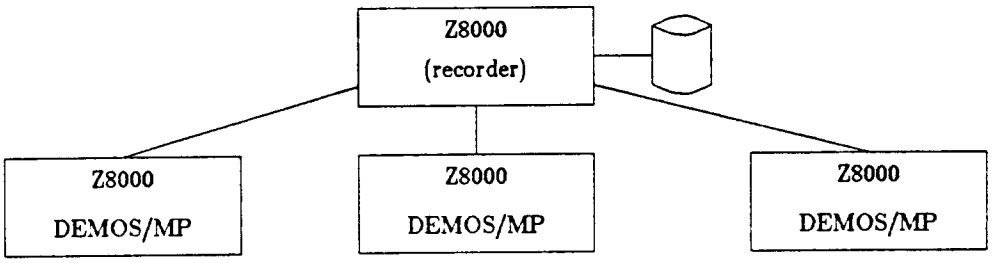
An initial implementation of published communications has been added to DEMOS/MP, a multiprocessor version of the DEMOS system originally created for the CRAY-1[Baskett et al 77,Powell 77]. This version was built to demonstrate how published communications can be easily added to an existing message-based system. The implementation includes the publishing of messages and the recovery of processes from their initial state and the published messages. Because of time and resource constraints we have not yet implemented process checkpointing and recorder node recovery, nor have we attempted any kind of TMR to make our recorder more reliable.

### 4.1. Experimental Environment

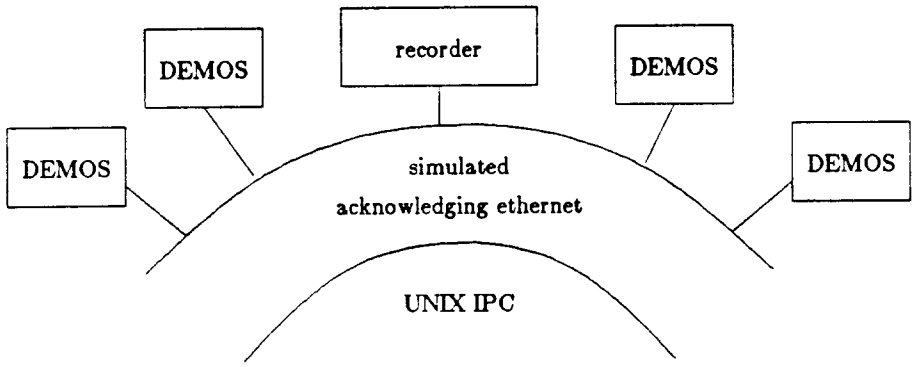
DEMOS/MP runs on a number of loosely connected Z8000-based nodes, connected via point to point low speed links (approximately 50,000 bits per second). The Z8000 is a sixteen-bit microprocessor made by Zilog[Zilog 80]. DEMOS/MP also runs under VAX UNIX[Ritchie and Thompson 78], where we have created a simulated multiprocessor environment. Generally, all software except low level device drivers is developed and debugged on the VAX system. The software can then be moved without change to the Z8000 systems.

For publishing, we have converted one of the nodes into a recorder for messages. This recorder must be able to reliably receive any messages seen by other nodes in the network. 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 ( Figure 4.1a). Any messages received incorrectly by the recorder are not passed on. In the version running under VAX UNIX, an Acknowledging Ethernet[Tokoro and Tamaru 77] is simulated using a low level protocol on top of the datagram sockets provided by Berkeley's 4.2 UNIX implementation ( Figure 4.1b). The Acknowledging Ethernet is described in Chapter 6.

We start this chapter by presenting the basics of the DEMOS operating system and the functions added by DEMOS/MP. We then show how publishing is added to this system.




**(a) the Z8000 version**




**(b) the VAX UNIX version**

**Figure 4.1: Experimental Configurations**

## 4.2. DEMOS

Our system stems from the original DEMOS system developed for the CRAY-1. DEMOS/MP preserves the organization, interprocess communications, and process structure of DEMOS. Therefore, we begin with a description of these aspects of DEMOS.

### 4.2.1. Organization

DEMOS is made up of cooperating processes and a message kernel ( Figure 4.2). The *message kernel* provides all communications between processes. It executes as privileged code and resides in the kernel address space. User level processes access the message kernel via kernel calls.

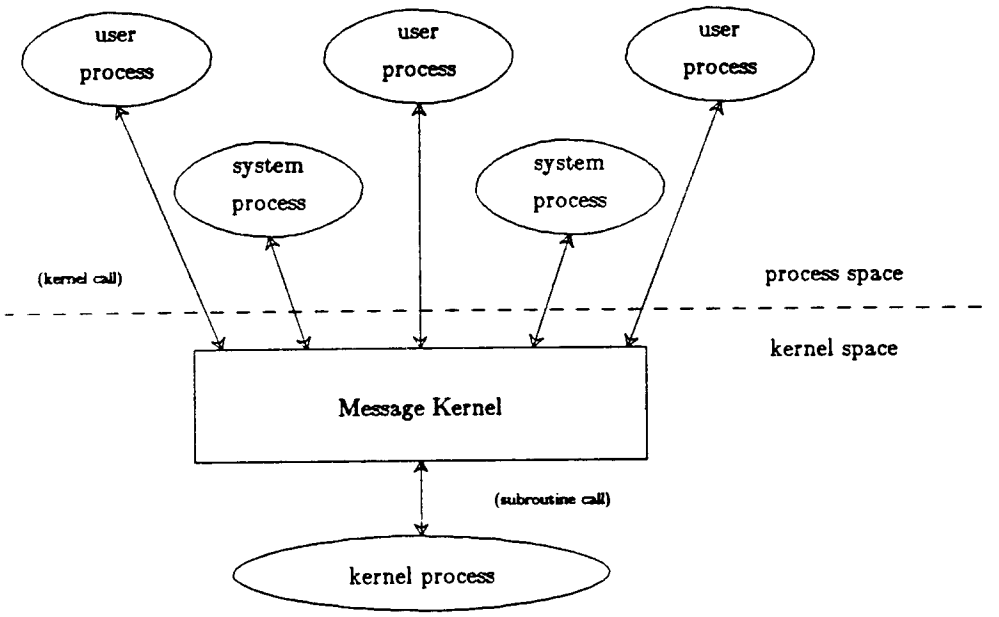



**Figure 4.2: DEMOS Kernel Organization**

The *kernel process* also resides in the kernel space. It provides a standard interface for the network, disk control, real time clock, and process control. Thus, it provides an abstraction that masks the particular personality of the underlying hardware. User level processes make requests of the kernel process by sending it messages. The kernel process consists of an infinite loop that alternately processes these request messages, polls and handles interrupts, and schedules non-kernel processes to run.

System processes are user level processes that are an integral part of the operating system. While the kernel provides primitive functionality, the system processes provide structure and policy. The file system, job control system, interval timer, and command interpreter are all system processes. System processes are distinguished from application (user supplied) processes only by the way they are created. The system processes are created by the kernel process when the operating system starts up.

### 4.2.2. DEMOS interprocess communications

DEMOS interprocess communications is based on messages. Three objects are important to communications: links, channels, and messages.

#### 4.2.2.1. Links

In any message system, processes have to be able to name each other in order to send and receive messages. In DEMOS, this "name space" is

implemented using a special protected object called a *link*. A link is much like a capability[Fabry 74]. It allows access and is immutable and unforgable. A DEMOS process must have a link to another process in order to send it messages.

Links exist outside of the address space of the processes, either in messages or in kernel resident link tables. A link can only be accessed in certain kernel calls, such as link creation, link destruction, and message send. The process always refers to a link via a *link id*, which is the link's index into the link table. Thus, the message kernel controls all access to links.

For a process to receive messages, it must create a link to itself. It can then pass that link to another process in a message.

It is possible for a process to determine over which link a message to it has been sent. It does this by assigning a number, called a code, to a link when the link is created. Any message sent using that link will contain the code in its header. Whenever a process performs a kernel call to receive a message, the kernel returns not only the message contents, but also the code from the message header.

The code allows links to be used as pointers to resources. For example, when the file system opens a file it returns to the client process a link whose code identifies the file. The client then requests reads and writes by sending messages over the link. Using the code returned with the request messages, the file system can tell which file is being read or written.

When a process is created, the creating process may insert a number of initial links into the new process's link table. This solves the rendezvous problem for processes. When the kernel starts the system processes, it starts them all with one link, a link to a named-link server. A system process can then send messages to the server containing a link and a name for the link. Another system process can then obtain the named-link from the server by sending the server a request containing the name of the link desired.

#### 4.2.2.2. Channels

The DEMOS message kernel maintains a queue of input messages for each process. Whenever a message is sent to a process, it is appended to the end of the queue. Normally, a process will read the messages in its queue in the same order in which they arrive. However, some messages may be more urgent than others. A process may wish to read those messages before others which are ahead of them in the queue. In DEMOS, this is done using channels.

When a process creates a link, it specifies the channel which the link belongs to. Any message sent over the link will contain the link's channel number in its header. Whenever a process performs a receive kernel call, it

specifies the channels from which it is willing to receive a message. Instead of returning the next message in the queue, the message kernel returns the next message in the queue which belongs to one of those channels. Thus, the process can receive messages selectively.

#### 4.2.2.3. Messages

Messages consist of three parts: a header, a passed link, and a body. The header contains the code and channel of the message in addition to information needed to route the message to the correct process. These fields are obtained from the link over which the message is sent.

A message can include one link by specifying its link id when sending the message. The link is removed from the sender's link table and copied into the message. When the message is read the link is moved into the receiver's link table. The receiver is told the link id of the link.

The body of the message is not interpreted by the kernel. It has a maximum size specified by the implementation. It is left to the communicating processes to agree as to the contents and format of a message.

### 4.2.3. Process control

The process control system of DEMOS consists of three processes: the kernel process, the memory scheduler, and the process manager. The reason for this three way split is modularity. Each process provides a separate address space in which to perform a particular function.

The three processes are connected serially. The process manager has a link to the memory scheduler and the memory scheduler has a link to the kernel process. All user level process control requests are made to the process manager. The request is then passed through the three processes, each performing its particular function. Eventually a reply is passed back up to the requester.

The process manager maintains all information about process groups, called jobs. Each time a user logs in, he starts a new job. All processes created by his login session are part of the same job. A job has associated with it certain limits to control the amount of resources used by a user.

The memory scheduler handles problems associated with swapped processes. Many kernel operations in process control abort if the process is swapped out. The memory manager makes sure the operations are successfully completed once the process is returned to memory.

The kernel process is the lowest level of the process control system. Its provides the primitives needed by the other two levels to create, change, and

destroy the kernel resident state of a process.

## 4.3. Distributing DEMOS

The original DEMOS supports the distribution of computation across processes via messages. The operating system is itself a prime example. Extending this to include distribution of processes across processors turned out to be a simple matter. The additions needed were:

1) network wide process names
2) remote process creation
3) remote message routing

In DEMOS/MP all of these have been added in a manner that is transparent to the processes. Each processor has its own message kernel and kernel process. A kernel process on a processor controls only the processes on that processor. Processor assignment for system and applications processes can be arbitrary, from a functional standpoint, since the placement is transparent to the processes. However, the performance of the system may be sensitive to where processes are placed. In the current implementation, placement of system processes is up to the system administrator and placement of applications processes is dependent on an easily changed policy algorithm in the job controller.

The rest of this section describes how the three distribution problems, listed above, are solved.

### 4.3.1. Network wide process names

Associated with each process, in single processor DEMOS, is a unique identifier. In DEMOS/MP, this identifier is made unique, network wide, by appending to the single processor ID the unique ID of the processor on which it was created. Processes maintain this identifier, even if they should migrate[Powell and Miller 83].

Due to the nature of links, this change in process identifier is completely transparent to processes. As explained above, a process names another process only via links (There are two exceptions to this rule, the job controller and memory scheduler. These exceptions will be addressed in detail later in the thesis). Since the process identifiers exist within the links, the processes do not have to be changed to support the new name space.

### 4.3.2. Remote process creation

Process creation remains much the same as in DEMOS. The difference lies in the memory scheduler. Instead of a link to just one kernel process, the

memory scheduler maintains a link to the kernel process of each node, allowing it to create processes on all nodes.

When a user level process requests a process creation it may supply an optional parameter specifying on which machine to create the process. If the parameter is not present, the memory scheduler chooses the node from which the request came.

### 4.3.3. Remote message routing

A network interface was added to the message kernel in DEMOS/MP. If neither sender nor receiver crashes and network failures are temporary, the network guarantees that:

- messages are not duplicated
- all messages sent arrive at the receiver's processor
- all messages from one process to another arrive in the same order in which they were sent

The rest of this section describes the details of the network that ensures these properties.

The layering of the system is same the same as many contemporary networks such as the XEROX's internet protocols[XEROX 81] and the DARPA TCP/IP protocol[DARPA 82a,DARPA 82b]. Figure 4.3 shows the organization of the network interface. The organization is strictly hierarchical with one exception, the interrupt servicing. The kernel process polls for interrupts from all devices. Rather than filter interrupts through a number of layers, the kernel directly interfaces to the lowest layer in addition to the highest.

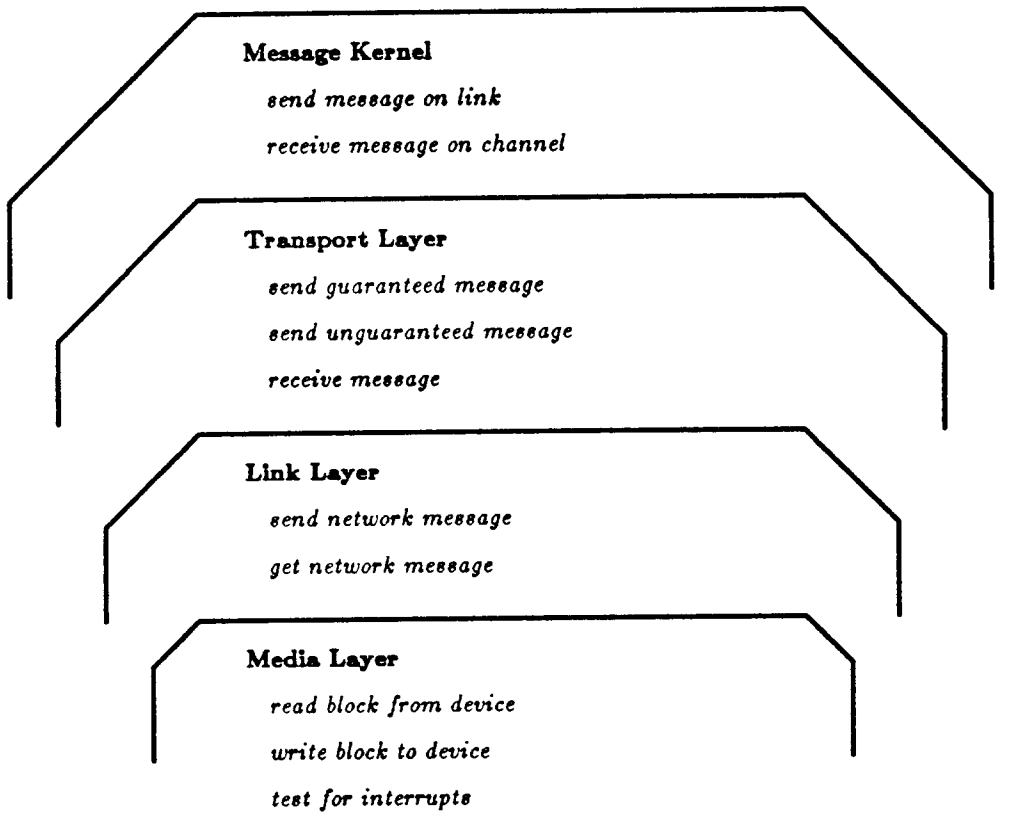



**Figure 4.3: Network Interface Organization**

The lowest layer in the network is the *media* layer. The media layer creates an abstract network device for the rest of the system. The abstract device can be read from, written to, or have its interrupt status polled. To do this, the media layer often has to provide interrupt time services such as buffering of input messages. The media layer is the only layer that differs in the VAX and Z8000 implementations.

The next layer is the *link* layer. It is responsible for assuring that only error-free messages are transmitted to upper layers. It does this by wrapping all messages with a rotating checksum and by checking the message type for validity. Any messages with an incorrect checksum are discarded.

The most complex part of the network is the *transport* layer. The transport layer provides:

- unguaranteed, high priority messages
- guaranteed messages
- duplicate message suppression

- route through
- network time

The transport layer provides two types of messages, unguaranteed and guaranteed. The idea of supporting both guaranteed and unguaranteed messages was pioneered at XEROX PARC in their internet protocols. Other networks, such as that provided by Berkeley's 4.2 UNIX implementation[UCB 82], have since adopted the idea.

Unguaranteed messages exist for the kernel process when sending dated or statistical information such as routing information. Such messages have no need for guarantees since they are sent periodically and since they would often be out of date if retransmission were necessary.

Guaranteed messages can also be sent. They are guaranteed using an end-to-end acknowledgement protocol. By end-to-end, we mean the processor from which the message originates expects an acknowledgement from the processor on which the destination process resides. Until this acknowledgement is received, the originating processor periodically resends the message. End-to-end acknowledgement is popular in networks, such as most local area networks, where messages do not have to be forwarded through many processors before reaching a destination.

Message resends are a potential source of duplicate messages. To avoid them, each message is given a unique identifier (The identifier is made up of two fields: the unique identifier of the sending process and a number from that process's state block. This number is increased every time a message is sent by that process). Each processor keeps a cache of identifiers of recently received messages. If the identifier of a received message is found in this cache, then the message is discarded as a duplicate. The size of the cache is adjusted to make the lifetime of a message in the cache many times greater than the time for a message to follow the longest path through the network.

Message ordering between processors is currently preserved by allowing only one unacknowledged message to be in transit from each processor. The processor will send no other messages until the current one is acknowledged. (This scheme is inefficient when message traffic is high. It will be replaced in the future by a windowing scheme that will continue to preserve message ordering.) The message kernel itself guarantees that messages for a process within a processor will not "pass" each other. Therefore, message ordering is guaranteed across the network.

## 4.4. Making DEMOS/MP compatible with published communications

In Chapter three, we showed how processes could be recovered in a message based system. In our model, a recorder attached to the network records all messages to processes. A process can be recovered by restarting it at a previous checkpoint and replaying messages to it. The validity of our model hinges upon three properties of the system:

1) all messages must be published before being queued to the processes for which they are intended
2) processes receive messages in the same order in which they are seen by the recorder
3) processes interact only via messages

Although DEMOS/MP is a message based system, the system had to be examined and changed to guarantee the above properties. This section describes those changes.

### 4.4.1. Publishing messages before they are used

A recorder was added to the system by converting one of our DEMOS/MP nodes into a recorder. To make that node see all messages, we have modified the message kernel in DEMOS/MP to send all messages, including intranode messages, on the network before routing them to the intended process. By sending all messages on the network, we can guarantee to record all messages.

The recorder has the ability to receive all transmissions on the network. If it incorrectly receives a message or message acknowledgement, the recorder can block the transmission, ensuring that no other processor correctly receives it.

As we explained earlier, the network has three kinds of messages: unguaranteed messages, guaranteed messages, and acknowledgements for guaranteed messages. If a guaranteed message is blocked, it will eventually be resent by the transport layer. The blocking and resending continues until the recorder successfully records the message. The interference causes no malfunction in the case of the other message types. If an unguaranteed message is blocked, there is no problem. It after all was not guaranteed. If an acknowledgement is blocked, the message it acknowledges will eventually be resent. Once the duplicate message is received, it will be acknowledged again. Duplicate message suppression keeps the second copy of the message from being passed on to the process.

It is possible to discover the order in which messages are received at the receiving node by tracing the acknowledgements sent in response to messages.

### 4.4.2. Message ordering at the recorder matches that at the process

The published communications system described in Chapter 3 assumed that messages overheard over the network would be in the same order as they are received by the intended process. Thus, the recorder could tell, just by listening to messages and acknowledgements, in what order the process receives messages.

With channels, this is no longer necessarily the case. Channels allow messages to be received in an order different than that in which they are placed in the process's queue. The only way for the recorder to know in what order messages are received is to be actively informed of the order of reception. In DEMOS/MP, we do this by sending a message to the recorder whenever the use of channels causes messages to be read out of order. The message contains the id of the message read and the id of the first message in the queue, that is, the message that would have been read had channels not existed. Using this information, the recorder can determine the order in which messages have been read.

When a process is recovering, it must read messages in the same order as before the crash. The recorder replays messages to the recovering process in the order in which they were originally received. Whenever a recovering process performs a receive message, the channels specified must agree with the next message being replayed. If they do not, the receive message call returns with a code specifying no messages in the queue.

### 4.4.3. Processes interact only via messages

Our next problem was to ensure that all process interactions were only via messages. To this end, we had to determine what makes up the complete state of a process, and exactly what actions can be initiated by a process. We then examined each of these actions to see if they could affect the process state of another process.

A process's state consists of:

- *process address space* - This is the part of the process containing the program, its data, and its stack.
- *process control record* - This contains the run state of the process, various scheduling parameters, and the head of the queue of waiting messages. This information is resident in the kernel address space.
- *process save area* - This is the area in which variable length tables for the process are kept, such as paging information, the link table, and context switch information. This information is also resident in the kernel

address space.

The actions available to a process are:

1) execution of instructions belonging to the process's program.
2) calls to the message kernel
3) request messages to the kernel process for process control

Program execution was the easiest to examine and rule out as a possible problem. Execution of program instructions can change only the address space of that process. This is a direct result of the memory mapping hardware.

DEMOS provides 16 kernel calls for processes. When a process makes a kernel call, it traps to the message kernel. Since the message kernel has the ability to change any process's kernel resident state, this was a potential problem area. On examining the kernel calls we found that 5 operate upon links owned by the calling process, 10 cause message reception or transmission, and 1 causes the calling process to stop. All these calls, by virtue of returning a condition code to the calling process, change the caller's address space. However, never do these calls affect the state of a process other than the calling one except by adding messages to the other process's message queue.

Finally, we looked at the possible requests to the kernel process. As we have shown earlier, the kernel process is a special process resident in the kernel's address space. It is the part of the system that creates new processes, and can therefore, change the parts of the process state that it creates: the process control record and the process save area. This, in itself, would not be a problem since we have already conceded the need to inform the recorder of process creation. However, requests exist to allow the kernel process to change a process's run state, and to move links from one process's state to another's. Although the request is made to the kernel task via a message, the effect of the request is a shared memory interaction between the kernel task and the controlled process. This interaction is invisible to the recovery system and cannot be correctly recovered.

The simplest way to demonstrate the problem is to give an example of a kernel process request and show how it will lead to complications during recovery. Figure 4.4 shows the actions normally taken in a MOVELINK request. A MOVELINK request to the kernel process causes it to move a link from one process's state to another's. This is normally done immediately following process creation to allow a new task to talk to other tasks. Process A starts by creating the link. It then sends a message to the process control system to move the link to process B. The request eventually filters down to the kernel process which actually moves the link from one link table to another.

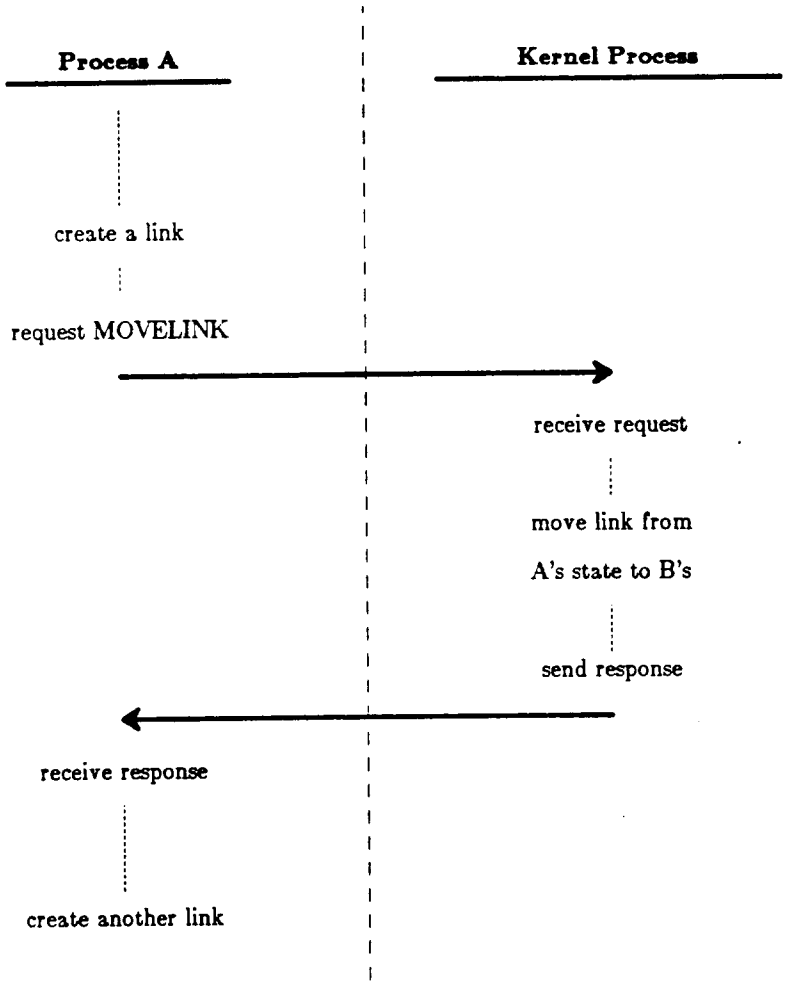



**Figure 4.4: Actions taken by MOVELINK**

Now consider what happens when the system crashes. Processes A and B and the kernel process are restarted and are resent their published messages. If process A recovers to a point after the create link call before the kernel process is replayed the MOVELINK request, the request will function correctly. However, if the kernel process recovers faster than process A, it may process the MOVELINK request before the link has ever been created.

This problem is circumvented by making process control yet another message stream to the process. The problem can then be handled in the same way as the channel problem. Only creation requests are sent directly to the kernel process. All other requests (move link, migrate process, stop process) are sent to the process itself. These are intercepted by the kernel process. The kernel process then temporarily assumes the identity of the controlled process while it performs the control functions.

This is done using a kind of link called a DELIVERTOKERNEL link. After creating a new process the kernel returns to the requester a DELIVERTOKERNEL link that points to the created process. All subsequent process control requests for that process must come over that link. When the message kernel receives a message sent via a DELIVERTOKERNEL link, it passes the message, not to the process to which it is addressed, but to the kernel process residing on its node.

While performing process control operations, the kernel may start up conversations with other processes. Whenever it does, any messages it sends are attributed to the controlled process. Also, any reply links, which it passes to other processes, are DELIVERTOKERENEL links pointing to the controlled process.

When recovering a process, process control messages are replayed just like all other messages. Their ordering is preserved with respect to all other messages to a process.

Figure 4.5 shows the actions involved in the MOVELINK call after the above changes. Process A starts the exchange by sending a DELIVERTOKERNEL request to process B. The kernel process, running as process B, then sends a DELIVERTOKERNEL request back to process A requesting the link. The kernel process, running as process A, then sends a DELIVERTOKERNEL message back to process B containing the link to move. Finally the kernel process, now running as process B, stores the link in process B's link table.

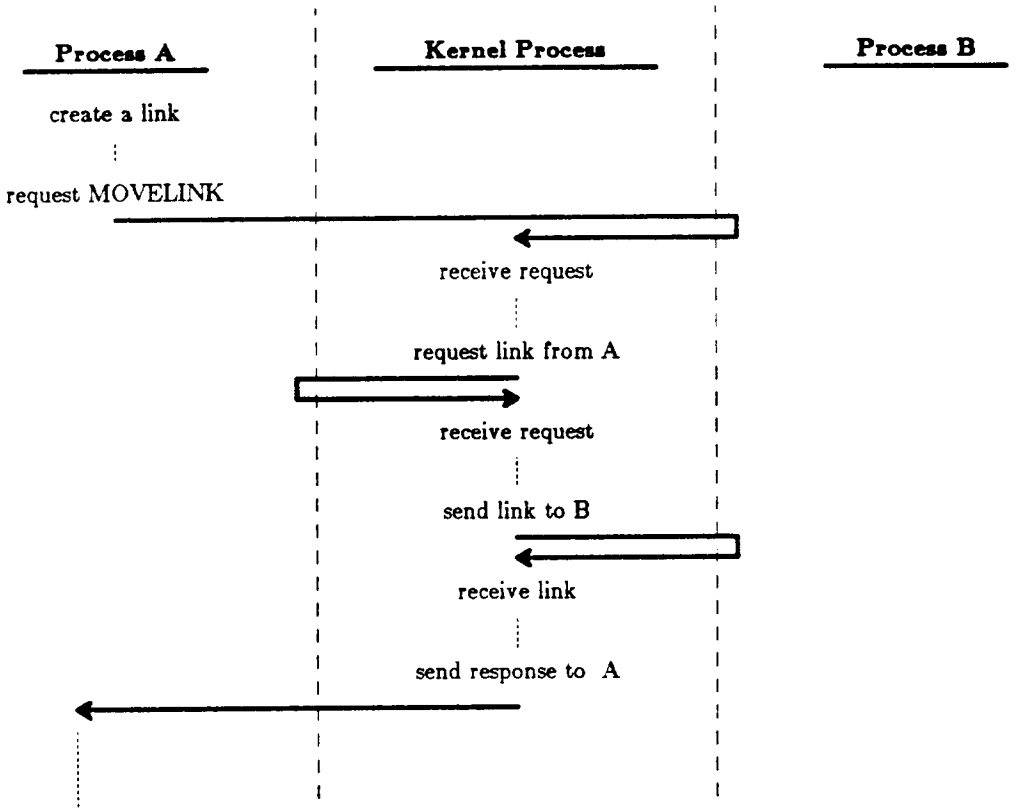



**Figure 4.5: Actions taken by the new MOVELINK**

## 4.5. Publishing messages

Our system consists of a number of DEMOS/MP processors, one of which is a recorder. Each runs a modified version of the message kernel. The message kernel for processing nodes has been modified so that all messages, even intranode, are broadcast on the network. A message must be received from the network before it can be delivered to a process. The message kernel on the recorder has been modified to pass all guaranteed messages and acknowledgements to the recording software. The messages are recorded for play back during recovery. The acknowledgements are recorded so that the recorder can determine the order messages are received in at the processing nodes. As we explained earlier, all messages not received by the recorder are aborted by the recorder so that no other node can receive them.

The recorder stores all messages on disk after a small amount of processing. The processing is performed to maintain a data base of running processes. Each entry in the data base contains the following information:

- the process identifier
- the identifier of the most recent message sent by the process
- a list of ids of messages received by the process (since the last checkpoint)
- the file name of the last checkpoint for the process
- the id of the first valid message
- a list of disk pages containing messages to the process
- whether or not the process is recovering

Information about process creation and destruction is provided by the kernel processes on the processing nodes. They have been modified to send a message whenever a process is created or destroyed. The recording software recognizes these messages and uses them to maintain its data base.

As messages are received they are timestamped and buffered. We then append to the process's data base entry the disk address of the buffer. When the buffer is full it is written to disk.

Before allocating a buffer to a disk page, the disk page is read in. Any messages that are no longer valid are removed and the buffer is compacted. It is then available for buffering new messages.

The process data base is just a summary of the information that appears on disk. If the recorder crashes, it is possible to rebuild the data base from the disk.

### 4.6. Failure detection

Our current version of DEMOS/MP, because of lack of facilities in the hardware, cannot detect single process failures. Instead, we simulate them in our VAX UNIX version. Processor crashes are detected using a timeout protocol. When the recording node starts up, its kernel process creates, on the recording node, a watch process for each processor in the system. Each watch process is given a link to the kernel process it is supposed to watch. The watch process periodically sends an "are you alive" request over this link. The kernel process on the processing nodes has been modified to reply to this request. If no reply is received in a predetermined interval, the processor being watched is assumed to have crashed. The watch process then outputs a query to the operator asking him what response should be taken. There are currently three:

- do not recover
- recover on the same processor

- recover on a spare processor that can assume the process's processor's identity

We have not yet integrated the ability to recover on a different processor.

Having chosen a course of action, the watch process sends a query to the kernel process requesting all of its data base entries for processes on that processor. It then starts up a recovery process for each one, including the kernel process.

### 4.7. Recovering processes

Individual processes are recovered by recovery processes. A number of changes have been made to the system to allow the recovery processes to work. The first is a new request recognized by the kernel process of a processing node, a request to recreate a process. If the process already exists, it is destroyed. The recreate request can specify a file to load into the process or it can indicate that the file is to be loaded from a previous checkpoint that will follow in subsequent messages. The recreate request also contains two message ids. The first is the message id to be given to the first message sent by the recovering process. The second is the id of the last message sent by the process before it crashed. So that a process will not resend old messages, the message kernel has been modified to not send any messages with id's less than this id.

The second major change is to the message kernel of the recording node. A new system call has been added to allow the recovery process to inject messages into the system without the use of links. When the recovery process is started, it is given a link to the kernel process of the processor on which recovery is being performed. It also starts with two links to its own kernel process. The first link is for messages concerning recovery. The second is to allow it to read the publishing disk. The recovery process starts by issuing a recreate request to the other processor. When it receives confirmation, it reads all the published messages and resends them to the process using the special call. It is up to the message kernel on the other processor to make sure the messages are handed to the recovering process in order and that all messages the recovering processor resends are not passed on. After the recovery process has sent the last published message, it sends a message to its the recorder's kernel telling it that the process is now recovered. The recovery process then terminates.

# CHAPTER 5

## Performance Studies

This chapter presents two performance studies of published communications. The first involves a simulation of the steady state system. Resource utilization of different parts of the system are studied under varying loads. The second study measures the DEMOS/MP implementation. The costs of publishing on the processing nodes and the recorder are determined.

### 5.1. A Queuing Model Simulation

In order to get an estimate 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 3.2. 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.

Three types of messages originate at the processing nodes: short messages (128 bytes long), long messages (1024 bytes), and checkpointing messages (1024 bytes). 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.

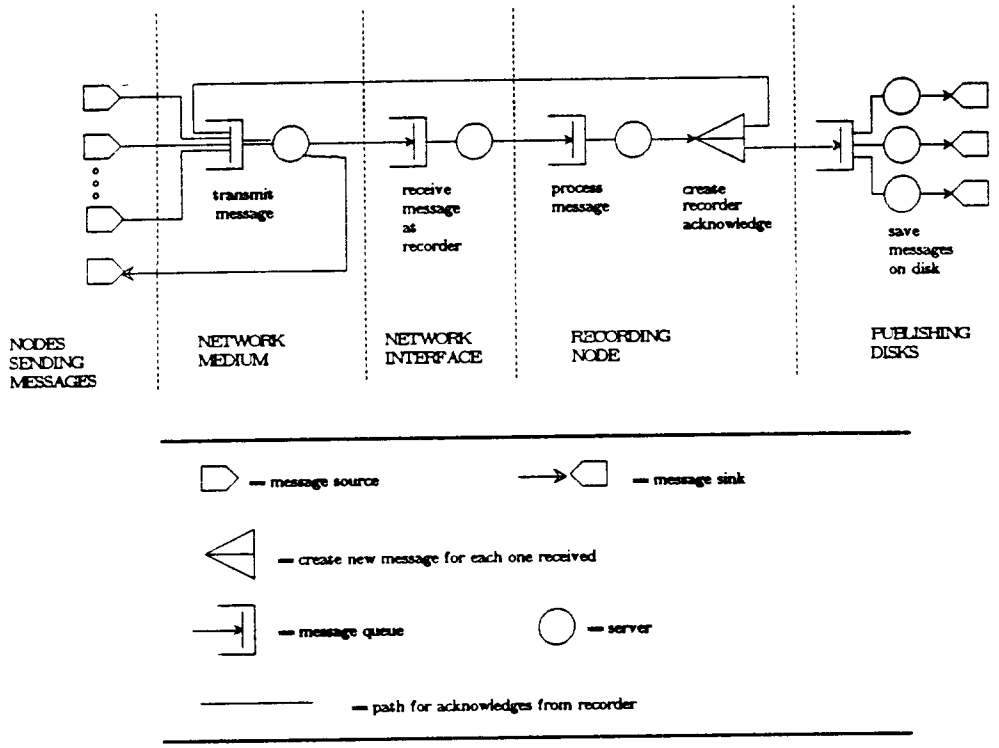



**Figure 5.1: The Open Queuing Model**

Figure 5.2 shows the values of hardware parameters chosen from our computing environment at Berkeley, which consists of DEC VAX 11/780's connected via a 10 megabit Ethernet.

| parameter | value |
|---|---|
| Ethernet interface interpacket delay | 1.6 ms |
| Network bandwidth | 10 megabits per second |
| Disk latency | 3 ms |
| Disk transfer rate | 2 megabytes per second |
| Time to process a packet | 0.8 ms |

**Figure 5.2: Hardware Parameters for the Queuing Model**

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 5.3 shows the distribution of state sizes.

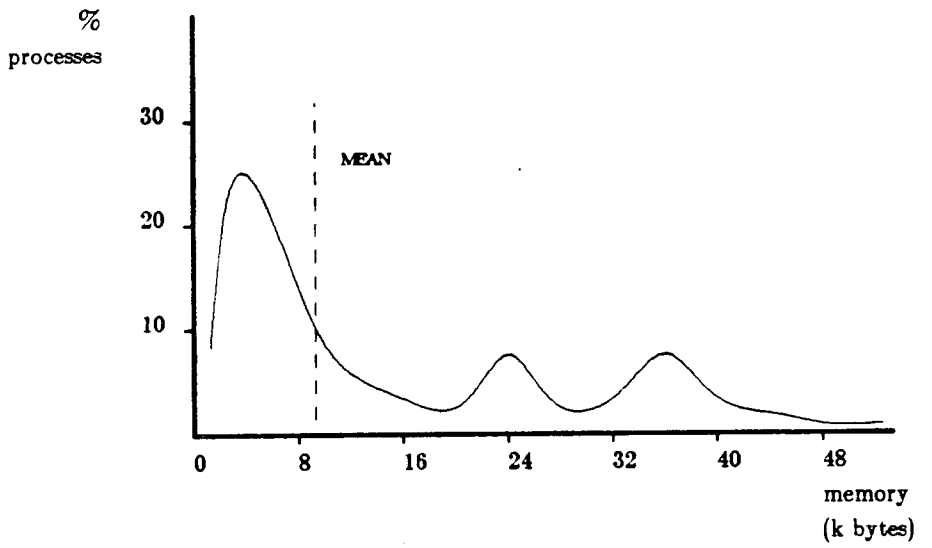



**Figure 5.3: State Sizes for UNIX Processes**

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. This conversion is consistent with what we would expect to see if we were running DEMOS instead of UNIX.

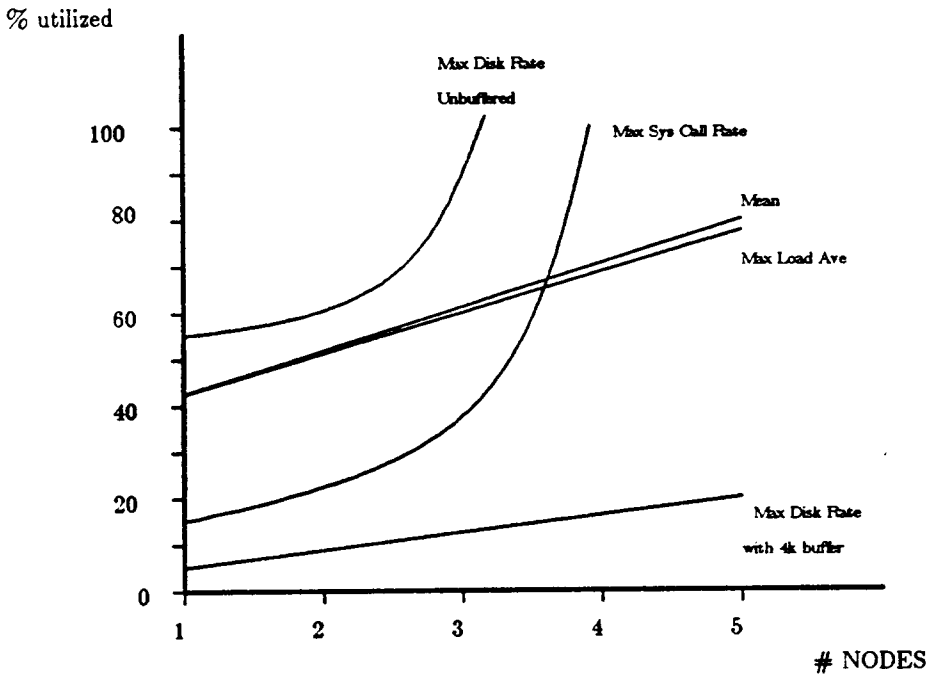
Using these measurements, four operating points were established, one representing the mean of each parameter and the other three representing the measurements when each of the parameters was maximized. Figure 5.4 shows the parameter values for those operating points.

| description of operating point | 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 |

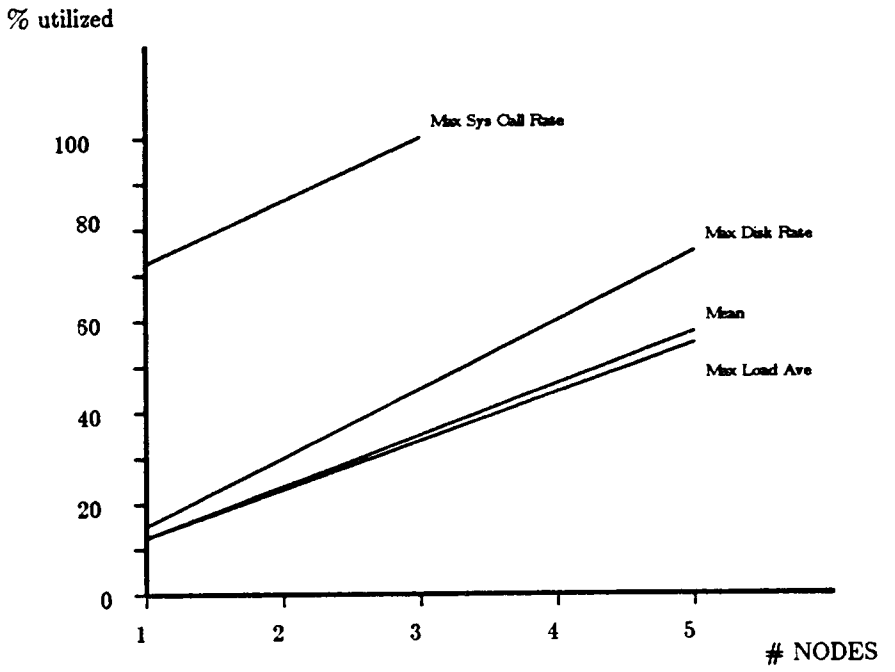
**Figure 5.4: Operating Points for the Queuing Model**

The system was simulated for 1 to 5 processing nodes and 1 to 3 disks at the publishing node. Figure 5.5 shows plots of the utilization of the publishing node processor, its disk system and its network interface.

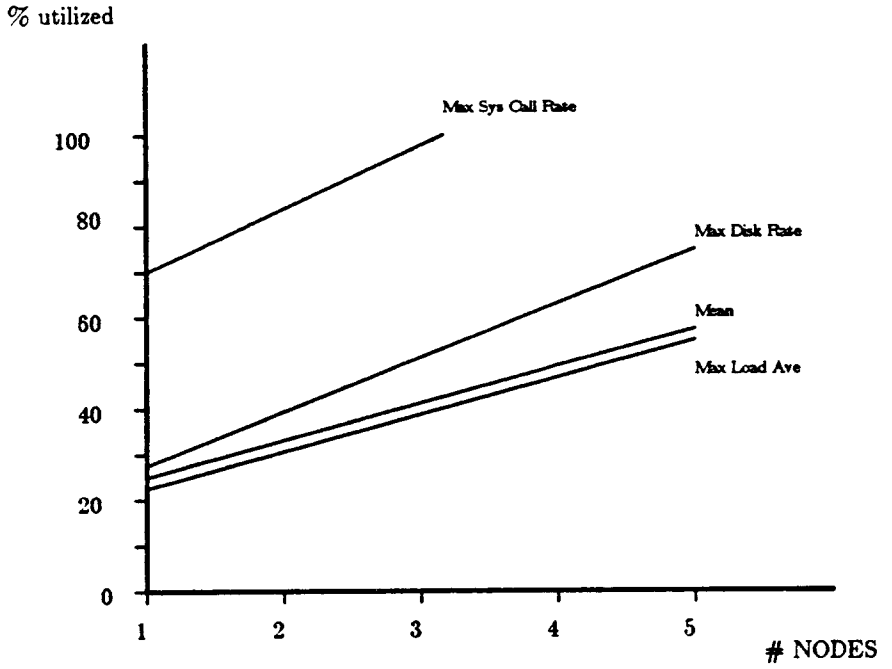
The system stayed within physical limits with two exceptions. The first was the saturation of the disk system used with the maximum long message rate. This saturation was removed by allowing messages to be written out in 4k byte buffers rather than forcing one disk write per message. The second problem 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.




**(a) Disk Utilization**




**(b) Recovery Node Utilization**




**(c) Network Interface Utilization**

**Figure 5.5: Percent Utilization of System Components**

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.

## 5.2. Measurements of the DEMOS/MP implementation

The implementation, described in Chapter 4, was measured. Since the DEMOS/MP system is a research vehicle, it does not support a community of users or any applications. Because of this, we have not measured the system under any realistic loads. Instead, we have measured some of the basic system dependent parameters from which load dependent ones can be inferred. Other researchers, desiring to use published communications, can then determine the costs they would incur for their particular systems.

The two basic performance questions for published communications are:

- What does it cost the processing nodes?
- How long does it take to publish a message?

The experiments outlined below determine the basic parameters that can be used to answer these questions under varying loads.

The measurements were obtained from the DEMOS/MP implementation running under the simulated multiprocessor environment on a VAX 11/750. All CPU times presented are CPU times for the VAX 11/750. Transmission times for messages are computed assuming a 10 megabit network.

### 5.2.1. Processing node costs

Of all the changes made to DEMOS/MP to support publishing, two actually add to the work done by a process node:

1) broadcasting intranode messages on the network

2) advising the recorder of process creation and destruction

The costs of both are determined by direct measurement.

We first determined the effect of sending intranode messages over the network medium. In particular, we measured the increase in CPU utilization and the increase in transmission time. The transmission time is the time elapsed from the send message call of the sending process to the receive message call of the destination process. To make the measurements, the program shown in Figure 5.6, was run on versions of DEMOS/MP with and without publishing. In each case, the system was otherwise quiescent, that is, it had no other processes running on it.

```
--- Get the value of the real time clock
startReal := Get_Real_Time;

--- Get the CPU time, since system start, spent outside the idle loop
startCpu := Get_Run_Time;

--- Send the message 512 times
for i in 1..512 do
   SendMessageToSelf;
   ReceiveMessage;
od;

--- Calculate time for each Send/Receive
realTime := (Get_Real_Time  - startReal) / 512;

--- Calculate total CPU time for one Send/Receive
cpuTime := (Get_Run_Time  - startCpu) / 512;
```

**Figure 5.6: A program to measure message costs**

In this program, Get_Real_Time returns the real time. Get_Run_Time returns the CPU time that the kernel spends outside of the idle loop.

Figure 5.7 shows the values of realTime and cpuTime for both versions of DEMOS/MP. (It should be noted that times depend on the network protocol. However, our protocol is close enough to commercial protocols (XNS, TCP/IP, DECNET) for the numbers to be meaningful to other systems.)

| Times with/without publishing for intranode messages | | |
|---|---|---|
| | realTime (ms) | cpuTime (ms) |
| with | 51 | 48 |
| without | 23 | 22 |

**Figure 5.7: Per Message Overheads**

In the version without publishing, the 1 ms difference between the CPU time used by the kernel and the elapsed real time is the time used by the user process. This difference is 3 ms in the version with publishing since an additional 2 ms are spent in transmitting the message over the network medium. Finally, the additional 26 ms of CPU time used by the version with publishing is due entirely to the network protocol and to the servicing of the network device interrupts. Of this time, less than 1 ms is attributable to copying the message into and out of device buffers.

We then ran a similar experiment to determine the increase in CPU cost during process creation and destruction. This increase is caused by two things: notification to the recorder of process recreation and destruction, and the publishing of messages sent between the three parts of the process control system when creating a process. A null process was created and destroyed 25 times on a system with publishing and one without. The total CPU time of the system was measured in each case to determine the average increase in CPU time in the publishing system due to notification of process creation and destruction. Figure 5.8 shows the results.

| CPU time with/without publishing for creation and destruction of a null process | |
|---|---|
| | CPU time (ms) |
| with | 5135 |
| without | 608 |

**Figure 5.8: Per Process Overheads**

Once again, the difference between CPU time used in the two versions is directly attributable to the servicing of network protocols.

These two experiments have shown us that most of the cost of publishing is caused by the use of the general message protocol for publishing intranode messages. Since intranode messages are transmitted on the network only for the recorder to see, it may be possible to streamline the protocol for these transmissions, thereby reducing the cost of publishing intranode messages.

### 5.2.2. Publishing time for messages

In the queuing simulation, we assumed that the recorder could receive a message and publish it in 0.8 ms. No attempt was made to meet this limit in our implementation. However, we measured our implementation to determine if, with tuning, the could be improved to meet this goal.

As in the previous experiments, we created a process to send a number of messages to itself. By measuring the total CPU time used by the kernel both before and after running this process, we determined the average CPU time taken to process a message. This time was 57 ms per message. After analyzing the code involved, we reduced this number to 12 ms by replacing subroutine calls by inline routines.

In this experiment, we used a modified DEMOS/MP kernel as the software for the recorder. As a result, all messages must go through all layers of the network protocol before being published. By intercepting and publishing the messages directly at the media layer of the protocol, we feel that the per message cost can be reduced to the desired 0.8 ms or lower.

# CHAPTER 6

## Extensions and Applications

We have presented a model of published communications and a simple implementation of it. We now show how the model can be extended and used.

### 6.1. Getting messages to the recorder

We have stated that the recorder must publish all messages on the network. If it cannot receive a message, the processor for which the message is destined cannot be allowed to receive it. One way to achieve this is to use a transport protocol that ensures it. For example, the processor receiving the message could be forced to wait for an acknowledge from the recorder before using the message. Solving the problem at the transport level may actually be acceptable for many systems. However, special purpose solutions, built into the network interface devices, can dramatically reduce performance degradation. We describe here solutions for the two most popular types of local area networks, CSMA/CD (Ethernet like) and token rings[Wolf and Liu 78,Pierce 72,Farmer & Newhall 69].

#### 6.1.1. Ethernets

A solution to the recorder acknowledge can be borrowed from a variant of standard Ethernet, the Acknowledging Ethernet, designed and studied by Tokoro and Tamaru[Tokoro and Tamaru 77]. In the standard Ethernet, the network is available to all nodes for transmission whenever they detect no transmission on it. If two nodes transmit at the same time (collide), they will detect the condition, cease transmission, and then retry after pseudo randomly different intervals.

The Acknowledging Ethernet works much the same way. The difference is that a time slot is reserved after each message is sent. During this time slot, only the receiver is allowed to transmit. If it has correctly received the message, it broadcasts an acknowledge; if not, it does nothing. When the network is not busy, as in Figure 6.1, both the standard and Acknowledging Ethernets behave in much the same way. Whenever a message is sent, the receiver will acknowledge immediately following reception.

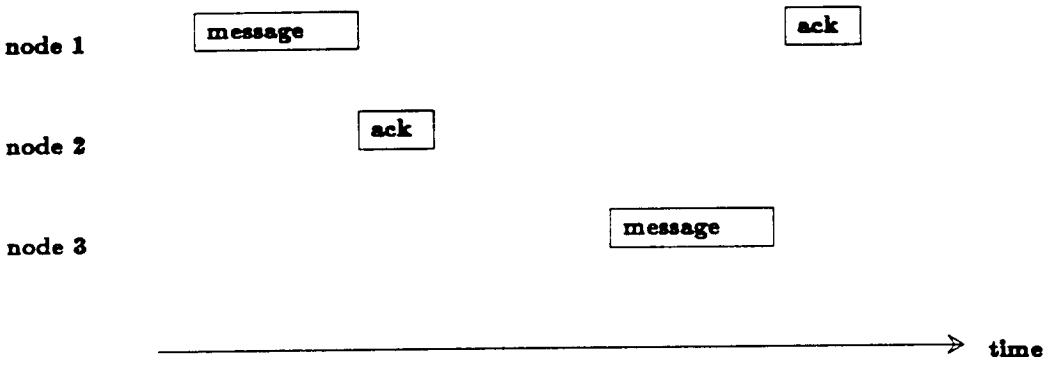



**Figure 6.1: Lightly loaded network**

However, when the network is busy, we have the situation depicted in Figure 6.2. In both types of Ethernet, the receiver will attempt to broadcast an acknowledgement following the reception of a message. On the normal Ethernet this acknowledge, with high probability, will collide with a transmission from some other node. Since no useful information is being transmitted during the transmission, some network bandwidth is lost. In the acknowledging Ethernet, the network will be reserved following a message for that message's acknowledgement. Therefore, there will be fewer collisions and the network will be better utilized.

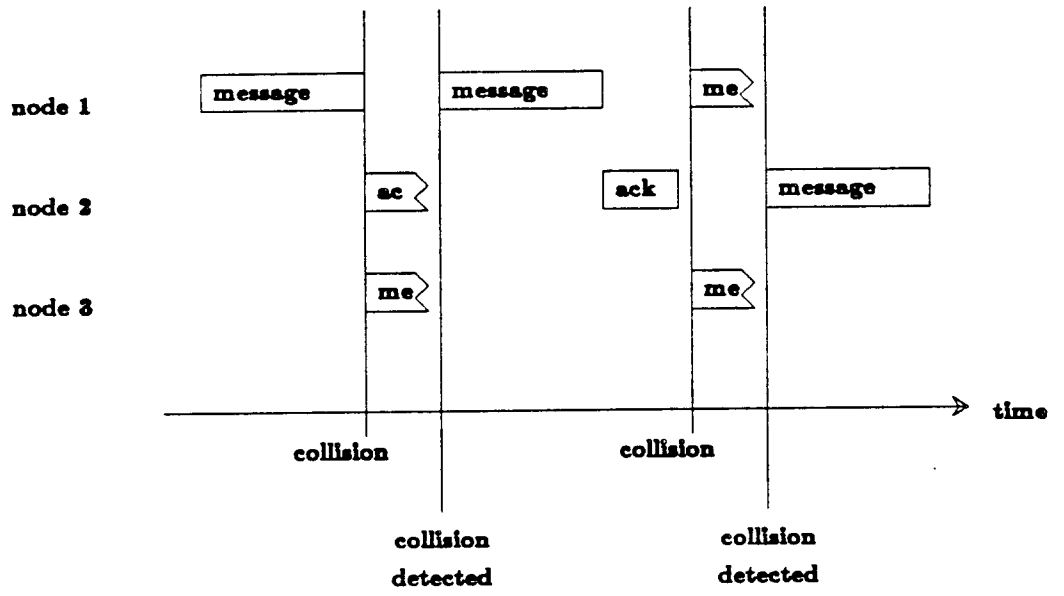



**Figure 6.2: Heavily loaded network**

The same principle can be used for published communications to acknowledge receipt by a recorder. A time slot is reserved after each message

transmission. During that time slot, the receiver waits for an acknowledge from the recorder. If one appears it accepts the message and eventually acknowledges it. If not it discards the packet exactly as if it had received a bad packet. The transport protocol will eventually cause the message to be resent.

### 6.1.2. Token rings

A similar solution can be used for token rings. In a token ring, one or more message slots circulate around the ring. The slot is preceded by a token field, Figure 6.3. When a node wants to send a message, it waits for a token indicating a free slot. It then removes the token and fills the slot with the message. When the message reaches the destination, the message is removed and the token is reinserted.

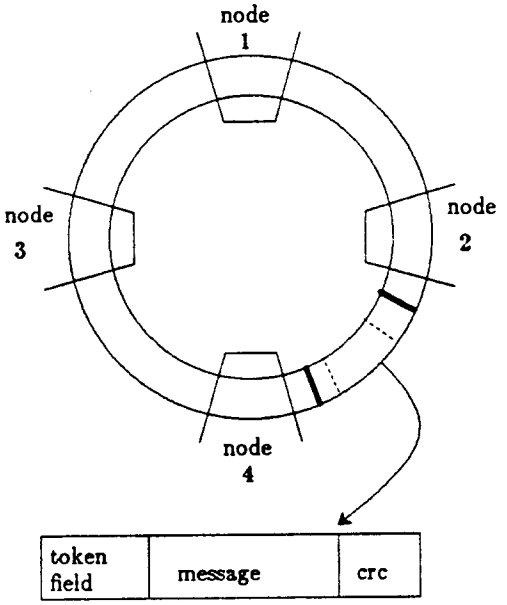



**Figure 6.3: A message in a ring**

For published communications we add an acknowledge field to the message slot, Figure 6.4. When a message is inserted into the ring, the acknowledge field is empty. Messages that have an empty acknowledge field are ignored by all nodes except the recorder. When the message passes the recorder, the recorder fills the acknowledge field and reads the message. If the message is incorrectly received, the last few bytes of the message (usually the checksum) are complemented, thereby invalidating the message. The message can now be read by the node for which it was destined. If the recorder could not successfully read it, neither will the receiver due to the invalidated checksum.

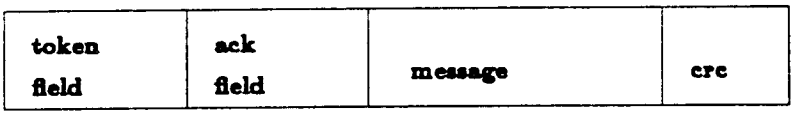



**Figure 6.4: Token ring with acknowledge**

## 6.2. Other configurations

So far, this thesis has considered only star configurations and broadcast media. This is due to their property of having a single point at which all messages can be intercepted and recorded. However, more general configurations can be supported if we allow more than one recorder to be used. Each recorder would publish messages for a disjoint subset of processors. In the most extreme example, we could attach one recorder to each processor.

It is rare, however, for networks to be so disjoint. More likely are cluster configurations made up of a number of a number of broadcast media network connected via a store and forward network. CM*[Swan et al 77] is probably the most famous of such cluster networks. However, a more common form of cluster is the local area network. Many LAN's are now attached to other LAN's via general topology store and forward networks. For example, our Ethernet at Berkeley is connected via the ArpaNet to the ring nets at MIT. Each of our networks can be considered clusters of the ArpaNet.

In these networks, a recorder can be attached to each cluster to perform recovery for that cluster alone. The great advantage to this scheme is autonomous control. Each cluster can decide for itself how and whether or not it will perform recovery. This is not possible with many other recovery mechanisms.

## 6.3. Multiple recorders for reliability

Recorders can be made more reliable by using TMR techniques, battery backups, etc. However there is always the possibility that the recorder will fail. Since the recorder will eventually recover, the system will not malfunction. However, while the recorder is down, no messages can be sent on the network.

Network availability can be increased by providing multiple recorders. During normal operation, all recorders record all messages. If there are $n$ recorders, $n-1$ can fail before the network becomes unavailable. There are three problems associated with this:

1) Coordinating recovery of processes between different recorders

2) Ensuring that all recorders record each message

3) Recovering recorders that failed

A fairly simple solution can be applied to all three problems.

Assume a broadcast network with $n$ processing nodes, labeled $P_i$, and $m$ recorders, labeled $R_j$. At any one time only one recorder is allowed to recover any particular processing node. We achieve this by assigning an $m$ element vector, $V_i$ to each processing node $P_i$. Each vector describes a priority ordering for all the recorders. If processor $P_i$ fails, it is recovered by the highest priority recorder in $V_i$ which is functioning.

For this recovery coordination to work, each recorder must be able to find out whether or not the higher priority recorders are functioning. It does this by querying them. Each recorder contains a copy of all the vectors, $PV_i$, which reference it. Whenever a recorder, $R$, detects a failure of $P_i$, it queries all recorders of higher priority than it in $V_i$ to see if any are willing and able to perform recovery. If they are not, or they do not answer in a set interval, $R$ performs the recovery. If some other recorder accepts the job, $R$ performs no recovery but continues to monitor node $P_i$. If $P_i$ does not recover in a set interval, $R$ periodically requeries its higher priority nodes to see if they are willing to recover. This is necessary to insure that $R$ will restart the recovery should the higher priority recorder fail during recovery of $P_i$.

Like the single recorder case, a positive acknowledgement is necessary from each recorder to insure receipt of messages. Once again, the solutions can be network specific. For the Ethernet, instead of adding one acknowledge slot, we add one for each recorder. The same applies for the token ring. Each message must have an acknowledge from all recorders before it can be used. If a recorder in $P_i$ detects the failure of a recorder of higher priority, it supplies the acknowledges for that recorder in addition to its own acknowledge, even though the higher priority recorder did not receive the message.

When a recorder recovers from a failure, it has to be brought up to date since its published messages do not include those sent while it was down. This could be done by stopping the system momentarily and having some other recorder bring it up to date. However, we can instead use a method that takes into account the periodic checkpointing of messages. Each process will eventually checkpoint (or can be forced to). When a recorder restarts, it queries the processing nodes for information about running processes, to rebuild its internal tables. Eventually, all the processes will naturally checkpoint or be forced to. The recorder will then be up to date and able to accept recovery responsibilities.

## 6.4. Transactions using published communications

In Chapter 2, we described the recovery aspects of atomic transactions. As a recovery mechanism, they can be completely replaced by published communications. However, atomic transactions are typically used in distributed data base applications also as a concurrency control mechanism. Transactions are usually made up of a number of phases. Early phases obtain information, work on it, and store on reliable storage intentions of updates to be performed should the transaction commit. The last phase is a commit phase at which point all processes taking part are committed to complete. Failures that occur before the last stage begins cause all the processes to abort. The state of the transaction, that is, whether or not it has committed, must be preserved across failures so that all processes react correctly. If the processes reside on different processors, then each processor must provide reliable storage for the intentions and transaction state records.

With publishing, the transaction semantics remain the same. However, there is no need to store intentions and transaction state in stable store. When a crashed process recovers, its intentions and transaction state will be rebuilt along with the rest of the process state. This means that each processor need not have reliable storage for the processes taking part in transactions. Only one reliable store is needed, the publishing storage. Depending on the size and design of the reliable stores, the single reliable store may be cheaper and more reliable than a number of them.

## 6.5. Debugging using published messages

One of the great problems of distributed debugging (or, for that matter, of any kind of debugging) is finding out what happened after the fact. Often, the trail to a problem's original cause has disappeared by the time the problem becomes apparent. A programmer would like some way of backing up a process, or processes, to the point where the problem originally occurred.

Published communications offers this as a side effect. When a process dies due to hardware failure, it is restarted at a previous point and brought up to date. It would be easy to change the publishing system to allow this to be done whenever a process terminates abnormally or whenever the programmer requests it. If requested, the process could not only be restarted at a previous checkpoint but also placed in a debug mode so that the programmer could step through its previous execution and watch what happens. In order for this to work, we need to the recorder would have to not the checkpoint at process termination.

Of course, the error may have occurred before the last checkpoint. If this is the case, the programmer may choose to abort the distributed computation

and restart it with a larger or even infinite checkpoint interval in order to be able to backup past the cause the next time it occurs. Alternatively, we might cause all messages and checkpoints belonging to a debugged process to be saved.

## 6.6. Optimizations

A number of optimizations can be performed to improve the performance and reduce the cost of publishing. In this section, we investigate two optimizations: not publishing recovery information for all processes and not publishing intranode messages.

### 6.6.1. Not recovering all processes

In previous sections, we have assumed that all processes must be recoverable; therefore, all the discussions and analyses have assumed that all messages must be published. However, there are a large number of processes which do not need to be recoverable. If we do not publish messages for these processes, we may greatly increase the capability of the recorder.

As an example, consider the processes measured to provide operating points for the queuing simulation in Chapter 5. Among the processes measured were a large number that could easily be restarted by the user should a crash occur. In general these were equipotent commands that provided information either about the system's use (man, apropos) or about its current state (ps, vmstat, pwd). If a crash were to occur during their execution, the user may not want to restart them. In the case of the system status commands, recovery may actually be the wrong action since the system state may have changed considerably subsequent to the crash.

The measurements also contained a number of I/O intensive processes. Most prominent among these were the disk to tape backups, which accounted for 15% of the messages in the maximum disk access rate operating point. If these processes were not considered recoverable, the recorder would be able to support one more VAX on the network.

### 6.6.2. Recovering nodes rather than processes

As we have seen, the greatest steady state cost incurred by publishing messages is the routing of intranode messages onto the network. This is done to allow the recorder to publish all messages. Without it, we would not be able to recover individual processes. However, not all sites may wish to recover single processes. For a number of reasons, they may wish to recover a node as a unit. Some may not be able to afford the extra cost for intranode messages. Others may find that node crashes are much more prevalent than single process failures, e.g., personal computers. For these systems, we would

like to treat the complete node as a single process. To do this with published communications, the node's behavior will have to be deterministic upon its input messages.

The only part of a node's behavior which can be determined from outside of the node is its output messages. These messages are just the collection of extranode messages sent by the processes within the node. Since the processes within the node are deterministic upon their input messages, the node's output messages will be deterministic upon its input messages provided that we can guarantee two properties:

1) During recovery, all processes in the node will receive the same messages in the same order as they did before the crash.

2) During recovery, the messages sent by the different processes will be interleaved in the same way they were before the crash.

Messages received by a process are made up of both intranode and extranode messages; to guarantee property 1, we have to synchronize the two. Also, the order in which intranode messages are received by processes depends on the scheduling. For example, if two processes are sending messages to a third, the order that messages are sent will depend on how the execution of the two processes are interleaved. To guarantee property 1, we must also make the scheduling of processes deterministic. This will, not surprisingly, also ensure property 2. If the sequencing of all messages sent by processes in the node is ensured, then so must some subset of those messages.

Therefore, the problems facing us are:

1) How do we guarantee that messages from off node will be correctly ordered with messages within the node?

2) How can we make scheduling deterministic?

Obviously, these problems have many solutions. We will outline one set of solutions here that assume that we are willing to double the number of extranode messages if that will allow us not to put intranode messages onto the network.

Synchronizing intranode and extranode messages during recovery can be done in the following way. Whenever an extranode message is received by a node, the node sends a message to the recorder. The message contains the unique identification of the extranode message and the intranode message last sent before it. When the node is recovered, the recorder would send with each replayed message the id of the intranode message it is to follow.

The ability to make a scheduler deterministic depends on the scheduler used on any particular node. Since schedulers vary greatly with respect to when processes can be interrupted and what fairness means in their

algorithms, we will not attempt to provide a general solution. However we can give a simple example of a deterministic scheduler to provide at least a starting point for others.

Our scheduler is a round robin scheduler that uses a single queue. The scheduler always runs the first process in the queue. The process runs until it has executed a predetermined number of instructions or until it attempts to read a message and none exist in its queue. If it stops because no messages are available it is taken off the queue. Otherwise it is put at the end of the queue. Processes waiting for messages are put back at the head of the queue whenever a message becomes available. In the absence of extranode messages, this algorithm is completely deterministic on the number of instructions executed. If a process is run again, it will behave the same way it did the first time since all factors controlling the algorithm (blocking of processes, unblocking of processes, length of execution interval) depend on the number of instruction executed.

The arrival of extranode messages, however, makes the algorithm nondeterministic since the extranode inputs are not synchronized with instruction execution. To make the scheduler behave the same during recovery we need to add some means of synchronizing the arrival of intranode messages with the instruction stream. This can be done by changing the information in the message sent out whenever an extranode message is received. Since we are counting the number of instructions executed, we can inform the recorder of how many instructions have been executed prior to receipt of the message. When replaying messages to the node during recovery, the extranode message will contain the instruction count. The recovering node will not use the message until that time. Since this time is more accurate than the message ordering information in the scheme for synchronizing intranode and extranode messages, it can replace it.

In many processors, it may not be possible to stop a process after executing some number of instructions. In that case, the scheduling algorithm can count some other quantity such as the number of kernel calls made by the processes. It is only necessary that the processes be deterministic upon that counter.

## 6.7. Summary

In this chapter, we have shown that published communications is indeed applicable to a large number of systems. We did this first by demonstrating how the two most popular types of local network, rings and Ethernets, can be changed to efficiently support message publishing. We then extended the number of applicable systems by showing that publishing is compatible with

clustered networks such as CM* and groups of LAN's connected via gateways.

This chapter also shows some possible secondary applications of message publishing. First, message publishing can replace the need for per machine stable storage in systems that use transactions for concurrency control. Therefore, publishing may greatly decrease the cost of such systems. We have also shown how a distributed debugger may be designed using the published information. An extremely simple form of the debugger has been invaluable in the debugging of the publishing system itself.

Finally, we have pointed out two ways to decrease the cost of the publishing system. By recovering all processes on a node as a unit, or by not recovering some processes, we can greatly reduce the number of messages that the recorder needs to publish. Therefore, we can use a lower performance, lower cost recorder if these optimizations are made.

# CHAPTER 7

## Conclusions

The intent of this research has been to develop a distributed recovery mechanism that is transparent to the user, allows time bounded recovery, and does not perturb non-failed parts of the system. Chapter 3 presented the basic model of published communications. In it we showed how published communications meets these goals.

However, published communications is meant to be much more than another mechanism that meets a set of carefully worded recovery goals. It is an attempt to make distributed systems simpler to deal with. By choosing a system with a single communication mechanism and a deterministic model for processes, we have greatly reduced the complexity of the system and, therefore, the recovery system. A simple recovery system, transparent to the user, in turn reduces the complexity of the programs.

We hope not only that publishing will be accepted as a feasible recovery technique, but that it will point the way to simpler distributed mechanisms.

### 7.1. Future Work

Our DEMOS/MP implementation is only the smallest usable subset of published communications. Before it can be accepted by the community at large, a complete version with checkpointing needs to be implemented in a widely used system. Network devices should be adapted with the recorder acknowledgement features to minimize the effects of publishing.

The variants discussed in the last chapter should be tried. Of special importance is the integration of publishing and a distributed debugger. The research in distributed systems is snowballing. However, current distributed debuggers are, at best, primitive. The publishing debugger offers a valuable tool in the ability to observe process histories and for obtaining fine control of debugged processes.

An investigation should be made into integrating publishing with process migration. This would allow processes to be recovered on processors other than the one on which they failed. In many cases, this would greatly speed up recovery time.

Finally, network protocols and distributed systems should be reengineered with publishing in mind. Currently, distributed applications and protocols

assume the worst, that is, that processes will die and that messages will be lost. Often, many redundant checks are made at different levels of programs and protocols to be able to survive the expected problems. As a result, current distributed applications tend to be much more complicated than their non-distributed equivalents. It is our claim that software engineered under a publishing system would be much less complicated and, perhaps, faster as a result of fewer checks. It should be determined experimentally whether or not this claim is correct.

# BIBLIOGRAPHY

[Arens 81]
G. 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 Operating Systems Principles*, pp. 22-29 (Dec 81).

[Baskett et al 77]
F. Baskett, J. Howard, and J. Montague, "Task Communication in DEMOS," *Proc. of 6th ACM Symposium on Operating Systems 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 Operating Systems Principles*, (Oct 1983).

[DARPA 82a]
DARPA, "Internet Protocol," RFC 791 (1982).

[DARPA 82b]
DARPA, "Transport Control Protocol," RFC 793 (1982).

[Diffie and Hellman 77]
W. Diffie and M. Hellman, "Exhaustive Cryptoanalysys of the NBS Data Encryption Standard," *Computer* **10**(6) pp. 74-84 (1977).

[Fabry 74]
R. Fabry, "Capability based addressing," *CACM* **17**(7) pp. 403-412 (July 1974).

[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).

[Farmer & Newhall 69]
W. Farmer and E. Newhall, "An Experimental Distributed Switching System to Handle Bursty Computer Traffic," *Proc. of the ACM Symposium on Data Communications*, pp. 1-33 (Oct 1969).

[Fraser 79]
A. 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. 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 Operating Systems Principles*, pp. 93-99 (Dec 1977).

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

[NBS 75a]
National Bureau of Standards, "Data Encryption Standard," *Federal Register* **40**(52)(March 1975).

[NBS 75b]
National Bureau of Standards, "Data Encryption Standard," *Federal Register* **40**(149)(Aug 1975).

[Von Neuman 56]
J. Von Neuman, "Probabilistic logics," in *Automata Studies*, , Princeton University Press, Princeton, NJ (1956).

[Pierce 72]
J. Pierce, "Network for Block Switching of Data," *Bell System Technical Journal*, pp. 1133-1143 (July 1972).

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

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

[Randell 75]
B. Randell, "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering* **SE-1**(2) pp. 220-232 (June 1975).

[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).

[Randell et al 78]
D. Randell, P. Lee, and P. Treleaven, "Reliability Issues in Computing System Design," *ACM Computing Surveys* **10**(2) pp. 123-166 (June 1978).

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

[Russell 77]
D. Russell, "Process backup in producer-consumer systems," *Proc. of 6th ACM Symposium on Operating Systems Principles*, pp. 151-169 (Dec 1977).

[Sauer et al 81]
C. Sauer, E. MacNair, and J. 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. Schneider, "Synchronization in Distributed Programs," *ACM Transactions on Programming Languages and Systems* **4**(2) pp. 179-195 (1983).

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

[Shooman 68]
M. Shooman, *Probabilistic reliability; an engineering approach.* 1968.

[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).

[Svobodova 81]
L. Svobodova, "Recovery in distributed processing systems," *NATO AGARD conference proceedings*, (AGARD-CP-303)(June 1981).

[Swan et al 77]
R. Swan, S. Fuller, and D. Siewiorek, "Cm* - A Modular, Multi-Microprocessor," *Proc. of the National Computer Conference*, pp. 637-644 (1977).

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

[UCB 82]
University of California, "4.2 BSD UNIX Programmer's Manual," University of California, Berkeley (October 1982).

[Wilkinson 81]
W. 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).

[XEROX 81]
XEROX Corporation, "Internet Transport Protocols," XEROX System Integration Standard 028112 (December 1981).

[Young 74]
J. Young, "A First Order Approximation to the Optimum Checkpoint Interval," *CACM* **17**(9) pp. 530-531 (Sept 1974).

[Zilog 80]
Zilog, Inc., "Architectural Features," Z8000 CPU Technical Manual (May 1980).