Fault Tolerance Under UNIX®

ANITA BORG, WOLFGANG BLAU, WOLFGANG GRAETSCH,
FERDINAND HERRMANN, and WOLFGANG OBERLE
Nixdorf Computer

The initial design for a distributed, fault-tolerant version of UNIX based on three-way atomic message transmission was presented in an earlier paper [3]. The implementation effort then moved from Auragen Systems[1] to Nixdorf Computer where it was completed. This paper describes the working system, now known as the TARGON/32.

The original design left open questions in at least two areas: fault tolerance for server processes and recovery after a crash were briefly and inaccurately sketched; rebackup after recovery was not discussed at all. The fundamental design involving three-way message transmission has remained unchanged. However, in addition to important changes in the implementation, server backup has been redesigned and is now more consistent with that of normal user processes. Recovery and rebackup have been completed in a less centralized and thus more efficient manner than previously envisioned.

In this paper we review important aspects of the original design and note how the implementation differs from our original ideas. We then focus on the backup and recovery for server processes and the changes and additions in the design and implementation of recovery and rebackup.

Categories and Subject Descriptors: C.2.4 [**Computer Systems Organization**]: Computer Communications Networks—*distributed systems*; D.4.3 [**Operating Systems**]: File Systems Management—*maintenance*; D.4.4 [**Operating Systems**]: Communications Management—*message sending*; D.4.5 [**Operating Systems**]: Reliability—*backdrop procedures, checkpoint/restart, fault-tolerance*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Atomic multiway message transmission, crash handling, file system availability, roll forward recovery, server architecture

1. INTRODUCTION

1.1 Background

The 1980s have produced reports on a number of efforts, both commercial and academic, to provide fault-tolerant operation. These efforts have ranged from languages supporting fault-handling semantics [6, 7], to special-purpose mechanisms for database recovery [4, 5, 15], to hardware- [11, 12] or software- [2, 3, 9]

® UNIX is a trademark of Bell Laboratories.

[1] Auragen closed its doors in May 1985.

Authors' current addresses: Anita Borg, Western Research Lab., Digital Equipment Corp., 100 Hamilton Ave., Palo Alto, CA 94301; Wolfgang Blau, Tandem Computers GmbH., Postfach 560214, Ben-Gurion-Ring 164, 6000 Frankfurt/Main 56, West Germany; Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle, Nixdorf Computer GmbH, Unterer Frankfurter Weg, 4790 Paderborn, West Germany.

based systems allowing recovery of arbitrary programs. Our work falls into the last category.

In an earlier paper [3], we presented the original design and early implementation of a UNIX-based operating system whose goal was to ensure that an arbitrary process could survive any single hardware failure. The system has since been revised and completed at Nixdorf Computer. This work describes the Nixdorf system known as TARGON®/32. The system uses a process-pair scheme reminiscent of that introduced by Bartlett [2]. Process checkpointing makes heavy use of the virtual memory pager. Recovery of actions between checkpoints is supported by atomic three-way message transmission. Since only processes survive a crash, critical operating system functions have been moved out of the kernel into recoverable server processes. The current, running, system recovers from any single hardware failure, some combinations of more than one failure, and also recovers from many nondeterministic software problems.

Unlike the Stratus system [12] and to some extent DEMOS/MP [8, 9], which require dedicated hardware to support fault tolerance, we demand that all processors be available for productive execution in the absence of failure. Processors never exist solely as backups. The Stratus system and others based on total hardware redundancy use duplicate hardware only to mirror ongoing computations. While recovery is immediate should a failure occur—making such systems appropriate for critical real-time applications—the extra hardware provides no increase in computing power. DEMOS/MP requires a special network node to accumulate information needed for recovery. This design requires hardware that is apparently used for nothing but recovery, yet does not support real-time processing because of the time required to recover crashed processes. Since our system was designed with the pressures of a transaction processing environment in mind, rather than those of real-time process control, we were willing to sacrifice immediate recovery for productive capacity. Still, efficiency in the absence of failure should suffer only minimally, and recovery should not take longer than a few tens of seconds. Our backup and recovery mechanisms are distributed throughout the system using a small portion of the productive capacity of each machine on a local network.

Our requirements differ from Tandem's NonStop® OS [2] and Tolerant's [14], in that complete transparency was a primary goal. In each of these systems, mechanisms are provided by the operating system that allow recoverable user programs to be written. For user programs to be recoverable under NonStop, they must explicitly checkpoint critical data and must either be rewritten or run through a special preprocessor. Tolerant's use of database recovery techniques requires that the boundaries of a transaction be specified using **begin-transaction** and **end-transaction** system calls, so that they can be restarted after a failure. As in NonStop, this requires rewriting or preprocessing. On the other hand, we require that the TARGON/32 system be able to backup and recover any unmodified user program. Since backing-up processes does take some resources, we felt it necessary that fault tolerance of processes, or at least groups of processes, be optional, and that the degree to which they are

® TARGON is a trademark of Nixdorf Computers.

®NonStop is a trademark of Tandem Computers.

backed-up also be up to the user. Once these choices are made, however, backup creation, checkpointing, and recovery should be done automatically and transparently.

An alternate method for providing automatic and transparent fault tolerance is suggested by Strom and Yemini [13]. This method uses checkpointing combined with delayed logging of messages, that is, messages can be sent and used before they are copied to an alternate location, in this case a log on stable storage. The drawback of this method is that processes throughout the system can be required to rollback and reexecute code. That is, not only those processes that were running on a crashed machine must recover, but often correspondents must recover as well. On the other hand, this method is not constrained by the requirement of atomic message transmission and is therefore more easily portable and extendable.

For our backup and recovery scheme to work, three criteria must be met:

—a crashed process's state must be available;
—all messages that would have been available to the primary in that state or since that state was reached must be available in the correct order; and
—the process must behave deterministically.

We shall briefly describe how we meet these criteria: we specify the hardware requirements, then the organization of user, server, and backup processes, and lastly, the algorithms for backing-up user processes. Next we show why the recovery algorithms for user processes are problematical for server processes and how they have been modified to ensure server process recoverability. Finally, we look into the details of crash detection, recovery, and rebackup.

1.2 System Architecture

A Targon/32 system is a local area network of 2 to 16 machines connected via a fast dual bus. Each machine is a shared-memory multiprocessor consisting of three processors. Three was the number of processors required to balance processor and memory speeds. Each machine runs an operating system kernel responsible for the creation and scheduling of processes and for managing intermachine and interprocess message communication. One of the three processors executes kernel code to handle incoming and outgoing messages and the creation, maintenance, and recovery of backup processes. The remaining two processors execute UNIX-style processes as well as much of the system-call-related kernel code. Kernels are independent in the sense that they are not backed-up. While processes recover after a crash, the kernel state of the crashed machine is lost. In the remainder of this section we detail only those hardware characteristics that are directly relevant to the support of process backup and recovery.

A machine may or may not be directly connected to any peripherals, but all peripheral devices are dual-ported. Disks can be mirrored. A bus or a peripheral device can crash without affecting process execution. Should a hardware error cause an individual machine to crash, its processes must be recovered on another machine.

Two machines play a special role: those machines that are connected to the disk containing the root file system. The file system is hierarchically organized and accessible through a single root. One of the machines is known as the *root machine*; the two together are referred to as the *root pair*. As will be detailed later, those parts of the system that are centralized run on (and are backed-up on) the root pair.

Our backup and recovery schemes require atomic three-way message delivery. Messages between backed-up processes are sent to three destinations: the target process and both the sender's and receiver's backups. Either all three destinations must receive the message, or none receive it. The arrival of the message at its three destinations must not be interleaved with that of any other message, ensuring that a primary and its backup always receive messages in the same order. In other words, if two messages are sent, one must reach all its destinations before the other arrives at any of its destinations.

In our implementation, the bus hardware and low-level software driver protocols guarantee such atomicity using the following algorithm:

—All machines listen for their address to come across the bus.
—A machine wishing to send a multiway message (the sender) requests bus mastership. On receipt of mastership, it transmits the three destination machine identifiers and waits.
—A machine seeing its address on the bus prepares to receive. If it cannot take the message at that time (e.g., it is currently receiving a message on the other bus), it sends a NACK.
—A machine that can neither receive nor NACK is dead. The mechanism for determining that a machine is dead is described in Section 5.1.
—If the sender receives no NACKs within a specified period of time, it sends the message across the bus once.
—The message is picked off the bus by each of the ready receivers.

2. PROCESSES

2.1 Processes and Their Backups

The fundamental recoverable unit of execution is the process. A process is an execution of a program whose scheduling and access to local resources is controlled by the operating system kernel. Processes execute kernel code only via a limited number of system calls. Processes communicate with each other and receive all input via message. They can be run backed-up or not, at the discretion of the user or system administrator.

Each backed-up *primary process* has an inactive *backup process* on another machine. A backup process consists of sufficient information to begin execution and recompute, based on the same code and input messages that were used by the original primary, and eventually catch up and take over completely as a primary process. As shown in Figure 1, communication between processes uses three-way atomic broadcast to ensure that all messages to a primary process are also sent to its backup and all messages sent by a primary process are counted by its backup as *writes-since-sync*.

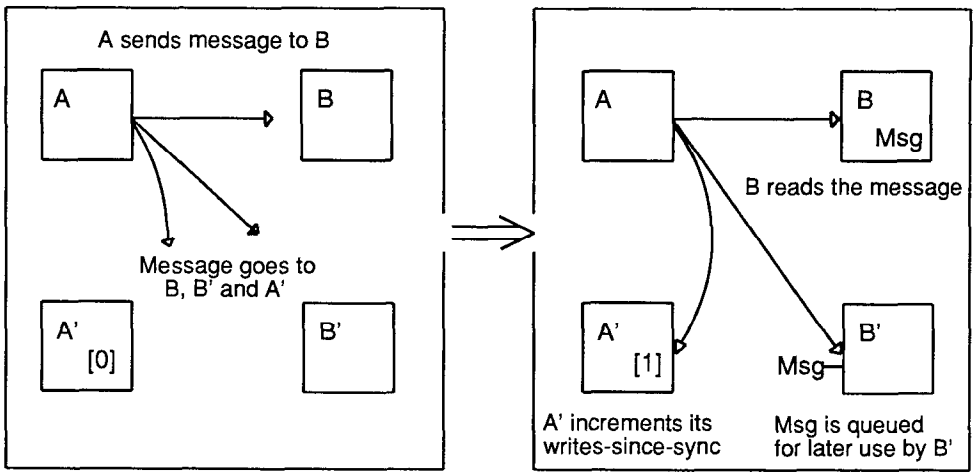

Fig. 1. Three-way message transmission between primary processes A and B. The message goes to backup processes A′ and B′ as well.

In order to avoid recomputation from the primary's initial state by the backup upon failure, a primary process and its backup are periodically synchronized so that the backup can recover from a more recent point. That is, the primary's state is saved for the backup. We refer to this as the *sync* operation. A primary and its backup are automatically synchronized whenever the primary has read more than a system-defined number of messages, or if it has executed for more than a system-defined amount of time since last synchronization. Synchronization utilizes the virtual memory system to save the state of the address space: dirty pages are sent via message to the page server (see Section 3.2.1), the page server keeps duplicate copies of pages sent after a sync. Also, a *sync message* containing a small amount of state information is sent to the kernel in the backup's machine.

On failure of the primary's machine, the backup will demand-page in the address space of the primary as of the last synchronization and recompute the current state based on the same messages that were seen by the primary. It will avoid resending messages already sent by the primary by using its writes-since-sync count. After this, any messages saved for the backup, but already read by the primary, are discarded and the backup's writes-since-sync is zeroed.

2.2 Server Processes

A number of systems including DEMOS/MP [8, 9] and Accent [10] have moved some functions usually thought of as provided by the operating system out of the kernel and into *server processes*. Server processes execute in their own address space but are able to provide services to other processes. This has most often been done for reasons of modularity, modifiability, and distribution; to this list we add recoverability.

In our system, functions that must be globally available, globally consistent, and backed-up cannot be provided by the independent local kernels whose state

is lost on crash. These functions are performed by server processes. Some servers execute essentially like user processes, but are the privileged repositories of special system information. Others, called peripheral servers, manage access to logical or physical devices and must reside in the machines connected to their associated devices.

The following servers always exist:

—*File servers*—A file server manages all access to a file system. It receives requests from user processes (e.g., open, read, write) and communicates with the local disk driver via messages. It buffers disk blocks in its address space. There is one root file server that resides in the root pair: its primary is on one machine, backup on the other. There is also one file server for every other mounted file system.

—*Page servers*—A page server manages virtual memory backing store. It receives pages for storage and requests for pages via messages. Each server uses its own logical disk and manages the pages for some subset of the system's primary processes. It also maintains the data space for backup processes. The number of page servers in a system is configuration-dependent.

—*TTY servers*—A TTY server manages communication with terminals and related devices. There is one TTY server in each machine connected to such devices.

—*Raw servers*—A raw server manages unstructured access to disk or tape drives.

—*Process server*—There is one process server per system. The process server and its backup reside in the root pair. It is the repository of most centralized information in the system. The process server records the current system configuration and is responsible for downloading operating systems to newly booted machines. UNIX functions such as "ps", which lists all active processes in a system, and "kill", which allows an asynchronous signal to be sent to an arbitrary process in the system, require that global system information be available. Such requests are handled by the process server, which periodically collects state information from each machine.

When efficiency is essential, a server's address space is locked into memory and cannot be paged out. All page servers and the root file server reside permanently in memory.

2.3 Process Families and Backup Modes

In UNIX, all processes on a single machine form a single family tree rooted in a special system process. Parent and child processes are able to share local resources, for example, memory and open files. It only makes sense for processes sharing such resources to be backed-up in the same way and on the same machine. In order to support balanced distribution, as well as the simultaneous existence of recoverable and nonrecoverable processes, it was necessary to be able to start a new process at a specified location with different resource requirements and backup characteristics from its parent's; the familial relationships had to change.

The processes in a TARGON/32 system are divided into *families*. All members of a family reside on one machine and have a common ancestor; there is no

common ancestor for all processes systemwide. Server processes exist as single-member families. The process family is the basic backed-up unit in our system; all processes in a family are backed-up in the same way and all backups for the same family reside on the same machine.

A new family is created by the new operation **wexec** for *walking exec*. In UNIX, exec transforms the calling process into a new process executing new code in a fresh address space. Walking exec causes the creation of a new *head of family* process, in a new execution environment, possibly on a different machine, which shares nothing with the caller. It sends a request to the process server, which determines the locations of the new family and its backup. The process server then sends a message to the primary and backup machines where the head of family process is created. Thereafter, all descendents of the new process belong to the new family. The family shares no resources with its creator, can reside on a different machine, and can have different backup characteristics.

An argument to a **wexec** specifies how the family is to be backed-up. The current kernel allows families to be backed-up in two ways depending on whether, when, and where a new backup process is created after a crash occurs. *Quarterbacks* run backed-up until a crash occurs, but no new backup is created for them after a crash. *Halfbacks* have new backups created only when the machine in which the original primary or backup resided is returned to service. Peripheral servers are backed-up in this way because their primary and backup must be located in the two machines connected to the device they control. Process families can also run without backup. The backup mode for a family is specified in the **wexec** system call at the time the family is created. Families of all types are permitted to exist simultaneously and interact.

The original design envisioned a fourth option, *fullbacks*. These were to become backed-up again as soon as possible after a crash on any available machine. Unfortunately, some of the most critical processes in the system, the peripheral servers, can run and be backed-up on only those machines to which their associated device is connected; they are necessarily halfbacks. Unless peripherals are ported to more than two machines or servers can communicate with drivers on other machines, this constraint will remain in place. Thus far, it has seemed unnecessary to make user processes more robust than system servers can be, so fullbacks have not been implemented.

2.4 Interprocess Communication

A large part of process recoverability involves ensuring that primary and backup processes receive the same messages. We have described the hardware mechanism that assures atomic delivery. In this section, we consider the required software support.

Processes send and receive messages via *channels*. A channel is a recoverable two-way communication mechanism. At one end, it is represented by a routing table entry with identification and routing information, as well as a queue for holding unread messages.

From a process's point of view, a channel is just another version of the general UNIX-style file: it is opened, written to, and read from in a similar manner. From the kernel point of view, however, all variations of files are implemented

as forms of channels between user and server processes. For example, an open file is represented by a channel to the file server managing the file. A file read causes a request (essentially a remote procedure call) to be sent to the server, which reads the file and returns its contents in a message on the same channel.

Channels that are explicitly opened can be explicitly read from and written to. Other channels are transparent to the user and are used implicitly, for example, a process's root and working directories are channels to file servers. Open requests are sent over these channels. Completion of an open involves sending a message to both backup and primary so that data structures are set up in both places. Thus, once a channel is open, there are queues for incoming messages at both the primary and backup. This relies on the atomicity of message transmission [3].

When messages arrive at a machine, the low-level, high-priority bus interface code places all messages on a general input queue. It assigns an increasing *arrival number* to each message that can be used when a process wishes to read the earliest arriving message. Later, lower-priority kernel code deals with the messages in arrival order. Most messages are delivered to channels and placed on routing table queues. Others are handled directly by the kernel; for example, the sync and machine-dead messages described in Sections 3.2.1 and 5.1. A *read* on a channel causes a message or some part of a message to be removed from the queue.

3. BACKUP AND SYNCHRONIZATION OF USER PROCESSES

3.1 Creation of Backup Processes

Since many processes are short-lived, we decided to delay the creation of backup processes for as long as possible. Although any messages received by a process from the time of its creation must be saved in the backup machine, it is not necessary to save a state from which to recover until the parent process dies or syncs or the new child process syncs. A crash at any prior time will cause the parent's backup to recover and reexecute the child's creation.[2] The system must ensure that the child will then execute relative to the accumulated messages.

Upon process creation (the fork operation), a message called a *birth notice* is sent to the machine containing the parent's backup. The arrival of a birth notice causes routing table entries to be made for the channels that are created on fork and must be there to receive backup messages. Backup entries for channels inherited from the parent already exist. The birth notice is also used during recovery to assure that reexecution of a fork returns the correct value (see Section 5.2).

The remainder of the backup process, its state data structure and backup page account, are created the first time the new process syncs. When the parent process synchronizes with its backup or exits (dies), it must force any children that do not yet have backups to sync and thus create backups. This ensures that

[2] One exception is that backup processes for heads of family, including the servers, are created on receipt of the backup copy of the walking exec message, since these messages will not be resent on recovery.

shared resources will be correctly maintained. In many cases, short-lived processes will not have to have a backup process or a backup page account.

3.2 Synchronization of User Processes

The states of a process and its backup are made identical during the *sync* operation. The current address space is saved with the page server; the state data structure and message queues on the backup machine are updated. Since this operation goes on during normal execution, i.e., in the absence of failures, it is essential that it be efficient. In particular, it is essential that it interrupt the execution of primary processes for as short a time as possible.

A primary process executes a sync whenever its count of reads or execution time since last synchronization exceeds a configurable amount. The sync is initiated automatically by the kernel. Sync operations by one process are independent of those by another (and so do not delay other processes) except in the case of a parent forcing its child to sync (see Section 5.3).

Normally, processes sync only immediately before return from a system call or page fault, or at the beginning of a new time slice. This assures that the kernel stack can be easily reconstructed for the backup without reliance on local kernel data such as physical addresses. It is possible, however, that a process might be forced to sync in the middle of a system call while awaiting a response from a server associated with a slow device such as a terminal. In this case, the process syncs as though it were just about to enter the system call. Again kernel stack reconstruction is straightforward.

3.2.1 *Action by the Primary.* The sync operation takes place in two parts. First, the normal paging mechanism is used to send all dirty pages, via message, to the page server. A dirty page is one that has not been sent to the page server since its last modification. The page server, which sees no difference between these pages and any other it receives, adds them to the primary's page account. Since the user stack is kept in pages owned by the user, rather than in kernel space, it will be sent to the page server if it has changed.

The second part of the operation constructs a *sync message.* This message contains:

—All machine-independent information kept about the process's state. For example, the virtual address of the next instruction to be executed, accounting information, register save values, etc.

—Channel information for every open channel. If the channel has been read from, the number of messages read since the last sync is sent.

—A small amount of information allowing construction of the kernel stack on recovery so that the process appears to be just entering or just returning from a system call.

The sync message is sent to the machine of the process's backup and to the page server and its backup. Once the sync message has been placed on the outgoing queue by the primary, the process can continue normal execution; it need not wait for the page or sync messages to be sent. If the primary crashes before the message leaves the machine, the backup will take over from an earlier

point. Because messages leave the machine in the order in which they are placed on the outgoing queue, any subsequent message sent by the primary will not reach its backup until after the sync message has been processed.

3.2.2 *Updating the Page Account.* The page server's response to the sync message is to make the backup's account identical to that of the primary. Backup pages that are no longer needed are freed. Immediately after a sync, only one copy of each page will exist. The accounts will start to differ only when new pages are received from the primary. Then, two copies will be kept only of those pages that have been modified since the last sync. Thus the page server uses a copy-on-write strategy to avoid duplication of pages between the primary and backup accounts.

The page server receives the sync message if and only if the message is received at the backup's machine. Thus, the backup page account is guaranteed to be in a state consistent with that of the backup process.

3.2.3 *Updating the Backup.* When the sync message arrives at the backup's machine, the kernel uses the contents of the message to update the backup's state and channel information. If a state structure does not yet exist, one is allocated. Channel information and message queues are updated. The writes-since-sync count is zeroed and the state structure is updated.

At the completion of these operations, the backup's state is the same as that of the primary at the time it issued the sync (though the primary can have progressed further by the time the backup is actually updated). The messages available to the backup are consistent with that state, as is the backup's page account.

3.3 Deterministic Execution

In addition to the availability of messages and state, successful recovery requires that processes execute deterministically with respect to the messages they send and receive. Given that our operating system is UNIX-based, this was the most difficult criterion to meet.

Since the kernels on various machines are not synchronized, a user process and its backup must be insulated from local differences, for example, machine's local time, a process's priority at a particular point in its execution, or the number of pages it has in memory. Regardless of differences between machines, every interaction between the kernel and a backup after crash must appear to the backup as it did to the primary.

User processes interact with the kernel in two ways: synchronously via system call or page fault and asynchronously as the result of a signal.

3.3.1 *Synchronous Interaction.* We ensure that a system call will return the same value to a recovered backup as it did to the primary in one of two ways. Any information that is returned directly by the local kernel must be maintained by the backup kernel and unaffected locally. For example, the process id, which is returned by the **get process id** system call, is a globally unique identifier, which is sent to the parent's backup on fork and to the backup itself on first sync. Other synchronous system calls return information that is received via message. Since the same messages are available to the backup, they are certain

to return the same answer. For example, the **read** system call sends a read request message to the file server or to a tty server on a channel and receives its answer on the same channel. The **time** system call sends a time request message to the process server and receives its answer via message.

The sequence of page faults generated by a process is unavoidably dependent on the environment of the local kernel. However, as long as page faults are transparent to processes, they cannot affect computations or communication. Therefore, we simply do not back-up the user end of the channel to the page server on which pages are paged out and demanded back. The server end of these channels must be backed-up normally. If the server's machine crashes in the middle of servicing a page request, the server's backup must provide the page.

3.3.2 *Asynchronous Interaction.* The most problematic kind of asynchrony in a UNIX system is during the handling of signals. Signals (software interrupts) can be sent by an arbitrary process to any other process by specifying its process id. Since the association between process id and location is known only by the process server, signals are managed centrally by the process server. Any operation that generates a signal in UNIX (e.g., a call to **kill**, **alarm** expiration, or typing certain control characters at a terminal) generates a message to the process server requesting that a signal be sent. The process server either returns an error message or sends a signal message. The signal is sent to both a process and its backup and is queued on a process's *signal channel.*

Asynchrony results because signals are to be dealt with at the time of arrival rather than being ordered with respect to other messages or deterministically based on a system call. A signal can be *ignored* and discarded or it can be *handled,* causing the process to execute signal-related user code.

To maintain determinism, we must assure that any signal that is handled by the primary is handled in the same way by the backup, and any signal that is ignored by the primary is also ignored by the backup. This may not always be the case. For example, suppose that a signal message arrives for a primary and its backup. To be consistent with UNIX, the signal must be dealt with immediately by the primary at an arbitrary point in its execution, but is merely queued on a special signal channel for the backup. If the backup does not know when the primary took the signal and there is a crash, the backup will begin execution at the point of the last sync, will find the signal pending, and will deal with it immediately and at an earlier point in its execution than did the primary. If the primary modified the signal handling actions between the last sync and the point at which it actually took the signal, the backup will take different actions from those taken by the primary.

It is nearly impossible to send information to the backup specifying precisely at which point the primary dealt with the signal and then, on recovery, to have it delay signal handling until precisely that point. To assure that signals are handled at the correct moment, the primary syncs just before handling any signal. This guarantees that, on recovery, the backup will immediately find the signal pending and will handle it at exactly the same place as did the primary.

Signals that are ignored are removed from the primary's queue but are counted. Whenever a message is sent by the primary, the count of ignored messages since the last send is piggybacked on the message. The count is used in the backup's

machine to remove ignored signals from the backup signal channel. This ensures that on recovery only signals that were handled or were not dealt with by the primary are on the signal channel.

4. BACKUP AND SYNCHRONIZATION OF PERIPHERAL SERVERS

The early design [3] envisioned a totally different backup scheme for server processes, using active backups for servers that would keep themselves up to date. This drastic difference turned out to be unnecessary. The scheme for user processes does not work for peripheral servers because messages might be received in different orders and state might not be available. In spite of this, we were able to use the basic ideas with some modification. A server backup is inactive, communicates with user processes on normal backed-up channels, and synchronizes to provide state and to adjust message queues for its backup.

The scheme for user processes does not work for servers because the servers have been designed with efficiency in mind—and those designs actively violate the three criteria listed in Section 1.1. The servers for disk-like (block-special) and tty-like (character-special) devices are implemented differently from each other. Here we describe only the details for a file server. Other disk-like servers are similar but simpler. First it is necessary to understand the reasons for the violations of the criteria. We then describe in detail how these problems were overcome.

4.1 File Server Communication

The first source of problems is a method of communication between the server and disk driver that violates the criterion that the same messages arrive in the same order for primary and backup. If messages arrive in a different order, they might be handled and responded to by the recovering backup in a different order. A count of messages sent since last sync cannot be used to avoid resending messages on recovery.

The server communicates with user processes via normal backed-up channels, but it communicates differently with the disk driver. In order to avoid repeated transmissions of large blocks of data over the global bus and the memory usage required to maintain multiple copies of such messages, the file server and disk driver communicate over a local (not backed-up) channel. In fact, the driver has been written as kernel code, not as a backed-up process. On recovery, the backup file server will be missing all of the disk responses that went to the primary, and so must reissue all disk requests.

The file server reads all messages using a special system call, **svr_read**, which gives priority to messages from the disk driver. It returns a disk response if there is one, otherwise it returns the earliest arriving user request. If the file server were single-threaded, the recovering server could simply reissue disk requests and wait for their completion. This is how the initial version of the file server worked.

Current file servers are multithreaded to increase efficiency by making use of the time that the driver needs to handle a disk request. Instead of waiting for a disk response, another user request can be read. Since the servers are not interrupt-driven, one request is executed until no more can be done: the request

is complete, requires action by the disk, or finds a resource locked. Consequently, it is possible that multiple disk requests are outstanding. Since the disk controller does local optimization, the responses to these requests can arrive for the recovering server in a different order than they did for the primary server.

The solution is to modify **svr_read** so that it keeps a compact history of requests read. The history is periodically sent to the backup and used on recovery. Since user requests will be read in arrival order (the same for server primary and backup), and this defines the order in which disk requests are issued, disk requests can be deterministically numbered and their responses tagged with that number. The history only need reflect the interleaving of numbered disk responses in the sequence of reads. A history array of the form ⟨u2, d2, d1, u1, d3⟩ indicates that two user requests were read, followed by the response to disk request 2, then the response to disk request 1, then one user request, and finally the response to disk request 3.

The history is sent to the backup machine through the use of a special write system call, **svr_write**, which is used for all file server writes. A **svr_write** piggybacks history information on an outgoing message. That is, every message that goes out to a process, rather than to the disk driver, contains all history generated since the last write. If the history does not fit in the message, a special uncounted message is sent before the write is done. This is a very rare event, since writes are frequent. At the backup machine, the history segment from the message is appended to a backup history table before the message is counted and discarded.

On recovery, **svr_read** uses the available history to decide which message to return to the server. All user requests are guaranteed to be there. If a disk request is specified but has not yet arrived, the server waits. This forces the server to handle the identical set of messages read by the primary prior to crash.

4.2 File Server's Address Space

Another optimization causes the primary file server's address space to be unavailable on recovery. Most of the file server's address space is buffered disk blocks or inode blocks (UNIX terminology for file description blocks). One of the commands that the server handles frequently is the UNIX-style file system, sync, which we call *Fsync*. This causes all dirty buffered blocks to be written out to disk. As a result, most of the file server's address space is on its own disk. It seemed wasteful to duplicate this effort by writing the file server's address space to the page disk on sync. Instead of demand paging in its address space, a recovering file server can reconstruct its state from its own disk.

The solution involves combining sync and fsync into a single operation. Unlike user processes for which the sync operation is totally transparent, file servers (and other peripheral servers) explicitly initiate their own syncs. They sync either upon handling an fsync request or after handling some number of other requests. Before performing a sync the fileserver must wait for the completion of any outstanding disk requests. The operation itself involves writing the usual blocks to disk together with state information, including a list of the blocks currently in the buffer pool and its table of inodes. A sync message is sent to the backup machine where message queues are updated and the history vector is cleared.

A file server recovers explicitly. It comes up fresh, as though it had just been created. It then reads information from the disk to determine whether it is a new file server or a recovering backup. A recovering backup reads in state information, which allows it to reconstruct its buffers and inodes. Once this is done it begins to execute normally, doing **svr_read**s and handling user requests. The internals of **svr_read** ensure that while a history array exists, it is used to control the order of reads. The normal mechanism, using the write count, ensures that no messages are resent.

4.3 Availability of the File System

Since a recovering file server reconstructs its buffers by reading blocks from the file system, the file system in the state as of the last sync must be available. The existence of that version of the file system is also necessary during recovery as the file server redoes requests. For example, if a file has been deleted since sync and a read request is reissued, the disk driver, and thus the recovering file server, will behave differently than the primary. Unfortunately, the contents of the disk can change between syncs, at least during the *Fsync* that constitutes the first phase of the sync operation.

The solution is to use a copy-on-write strategy between syncs, rather than overwriting existing blocks. Logically this corresponds to keeping two versions of a file system.[3] An early version of the file system organization described here is discussed in Arnow [1].

There are two root nodes on disk. At any given time one of them is valid for recovery. We refer to the other as the alternate root. Associated with each root is state information (the state tables described above), the most recent being that associated with the currently valid root. Changes to the file system are done relative to a copy of the valid root kept in memory in the primary file server's address space, and in a nondestructive manner, as seen in Figure 2(a–d). Freed blocks, which contain the old data, are added to a *semi-free list*, and cannot be reallocated until after the next sync. Therefore, the unmodified file system still exists rooted in the valid on-disk root node.

If a crash occurs at any time between syncs, the recovering file server is able to determine which root to use because of information sent on the primary's last sync. It reads in the correct state information and reconstructs its buffers accordingly. Disk blocks that were used by the primary since the last sync appear to it as free blocks.

The difficult case is when a crash occurs during a sync. To see that the solution works in this case, consider the sequence of actions that take place during a sync. First, all dirty blocks except the root are written to disk, and old blocks are added to the semi-free list. Second, the state information is collected and written to the alternate state area. Third, the in-memory root is written to the alternate on-disk root block. Finally, the sync message is constructed and sent to the backup. It contains the information necessary to update message queues as well as specifying which on-disk state information and root block to use on recovery. Once the sync message has been sent, the semi-free list is added to the free list

[3] Note that a file server manages access to a subtree of the global file system. The current discussion refers to one such subtree and its local root.

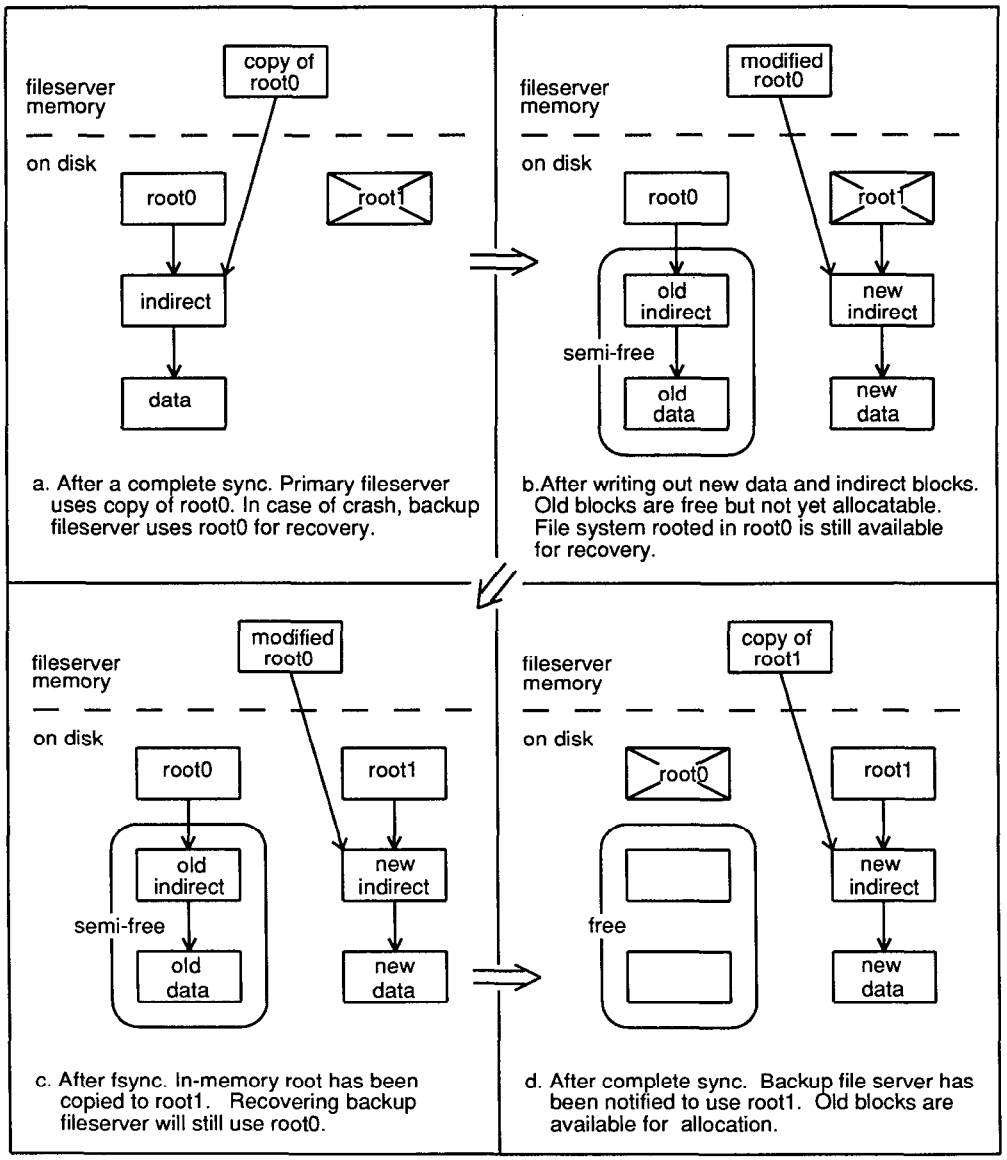

Fig. 2. Syncing the file system.

and the primary continues. Just before the sync message is sent, there are two copies of every modified data and indirect block.

At any time before the sync message is sent, the old consistent state is available. Any time after it is sent, the new state and file system will be used and message queues consistently updated. An additional benefit of this organization is that the file system as a whole is considerably more robust than a standard UNIX-style file system. Even if the entire system is shut down in an uncontrolled way

as the result of multiple faults or operator error, there will always be an entire consistent file system on disk.

5. CRASH DETECTION AND PROCESS RECOVERY

Major changes have been made in our original crash detection, crash handling, and recovery schemes [3]. They are much simpler and more distributed, requiring no interference with processes that are unaffected by the crash.

In this section we describe the crash handling and recovery procedures initiated by the detection of a major fault. Good fault detection is essential to any fault-tolerant system. Our system combines hardware and software diagnosis to detect a wide range of faults. In general, a fault that cannot be recovered from locally causes the entire machine to shut down. While the details of fault detection are beyond the scope of this paper, we are concerned with global detection of a machine crash on the local network, with action taken by the remaining machines to initiate recovery of backup processes, and the rebackup of halfbacks.

5.1 Crash Detection and Handling

A machine is considered to have crashed when it stops communicating with other machines in the system. It can die in two ways. First, if the machine's own diagnostic software or hardware determines that some necessary functionality is lost, it will bring down the machine, and under some circumstances, try to reboot it. Second, the machine can be ordered to die from outside. For example, a special diagnostic process that receives regular reports of transient locally-recoverable errors could decide that the errors are too frequent and will bring the machine down for further diagnosis.

The death of a machine is recognized by the rest of the system. The machines are organized in a virtual ring so that each has a left and right neighbor. Each machine periodically reports its existence to its right neighbor. Each machine expects a regular report from its left neighbor [16]. Should a machine not receive the expected notice from its left neighbor, it will retry communication and, if its efforts fail, it will:

—Determine whether it can communicate with any other machines in the system. If not, it assumes that it must crash, otherwise it assumes that its left neighbor has crashed.[4] Order the uncommunicative machine to die in case it can receive messages and is not aware of its problem.

—Announce the failed machine's demise by broadcasting a *machine-dead* message to all remaining machines.

—Locate a new left neighbor.

Note that except in the case of a two-machine system, a network partition is not possible as the result of a single fault. Communication is via a dual bus so that messages do not go through intermediate machines on route to their

[4] This is tricky in a two-machine system where one machine must decide that it is in control and must prevent the other from accessing peripherals. Info on the shared disk is ultimately used to make this determination. In fact, any time there is a shared disk, the uncommunicative machine must be prevented from modifying the disk.

destinations. Thus the failure of a single machine or a single bus leaves the network intact and leaves all machines with the ability to determine which one is dead.

The handling of a machine-dead message proceeds in two parts: the first assures correct communication with the rest of the system; the second causes backups of crashed processes to be brought up. Immediately upon receipt of a machine-dead message, the receiving machine stops trying to send messages to the dead machine. This is done at a low level by the bus interface before any modification is made to routing table entries. If the dead machine appears as a destination in an outgoing message, that destination is ignored. If it is the only destination, the message is discarded.

The machine-dead message is then placed on the input queue so that backups are brought up only after all other messages on the input queue have been handled. This guarantees that any sync messages that arrived before the machine crashed have been used to update backups before backup recovery is initiated.

5.2 Process Recovery

When the machine-dead message has arrived at the head of the input queue, backups for crashed primaries must be brought up. Before this can be done, primary processes whose backups were lost in the crash are modified: primary quarterbacks are marked *not backed-up*, primary halfbacks are marked *not currently backed-up*. Any channel whose other end is completely lost is made to look as though it had been closed from the other end, allowing processes to deal with the loss of a correspondent.

Actual recovery of a backup process is quite simple. Backups are linked together based on the location of their primary. For each process on the list associated with the crashed machine, the kernel must do the following:

—allocate and initialize structures needed for local kernel state and memory mapping;

—request a list of the pages held by the page server so that memory mapping tables can be correctly initialized;

—set up the kernel stack from the latest sync information; and

—put the process on the run queue.

At this point a backup process is ready to begin execution and can be scheduled independently of the recovery actions for other processes.

The above actions take place independently in each machine that contains backups of processes from the dead machine. Unlike the original scheme [3], message traffic is not disrupted. Destinations in the dead machine are ignored, and messages continue to be delivered to backups even as they are recovered.

5.3 Roll Forward

The period of execution during which a process reexecutes code that was already executed by the primary is called *roll forward*. A user process,[5] in user mode, executes normally during this phase, without knowledge that there has been a

[5] We have already described in Section 4 how this portion of recovery takes place for servers.

crash. In kernel mode there are three ways in which a process during roll forward acts differently from its primary.

When the process attempts to send a message and finds its writes-since-sync count positive, the count is decremented and the message is discarded.

When the process attempts to fork, it checks for the existence of birth notices. If one exists, there are three possibilities. If the primary child process (whose creation generated the birth notice) had synced before the crash, it exists as a backup process. In this case no new process is created, but the process id from the birth notice is used as a return value. The same action is taken if the primary child process lived out its full existence and died before the crash. If the child did not sync and did not die, however, the process id for the new child is retrieved from the birth notice. This guarantees that the new child will correctly acquire ownership of channels and thus messages.

Additionally, a process is not allowed to sync (the first point at which a new backup can be created) until it has completed roll forward. Therefore, the time during which a recovered process must run unbacked-up is at least as long as it takes for it to catch up to the state of its crashed primary.

6. MACHINE REINTEGRATION AND REBACKUP

After a crash, a machine can be repaired and reinstalled as part of the running system. It is necessary to be able to reboot and reintegrate the machine without greatly disrupting running processes. Once the rebooted machine is again part of the system, processes that were running unbacked-up must be able to create new backups for themselves. If a process originally ran as a primary in the rebooted machine, it is able to switch primary execution back to that machine in order to balance the load of primary processes.

6.1 Machine Reintegration

The reintegration of a machine is equivalent to initial boot in most of its phases. The machine begins by executing a first-level boot out of its local read-only memory. This causes the message processor to determine where the machine should get its operating system: if it is a member of the root machine pair and the other member is not booted, it must get its operating system from its disk;[6] otherwise, it is not the root, and must ask the process server to send it an operating system via message.

Once the kernel is booted, the machine notifies the process server that it is up, and the process server in turn notifies all other machines of the existence of a new machine with a *machine-up* message. As a result of this notification, the new machine is integrated into the ring of notifiers and recognizers. Since each machine has a unique id, it has a unique location in the ring. The process server can now assign new processes (heads of family) to the machine. If the machine is being booted initially, this is the end of the reintegration. If it has come up after a crash, process families can be rebacked-up.

6.2 Process Rebackup

When a machine comes back up after having crashed, new backup processes must be created for all halfbacks that lost their primaries or backups during the

[6] This will only be the case on initial boot.

crash. Once this is done, it might be desirable to switch the role of primary and backup in order to rebalance the load of active processes. This is clearly the case in a two-machine system where a crash followed by rebackup will result in all primaries executing on one machine and all backups being maintained on the other.

Process rebackup is initiated when a machine receives a machine-up message. If it contains primary families that expect to be backed-up in the new machine, then each such family is forced to perform a *resync*, which creates the new backup family. After resync, if the primary is found to be running in the original backup machine, the family can *switch sync* in order to reverse the roles of primary and backup.

6.2.1 *Resync.* The most sensitive part of resync is that which ensures that messages are neither duplicated nor missed as the new backup is created. This is particularly true in light of our reluctance to globally interfere with message traffic. A halfback can have channels to processes in locations distributed throughout the system. We do not wish to have to coordinate the correspondent's sends during resync. Therefore, when a machine crashes, routing table destinations to channels owned by halfbacks in the crashed machine are unaffected: messages are constructed complete with that destination; while the machine is down, the low-level bus interface code ignores the destination; when the machine comes back up, messages are sent immediately even if rebackup has not yet occurred. When a new backup is to be recreated, the primary might have messages on its queues which must be resent to the backup. Resending takes place as new messages arrive. Our resync algorithm ensures that the messages are queued in the correct order. It also allows us to use most of the code for an ordinary sync.

Each family needing a new backup resyncs independently of other such families. Within a family, the resyncs of individual processes must be coordinated in order to ensure that shared resources, in particular shared channels, are handled correctly.

A family is nearly tree-structured. As the result of process deaths, it is in fact a collection of subtrees. The root of a subtree is either the head of family or an orphan (a process whose parent has died leaving a gap in the tree structure). Resync begins by forcing the head of family (if alive) and all orphans to resync. Each resyncing process will do the following:

(1) Reconstruct either a walking exec message (if it is the head of a family) or a birth notice for itself and send it to the backup machine.
(2) Force all child processes to resync and wait until they have finished.
(3) Perform a normal sync. The arrival of the sync message causes the recreation of all necessary backup routing table entries.
(4) Wait for the rest of the family to finish syncing, or, if last in the family to finish syncing, notify the others and the backup machine. This involves sending a single *notify* message destined for both the local machine and the backup machine.
(5) Send to the backup machine copies of all messages currently linked to the process's routing table queues that arrived before the notify message. Messages on shared channels are resent only once.

The notify message plays a crucial role in the coordination of message traffic. Until it arrives at the backup machine, any messages arriving for the newly created channels are discarded. After its arrival they are delivered normally; all channels have been created. The notify message contains the current arrival number of the primary machine. The backup machine increases its own arrival number if necessary to ensure that all newly arriving messages are given a number higher than that in the notify message. All newly arriving messages in the primary machine will also be given a higher number. Resent messages (step 5 above), are specially typed. On arrival, they retain their original arrival numbers and are queued in arrival number order so that they are positioned correctly on the queue prior to new messages. This guarantees that the primaries and backups will have exactly the same messages on their queues when the primary resumes normal execution.

6.2.2 *Switch Sync.* Like resync, switch sync takes place family-by-family, but it is simpler because the content of message queues is already consistent across machines.

The head of family and orphans are forced to sync. This sync is special only in that it forces all children to sync before the parent completes, rather than just those children without backups. Each process, after doing the sync, turns itself into a backup process. This involves little more than freeing such things as its incore pages and kernel data structures, and adjusting its kernel stack. The last process in the family to switch sync sends a message to the backup machine, telling it to recover the backup family in the normal manner.

7. SURVIVING SOFT ERRORS

An additional benefit of our fault-tolerant design is that the system can recover from a large class of kernel software faults. Any error condition that is related to a local kernel's environment will probably not recur after recovery. The violation of boundary conditions and the exhaustion of resources such as buffer pools occur because of conditions specific to one kernel. When resulting inconsistencies are detected, they cause the affected kernel to stop. Its machine is declared dead by the rest of the network, and backups for its processes recover elsewhere. Since the other kernel's environments are all different, the same situation is unlikely to recur. In the late phases of our development effort, a number of kernel bugs were found as the result of crashes of individual machines which left the rest of the system functioning.

8. PERFORMANCE

Relative to a standard UNIX system, our performance is affected by the distributed message-based architecture and by the overhead for fault tolerance. We consider the two separately.

8.1 Distribution

On a single-machine system running without backups, message-based communication with server processes introduces operating system overhead for message management as well as additional context switches between user and server.

Measured against a standard UNIX system based on the same hardware, the average performance degradation for the Byte and AIM II benchmarks is about 10 percent. The range for individual system calls is 3 percent to 20 percent. The system calls most affected are those with the least functionality, such as time; those least affected are system calls which read and write data with a standard block size.

Message-based communication on a two-machine system introduces the additional time required to access the system bus and copy messages to and from memory when the user and server run on different machines. The average degradation for a single program, again compared with single-machine standard UNIX, increases to 15 percent.

Though the performance of a single program degrades when run on a two-machine system, the system can handle a substantially greater load. Adding a second machine to the bus allows the load to be increased by 70 percent before it begins to affect response time. Each additional machine also increases the overall capacity by approximately 70 percent of a single machine's performance. So, with three machines the capacity is about 2.4 times that of a single machine.

8.2 Fault Tolerance

Overhead for fault tolerance is best measured by comparing two versions of our system, one running with backups and one running without backups.

On a two-machine system, the only fault tolerance related factor affecting the performance of an individual process is the time required to generate a sync message. There is no additional cost for generating messages with multiple destinations. In such a system, there is no increase in message traffic as the result of fault tolerance. In larger systems, there is some increased contention for the local bus because of the potential need to simultaneously connect with multiple receivers to send a single message.

The capacity of the system as a whole is potentially affected by

—memory requirements for backup information,

—the ability of the paging device to keep up with process synchronizations,

—CPU requirements for managing backup message copies and sync messages.

These three factors result in a 10 percent reduction in performance.

Memory requirements for backup information and the traffic on the paging device are closely related. The longer one waits between synchronizations, the more space is used for queues of backup messages. On the other hand, frequent synchronizations add overhead to individual primary processes and require work of the pager. In our system, the optimum sync frequency is every 64 messages. This guarantees that many short-lived processes never sync. A process that never syncs never has a backup page account and uses no additional paging services. Based on average message size, the resulting space requirement is approximately 25 Kbytes of main memory per backup.

If processes synchronize too frequently, a single page server can become overloaded. The system must be configured to include the appropriate number of page disks and corresponding page servers. Synchronizing every 64 messages, a single page server is able to service up to 50 users.

Since one of the three processors in each machine handles all message delivery, including queuing of backup copies and processing sync messages, there is very little interference with user processes executing on the other two processors.

8.3 Overall Performance

Summarizing the above numbers we see that:

—Distributed message-based system organization reduces performance by 15 percent compared with a standard UNIX machine.

—The additional second machine increases performance of the distributed system by 70 percent.

—Fault tolerance reduces this performance by 10 percent.

Finally, if the benchmarks are run on a fault-tolerant two-machine system, the performance turns out to be 1.6 times that of a standard UNIX.

One final figure is of interest. It is important that recovery time in case of a crash be acceptable to a user. Our experiments have shown that the delay experienced by a user whose primary process dies is 5–15 seconds. Unless servers are affected by the crash, user processes whose primaries are not lost are not affected during recovery.

9. POSSIBILITIES: LIMITATIONS AND EXTENSIONS

In this section we consider the possibilities of using the ideas presented above in other contexts as well as possible extensions to the existing system.

9.1 Networks

One obvious question is whether this scheme could be made to work in a more general environment, for example, in an arbitrary network of machines without a specialized bus to guarantee atomicity of multiple message delivery. There are two drawbacks. First, the atomicity property is necessary and would probably be difficult to implement. Implementation using a software protocol would adversely affect performance since all traffic between two backed-up processes must necessarily go out over the net regardless of the location of the processes.

9.2 Fullbacks and Process Migration

The implications of adding fullbacks to Targon's repertoire are intriguing. Recall that a fullback family is to be backed-up again as soon as possible after crash on any available machine. The implementation is not difficult, but does involve sending notification to all machines that the family's backup machine has changed. The receiving machine would then change the destination in appropriate routing table entries. This would be done prior to creating the backup so that all messages are guaranteed to be sent to both primary and new backup before a resync is done.

With fullbacks in place, process migration is nearly trivial, though whole families must be moved. All that is needed is the ability to explicitly destroy the family's current backup. Then, to move a family from its current location the

following would be necessary:

—destroy the family's backup,
—rebackup the family in the desired location, and
—do a switch sync to cause primary execution to take place.

The only drawback is that the family will not be backed-up for a short period. Careful design could probably work around this.

10. CONCLUSION

This paper has reviewed and updated the design and implementation of a fault-tolerant operating system. The system uses message-based communication and inactive backup processes to ensure that processes survive hardware failures. Fault-tolerant operation is automatic and transparent to the user. This, together with UNIX compatibility, allows existing software to be run fault-tolerantly without modification. The updated design includes a complete reworking of server fault tolerance, which is much more consistent with that of normal user processes than was the original plan. We have described in detail the file server implementation. Also new is a simpler and more efficient crash and recovery management in which unaffected processes are not penalized during recovery.

The system works. It is currently in use in production environments and runs with very acceptable performance in a variety of configurations.

REFERENCES

1. ARNOW, D., AND GLAZER, S. A fast safe file system for UNIX. Unpublished paper written in 1984 for Auragen Systems Corp., Ft. Lee, N.J.
2. BARTLETT, J. A nonstop kernel. In *Proceedings of the Eighth Symposium on Operating Systems Principles* (Asilomar, Calif., Dec. 1981). ACM, New York, 1981.
3. BORG, A., BAUMBACH, J., AND GLAZER, S. A message system supporting fault tolerance. In *Ninth Symposium on Operating Systems Principles* (Breton Woods, N.H., Oct. 1983). ACM, New York, 1983.
4. GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The recovery manager of the system R database manager. *ACM Comput. Surv. 13*, 2 (June 1981), 223–242.
5. KIM, W. Highly available systems for database applications. *ACM Comput. Surv. 16*, 1 (June 1984), 71–98.
6. LISKOV, B., AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst. 5*, 3 (July 1983), 381–404.
7. LISKOV, B., AND LADIN, R. Highly-available distributed services and fault-tolerant distributed garbage collection. Programming Methodology Group Memo 48, MIT Laboratory for Computer Science, May, 1986.
8. POWELL, M., AND MILLER, B. Process migration in DEMOS/MP. In *Proceedings of the Ninth Symposium on Operating Systems Principles* (Breton Woods, N.H., Oct. 1983). ACM, New York, 1983.
9. POWELL, M., AND PRESOTTO, D. PUBLISHING: A reliable broadcast communication mechanism. In *Proceedings of the Ninth Symposium on Operating Systems Principles* (Breton Woods, N.H., Oct. 1983). ACM, New York, 1983.
10. RASHID, R., AND ROBERTSON, G. Accent: A communication-oriented network operating system kernel. Tech. Rep. CMU-CS-81-123, Dept. of Computer Science, Carnegie-Mellon Univ., Apr. 1981.
11. SCHNEIDER, F. B. Byzantine generals in Action: Implementing fail-stop processors. *ACM Trans. Comput. Syst. 2*, 2 (May 1984), 145–154.

12. *Stratus/32, VOS Reference Manual.* Stratus Computers, Inc., Marlborough, Mass., 1982.
13. STROM, R. E., AND YEMINI, S. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst. 3,* 3 (Aug. 1985), 204–226.
14. TOLERANT SYSTEMS, INC. Eternity series: Technology brief. Internal publication, July 1988, Tolerant Systems, San Jose, Calif.
15. VERHOFSTAD, J. Recovery techniques for database systems. *ACM Comput. Surv. 10,* 2 (June 1978), 167–196.
16. WALTER, B. A robust and efficient protocol for checking the availability of remote sites. In *Proceedings of the Sixth Workshop on Distributed Data Management and Computer Networks,* (Berkeley, Calif., Feb. 1982), pp. 45–68.

Received May 1987; revised October 1988; accepted October 1988