A Message System Supporting Fault Tolerance

Anita Borg, Jim Baumbach, Sam Glazer

Auragen Systems Corporation
2 Executive Drive
Fort Lee, New Jersey
07024

Abstract

A simple and general design uses message-based communication to provide software tolerance of single-point hardware failures. By delivering all interprocess messages to inactive backups for both the sender and the destination, both backups are kept in a state in which they can take over for their primaries.

An implementation for the Auragen 4000 series of M68000-based systems is described. The operating system, Auros™, is a distributed version of UNIX*. Major goals have been transparency of fault tolerance and efficient execution in the absence of failure.

1. Introduction

This paper describes the design and implementation of message-based interprocess communication to support fault tolerant computing in an on-line transaction processing environment. The system assures that all executing processes will survive any single hardware failure. The scheme works efficiently and automatically; little processing overhead is incurred and no programmer or user awareness is required for fault tolerant operation. A simple and general design is presented in the first half of the paper. After that, we describe the details of our implementation which is embedded in a distributed version of UNIX running on the Auragen 4000 computer.

Section 2 reviews some existing methods for implementing fault tolerance. Section 3 describes the goals and scope of our work. In Sections 4, 5, and 6, we introduce the design and algorithms on which the Auragen message system implementation is based. Section 7 describes our implementation, first summarizing the hardware design and then detailing the software supporting fault tolerance. In Section 8, we review the implementation with an eye toward prediction of the efficiency of the system.

2. Background

Though all fault tolerant systems require the duplication of hardware and software resources, they differ in the following areas:

1. The kinds of faults which are tolerated;
2. The manner in which duplicate resources are used to provide fault tolerance;
3. The extent to which additional hardware can be used to increase computing power in the absence of failures;
4. The amount of programmer knowledge required to write and run fault tolerant programs.

There is currently much interest in the analysis and detection of and recovery from software failure. However, the complexity and cost of implementing available solutions has limited their practicality. We are concerned with systems which attempt to guarantee survival in the presence hardware failures and to limit the effects of software errors.

Existing fault tolerant systems are similar in that all require that every **primary** process (executing program) have one or more **backup** processes (which may or may not be executing) capable of continuing execution if the primary fails.

In some systems, duplicate hardware is dedicated for backups. A process and its backups execute simultaneously on tightly coupled processor's and are essentially indistinguishable. If one processor fails, the others continue without interruption. Though recovery in case of a crash is instantaneous, the duplicate hardware provides no increased computational capability. This scheme is very costly for applications which do not re-

quire real-time response in the face of failure (6).

A second approach involves keeping an inactive (xecuting) backup process which, upon failure of the primary's CPU, can be brought up on another CPU to take over for its lost primary. The state of the backup as represented by the universe of values in its data space must be identical to that of the primary, or be capable of becoming identical. In the absence of failure, the duplicate hardware can be used to run additional primary processes.

Designs using inactive backups differ in the way in which and efficiency with which backup processes are maintained. One strategy is to explicitly checkpoint, i.e., to copy the data space of the primary to that of the backup, whenever the former changes. (1) Though the backup is inactive so that the extra CPU can be used for other purposes, the frequent copying of the primary's data space slows down the primary process and uses up a large portion of the added computing power. In addition, checkpointing is not an automatic action taken by the operating system, but must be called by a program, requiring either preprocessing or programmer knowledge of the requirements of fault tolerance.

The design described below also uses inactive backups. However, explicit checkpointing is replaced with a message-based strategy which is both automatic and efficient.

3. Goals

The goals of our research have been to design and implement an operating system mechanism which provides efficient fault tolerant operation suitable for use in an on-line transaction processing environment. Ease of use is a high priority goal in our overall design. As a result, a primary concern is that fault tolerant operation be entirely transparent to users of the system.

3.1 Fault Tolerance

We define a fault tolerant system as one which assures that all processes will survive a single hardware failure.

In our model, hardware consists of two or more **processing units**. A processing unit is a traditional computer, consisting of processor(s), memory, and optional peripheral devices. Processing units communicate via message over some communication medium (e.g., bus or network).

Each processing unit is assumed to run its own independent copy of the operating system kernel in order to support the execution of processes. 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.

The operating system kernel is presumed to be free of errors. Our design does not attempt to provide complete software fault tolerance, but provide software which will tolerate, i.e., quickly and automatically recover from, hardware failures.

A **failure** is an algorithmic or mechanical hardware malfunction which makes impossible the continued execution of a process (e.g., failure in an isolatable portion of memory) or all processes (e.g., failure of the processor itself) in one processing unit. Multiple hardware failures which cause more than on processing unit to become inoperable are not handled.

3.2 Efficiency

Our design presumes an on-line transaction processing environment, rather than real time process control. In addition, we assume that failures are infrequent. Therefore, we are concerned with efficiency during normal execution, but are willing to tolerate some inefficiency during recovery from a failure. The system should be efficient in the sense that it maximizes the productive use of hardware during normal execution, i.e., in the absence of failure. A solution which requires the dedication of substantial system resources solely for the support of fault tolerance is therefore unacceptable.

3.3 Transparency

We require that fault tolerance be transparent to both the programmer and the user. That is, no special programming should be required for a program to run fault tolerantly. Failure detection and recovery must be automatic. User programs should be completely unaware of the failure and a user at a terminal should notice at most a short delay during recovery.

4. The Auragen Approach

We distinguish two types of processes. **User processes** communicate (perform all input and output) via message and have no direct access to peripherals. Actual device IO is performed as the result of requests sent to **peripheral server processes**. Peripheral servers are associated with specific devices which they access via special system calls not available to user processes. They execute only in processing units with peripherals and handle all user requests for real IO.

This section and the next describe the algorithms and mechanisms used to provide fault tolerance for user processes. Peripheral server processes require a slightly different treatment which is discussed in section 7.9.

Our approach is based upon the following requirement of determinism in user processes: If two processes start out in the identical state, and receive identical input, they will perform identically and thus produce identical output.

Each primary process has an inactive backup resident in a processing unit other than that of the primary. A backup process is kept nearly up-to-date and is provided with all information necessary to bring itself up to the state of the primary, and continue execution as the new primary, should there be a failure.

Thus, if a primary and its backup are initially identical, and, if all input (messages) to the primary process is also made available to the backup, then on failure of the primary, the backup can catch up by recomputing based on the same messages which were used by the primary.

In order to avoid complete recomputation by the backup upon failure, a primary process and its backup can be periodically synchronized. In the intervening periods, when the backup is not identical to the primary, all messages to the primary are kept available for the backup. Upon synchronization, all messages previously read by the primary may be discarded. If the primary fails, the backup executes (rolls forward) from the point of last synchronization using the saved input.

5. The Message System

A message system embedded in the operating system kernel is used to support the above algorithms. It provides and controls interprocess communication including the transfer and routing of messages. It initiates the creation and deletion of backup processes and controls the periodic synchronization of primary and backup. It assures that the backup has the necessary information to take over in case of failure, and that it will interact correctly with the rest of the system.

The message system must assure that:

1. During normal execution, all messages sent to the primary which were unread or arrived since the point of last synchronization, are available to the backup.
2. The primary's state as of last synchronization is accessible to the kernel controlling the backup's processing unit.
3. The backup process, during recovery, reads the available messages in exactly the same order as did the primary.
4. Also during recovery, the backup will not resend any messages already sent by the primary.

The following sections describe how this is accomplished.

5.1 Multi-way Message Transmission

Every message which is sent from one primary process to another is actually sent to three destinations:

1. The requested primary destination process;
2. The backup of the primary destination process;
3. The backup of the sending process.

The underlying hardware or software must guarantee the atomicity of the multiple delivery of a message destined for more than one location.

That is:

1. Either all three destinations receive the message, or none receive it.
2. The arrival of the message to its three destinations is never interleaved with that of any other message, assuring that a primary and its backup always receive messages in the same order. In other words, if two messages are sent, one will reach all of 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.

The message is used in a different way at each of the three destinations:

1. At the primary destination, the message is queued up for reading by the primary destination process;
2. At the backup destination, it is queued up and saved for the backup of the destination process, to be read only upon rollforward after a failure;
3. And, at the sender's backup, a count of messages sent since synchronization is incremented and the message is discarded.

Thus, every backup process has a queue of messages which have been sent to its primary, and a count of all messages which have been sent by its primary. In addition, the primary keeps a count of the number of messages it has read since the last synchronization.

5.2 State Saving During Synchronization

During normal operation a primary and its backup are automatically synchronized whenever the primary has read more than a system-defined number of messages (allowing that many of the backup's messages to be discarded), or, if it has executed for more than a system-defined amount of time since last synchronization.

Any changes in the address space of the primary since last synchronization are stored so they are available for the backup in case of failure. This is accomplished by cooperation between the message system and the system's paging mechanism, which is detailed in Section 7.6.

Next, a **sync message** containing a small amount of state information is sent directly to the kernel in the backup's processing unit. This includes the count of the number of reads done by the primary since the last synchronization. The availability of this count allows any messages saved for the backup, but already read by the primary, to be discarded. After synchronization the backup will have the correct set of messages available.

The arrival of a sync message also causes the backup's count of messages sent since sync by the primary to be zeroed.

5.3 Message Ordering

It has been assumed that all messages arrive at the primary's and backup's processing units in the same order, and are queued up in FIFO order. It is not necessary that processes read their messages in FIFO order as long as the mechanism used to decide the ordering is deterministic. Auragen's implementation, which involves multiple input queues, each of which is accessed in FIFO order, is described in Section 7.4.

5.4 Avoiding Redundant Messages

One remaining concern is that the backup not resend any messages which were generated by the primary between the last synchronization and failure. Recall that the third message destination is the sender's backup, where the message is counted and discarded. Every time the backup, which has become the new primary, begins to execute code to send a message it checks the value of the count. If the count is positive (this message was already sent by the primary), it is decremented and the message is not sent; if the count is zero the message is sent.

6. Handling a Failure

When a failure affecting an individual process occurs, the kernel in the processing unit containing the process's backup is notified and makes the backup runnable. This includes notification of all of the process's correspondents. If an entire processing unit crashes, every other unit is notified. Each kernel then makes active all backups whose primaries were executing in the crashed unit.

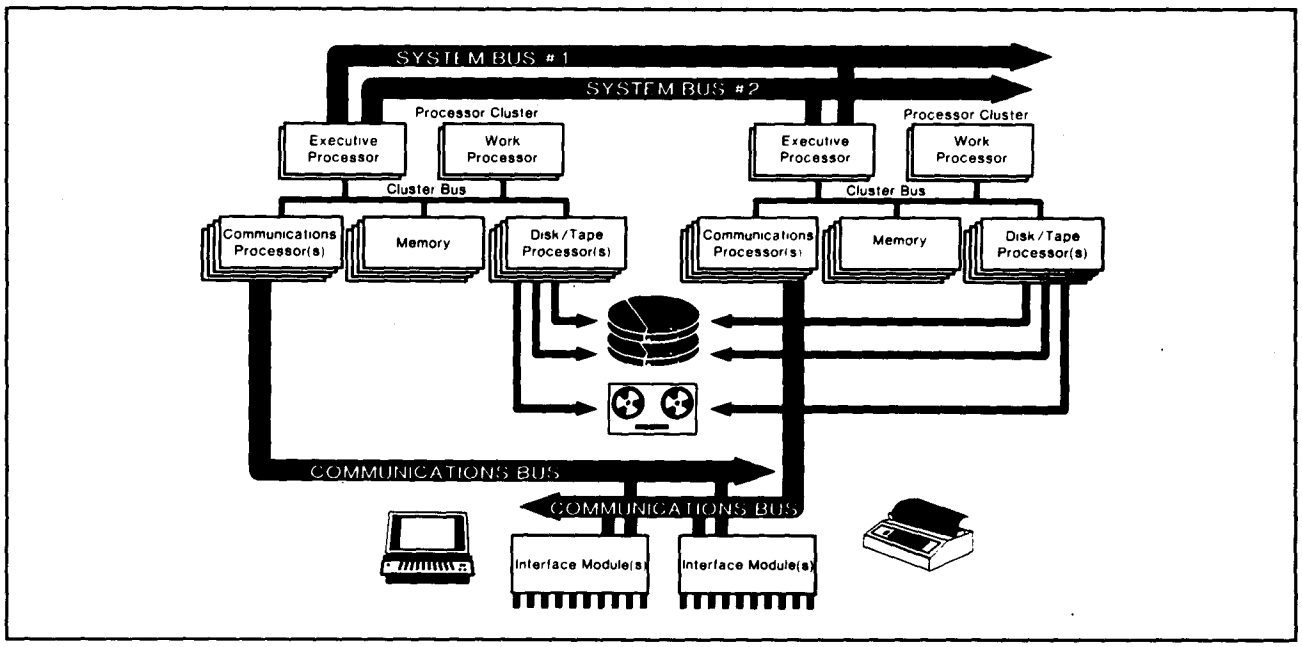
Each backup (after optionally creating a new backup process) begins executing in the state that the failed primary had achieved at the time of last synchronization. It has exactly the right messages available, is assured of reading them in the correct order, and has available the address space of the primary at the time of last synchronization via its page account.

7. Implementation

This section describes an implementation of the message system on the Auragen 4000 series of computers. The implementation project is ongoing.

7.1 Hardware

Auragen's basic processing unit is called a **cluster**. The Auragen 4000 consists of 2 to 32 clusters connected by a dual high-speed intercluster bus. Each cluster contains 3 to 7 Motorola M68000s and a large shared memory. Two of the processors in each cluster run user and system server processes. These are referred to as **work processors**. They execute either memory-mapped in user mode (with demand paging) or unmapped in kernel mode. A third processor, known as the **executive processor**, is connected to the cluster's memory and to the intercluster bus. It controls all intercluster message traffic. The remaining processors control peripherals and communication ports. All peripherals are dual-ported and connected to two clusters. In addition, disks are connected in pairs to facilitate mirrored files. It is possible for a cluster to have no peripherals.



7.2 The Operating System Kernel

The Auragen operating system, Auros™, is derived from and is compatible with Bell UNIX* System III. A copy of the Auros™ kernel resides in each cluster. The function of the Auros™ kernel is different from that of the standard UNIX kernel. It is limited in that it performs only cluster local operating system functions such as scheduling runnable processes, memory management, control of local peripherals, and message handling. It does not handle global resource control. Some portions of the kernel, in particular, those responsible for message transmission and delivery, execute only on the executive processor; others are executed directly by user processes on the work processors as the result of system calls.

The kernel is backed up only in the sense that an independent copy, capable of running processes, exists in each cluster. The kernels are not synchronized. This means that the identical kernel state will not be available to a backup process and thus must not be required. As a result, user processes synchronize to an identical state in user mode, never in kernel mode.

Operating system functions which must be globally available and consistent have been removed from the kernel to server processes. For example, file system control is the duty of file server processes, while page management is handled by one or more global page server processes.

7.3 Backup Modes

The kernel allows processes to be backed up in three ways depending on when and whether a new backup process is created after a crash occurs.

1. **Quarterbacks** run backed up until a crash occurs, but no new backup is created for them after a crash. Most relatively short-lived user programs will run in this mode.
2. **Halfbacks** have new backups created only when the cluster in which the original primary ran is returned to service. Peripheral servers are backed up in this way because their primary and backup must be located in the two clusters connected to the device they control.
3. **Fullbacks** have new backups created before the new primary begins executing. A system must consist of at least three clusters for fullbacks exist.

The backup mode can be specified by the user. The default mode, at least for the first implementation, will be quarterback.

7.4 Message System and Interprocess Communication

In Auros™, the message system is contained in and fully integrated with the kernel. It is the basis for process fault tolerance and controls all interprocess communication.

7.4.1 Channels

Processes send and receive messages via **channels.** An entry in a cluster-local table, the **routing table**, defines one end of a channel. A cluster's routing table resides in kernel space in main memory and is maintained by message system code executing either in the work or executive processors. A routing table entry contains:

1. All information necessary to route a message to the primary destination and to the backups of both the destination and the sender.
2. A queue for holding incoming messages.
3. Status information including the way in which the communicating processes are backed up and the type of process at the other end (system server or another user).

A channel between two backed up processes consists of four routing table entries, one for each primary and one for each backup.

A channel is opened and routing table entries are made as the result of UNIX **open** system calls executed by two processes wishing to communicate. Each **open** returns an integer file descriptor used to reference the channel in subsequent **read** and **write** system calls. We retain the the term 'file descriptor' and its abbreviation, fd, from UNIX, though channels do not necessarily represent files.

An **open** causes an **open request** message to be sent on a preexisting channel to a file server. If the name to be opened represents a file, the file is opened and an **open reply** is sent to the opener and its backup. If the name is a channel name, the file server pairs up openers to the same name and sends open replies back to the openers and to their backups. The arrival of an open reply at a backup cluster causes the creation of the backup routing table entry.

7.4.2 Message Sending and Delivery

The following operations occur when a message is written on an open channel:

1. The user process, in kernel mode, constructs a message containing routing information from the local routing table channel entry together with the message contents provided by the user program. The message is placed on the cluster's outgoing queue. The user process then waits if an answer is required (e.g., if the message was a file server request) or returns from the system call.
2. The executive processor, finding a message on the outgoing queue, transmits the message only once over the intercluster bus.
3. The transmission is picked up by all clusters

whose address appears in the message's routing information. The hardware guarantees that either all or none of the target clusters receive the message. Since a cluster may transmit or receive only one message at a time, messages are never interleaved.

4. Upon successful transmission to other clusters, the message is delivered to any local destination (there can be at most one) or is discarded.

Message delivery at the receiving end is also handled by the executive processor. Message protocol allows the processor to determine whether the message is for the primary destination, its backup, or the sender's backup.

1. If it is for the primary destination, the message is enqueued on the channel's routing table entry, and, any process awaiting a message on the channel awakened.
2. If it is for the destination's backup, the message is enqueued but no process is awakened.
3. If it is for the sender's backup, a count of 'writes since synchronization' is incremented in the routing table entry, and the message is discarded.

7.5 Determinism of User Processes

User processes are required to be deterministic. After a crash, the backup must roll forward, recomputing based on the same input provided to the primary which failed. However, the kernels in various clusters are not synchronized. Local information, such as cluster's local time, a process's priority at a particular point in its execution, or the number of pages it has in memory, may differ depending on the cluster in which it executes. A user process and its backup must be insulated from such differences. Regardless of differences between clusters, every interaction between the kernel and a backup after crash must must appear to the backup the same as it did to the primary.

We must consider the two ways in which user processes interact with the kernel: Synchronously via system call, and asynchronously as the result of a system call or signal.

7.5.1 Synchronous System Calls

Synchronous system calls fall into two categories. Some calls request local information concerning a process's state or identity, e.g., **getuid**, **getpid**. The user id **getuid** is cluster independent. However, the process id in standard UNIX* is an index into a local process table and is environmental. We have made the process id into 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 get information via message. Since the same messages are available to the backup, they are assured to return the same answer. Our system currently constrains all **reads** to be synchronous. If a message is not available when a **read** is executed the process must await a message. It cannot return with 'no message found', because the backup on rollforward may not find its message queue in the same state. This does not preclude a server from aborting a read on its device, as long as an error response is be sent to both the primary reader and its backup.

Two new system calls allow a process to group channels together (**bunch**) and then to await arrival of the first message on any channel in the group (**which**). Messages are given sequence numbers on arrival at a cluster so that the behavior of **which** can be replicated by the backup.

The **write** system call on a user-to-user channel can return as soon as the message has been placed on the cluster's outgoing queue. But, **writes** which require an answer from a server (e.g., a write to a file which may fail) cannot return until that answer arrives.

The **time** system call would return environmental information if handled as in standard UNIX. We have made that call the responsibility of the process server (see below) rather than the local kernel. **Time** sends a request via message, and receives its answer via message. The backup will have the same response available.

7.5.2 Asynchronous Interaction

The only truly asynchronous system call is **alarm**, which requests that an alarm signal be generated after a particular amount of real time. This is handled in the same way as other asynchronous signals such as those resulting from typing a control C at a terminal.

All asynchronous signals are sent via message and are queued up on a process's **signal channel**. This does not include synchronous signals such as those generated by a zero divide because they are sure to occur identically for the backup. Any signal which is ignored is removed from the queue and is counted as a 'read since sync.' Any asynchronous signal which is not ignored causes the process to sync just prior to handling the signal. If a crash occurs, the backup will find the signal and handle it immediately, at the same place as did the primary.

7.6 Operating System Server Processes

Recall that operating system services which must be both globally available and backed up cannot be provided by the unsynchronized and independent Auros™ kernels. Such services are provided by server processes which come in two varieties, system servers and peripheral servers.

System servers are processes which keep track of global system resources via tables in their address space. They are backed up, communicate via message, and execute in the same way as ordinary user processes. System servers are special in that they are started only by the operating system. When efficiency is essential, a server's address space is locked into memory to avoid page fault delays. One such server, the **process server**, keeps track of the location of all processes in the system. It periodically receives reports from each kernel and services requests for system status information.

Peripheral servers are associated with logical or physical devices. They receive messages via normal backed up channels requesting them to perform operations on the device. However, they are able to execute special system calls which control the associated device. The server must be in one of the clusters connected to the device, and its backup must be in the other. A peripheral server's address space usually resides permanently in memory, though only in those clusters having the corresponding peripheral. This does not greatly increase the amount of space taken up by the system; in UNIX* all such functions, as part of the kernel, reside in main memory.

There is a **tty server** in each cluster having terminals.

Three types of peripheral servers are associated with disks.

A **file server** is associated with each **file system**. Auros™ file systems are logically the same as UNIX file systems, i.e., they are created and accessed via the same system calls, but are internally structured differently to allow the file server to sync correctly. When a user process opens a file, it gets a channel to the appropriate file server, which at the server's end is associated with the file. Processes always have at least three channels to file servers: One for the text file currently being executed, and one each for the process's root and current directories.

A **raw server** is associated with each disk to handle requests for direct access rather than via a file system.

A **page server** is associated with disk space used to hold the modified pages of a user's address space which have been paged out. A page fault causes a message requesting the page to be sent to the page server if it was previously paged out, or to a file server if the page is a read only text page. When a modified page must be swapped out it is sent to the page server. The page server keeps one account for a primary process, and another for its backup. The backup's account contains all modified pages in their state as of last synchronization. The page server itself must permanently reside in memory.

7.7 Creation of Backup Processes

A backup process consists of a process control block (PCB) corresponding to the combined UNIX user and process structures, less the kernel stack, and a backup page account kept by the page server. Backups for most system servers and peripheral servers are created when the primary comes into existence. This is not true for all user processes. Their backups are created only when absolutely necessary to assure fault tolerant operation.

The processes in a cluster can be divided into families. All members of a family have a common ancestor process, the **head of family**. There is no single ancestor process for the entire cluster. All members of a family must have their backups in a single cluster. Backups for heads of families are created when the primary is created. However, a backup is not automatically created when a new child process is forked.

Upon fork, a birth notice message is sent to the cluster of the forking process's backup. A birth notice causes routing table entries to be made for channels which are created on fork; they must be there to receive backup copies of messages sent to the primary. Channels inherited from the parent process already have backup routing table entries. In case of crash, the birth notice is used during repetition of the fork to give the new child the same process id as its primary.

The birth notice does not contain complete state information and does not cause the creation of a backup process. Only the following events trigger the creation of backups for new processes.

1. If a process has executed long enough for a sync to be required, the first sync (sent to the cluster of the parent's backup) causes the backup to be created.
2. Whenever a process performs a sync, it must force any children which do not yet have backups to sync and create backups. This assures that their page accounts will be created correctly.

In many cases, short lived processes will not have to have a backup process or a backup page account.

7.8 Synchronization of User Processes

Recall that the synchronization of a user process and its backup is automatically initiated by the kernel. Whenever a process's count of reads or execution time since last synchronization exceeds a preset amount, the process is forced to perform a **sync** operation. It is possible to set the message count and execution time interval which trigger sync for each process.

The sync operation (at the primary's end) takes place in two parts. First, the normal paging mechanism is used to send all pages which have been modified since last sync to the page server. 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 page server sees no difference between these pages and any other it receives. It simply adds them to the primary's page account.

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

1. All cluster-independent information kept about the process's state. For example, the virtual address of the next instruction to be executed, accounting information, current values in registers, etc.
2. Channel information for any channel which has changed since last sync, i.e. opened, written to, read from. If the channel has been read from, the number of reads since sync is sent.

The sync message is sent to the cluster 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. If it crashes before the message leaves the cluster, the backup will take over from an earlier point. And, because messages leave the cluster in the order in which they are placed on the outgoing queue, any subsequent message sent by the primary will not reach its backup and be counted until after the sync message has been processed.

The page server's response to the sync message is to make the backup's account identical to that of the primary. After a sync, only one copy of each page will exist. The accounts will start to differ only when when new pages are received from the primary. Then, two copies will be kept only of those pages which have been modified since sync.

When the sync message arrives at the backup cluster, the executive processor uses the contents of the message to update the backup's state and channel information. For each channel represented in the message:

1. If the channel is new, the routing table entry (created as the result of an open reply message) is located and associated with the correct fd;
2. If the channel has closed, its routing table entry is removed;
3. If the count of reads since sync is positive, that many messages are removed from the associated message queue;
4. The writes-since-sync count kept in the backup routing table entry is zeroed.

Recall that either all or none of the destinations get the sync message. Therefore, the page account will not be updated unless the backup definitely is brought up to the state of the primary.

At the completion of the above operations, the backup's state is the same as that of the primary at the time it issued the sync (though the primary may 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.

7.9 Synchronization of Peripheral Servers

Synchronization of peripheral servers is different from that of other processes for a number of reasons.

First, peripheral servers must be core resident rather than paged. The page server cannot demand page its own tables and the file server cannot demand page its own text. The tty server cannot wait for a page before reading incoming characters. This means that a copy of the server's address space will not be available from the page server should the primary crash.

Second, the peripheral servers communicate with devices directly rather than via message. As a result, the requests for action given to the driver, and the answers it produces, will not be available in the backup cluster. Only the original requests from user processes will find there way there.

Our solution uses active backup processes with memory-resident address space for peripheral servers. The primary server repeatedly reads, services, and responds to users' requests, and periodically sends a state information to its backup. The backup server awaits synchronization messages from its primary. On arrival of a message, the backup uses the information to update its internal state and to discard any messages requesting services already rendered by the primary.

Since the server processes are part of the operating system, and perform their syncs explicitly in user mode, each can be written to send only that information which is actually needed to update the internal tables of the backup.

File server syncs are a case in point. A file server's cache of disk buffers is kept in its address space. The cache must periodically be written out to disk. Once written out to a dual ported disk, a substantial portion of the server's address space is available to its backup. If a sync is done at the same time, we avoid sending a large amount of information to the backup via the message system. The primary sends only information concerning the state of various pending requests and the numbers of requests it has handled since last sync allowing those to be removed from backup message queues.

This method involves a reorganization of the file system on disk. An old copy, i.e., in the state as of last sync, cannot be destroyed until the sync is complete, in case a crash occurs during the operation. This involves the duplication on disk of those blocks which have changed since last sync. An additional effect of such a reorganization, is to make the file system considerably more robust than is that in UNIX*.

7.10 Crash Handling and Recovery

Earlier, we noted that a hardware failure in a cluster could affect either the whole cluster or only some of the processes executing in the cluster. Our initial implementation has dealt only with the former case. That is, if a failure makes it impossible to continue running some of the processes in a cluster, then the entire cluster is brought down.

Local failure detection and diagnosis are done in each cluster. Its details are beyond the scope of this paper. Periodic polling of every cluster will discover the shutdown and notify the remaining clusters to begin crash handling.

7.10.1 Crash Handling

As soon as a cluster finds out that there has been a cluster crash in the system, the transmission of outgoing messages is disabled and two very high priority crash handling processes are scheduled. At most a few messages will accumulate on the outgoing queue before these processes begin running, one per work processors. These processes begin actual crash handling only after all messages have been distributed which arrived prior to notification of the crash. This assures that before any backup is brought up, the latest sync message from its primary has been processed.

The two synchronized crash processes perform the following operations:

1. The routing table is searched for references to the crashed cluster. If the primary destination has crashed, it is replaced by the backup destination. If the destination process is a fullback, the channel is marked unusable until notification arrives of the creation and location of the new backup.
2. Backups for halfbacks and quarterbacks which crashed are made runnable.
3. Fullbacks which are no longer backed up are located and linked for backup creation.
4. The outgoing queue is examined for destinations in the crashed cluster. They are adjusted like those in the routing table. Those whose destination processes are fullbacks are held until the location of the destination's new backup is known.
5. Backups of peripheral servers are signaled to begin recovery.

7.10.2 Recovery

System servers and peripheral servers must recover quickly:

1. Processes with pending requests, which are not otherwise affected by the crash should not be inordinately delayed.
2. Page servers and file servers must be available to supply pages demanded by user processes' backups.
3. The process server must be available to determine where new backups for fullbacks are to be located.

A backup user process is prepared to run by supplying any local information needed in the PCB and then scheduling its execution. Since it has no pages resident in memory, it will immediately page fault and gradually bring its address space into memory. Once it begins to execute, tests which are part of the normally executed code assure that it interacts correctly with the rest of the system. Messages which were already sent by the primary are not resent. On fork, the process checks whether it has any birth notices. If it does, it either avoids the fork altogether if the child process already exists, or uses information in the birth notice to fork a child with the same identity as its primary.

8. Efficiency Considerations

The Auragen 4000 project is an ongoing effort. A preliminary version of the operating system is now running on prototype hardware. However, it is not complete and contains substantial amounts of debugging code so that realistic performance measurements are not available.

However, we shall discuss the potential areas of efficiency and inefficiency by considering the overhead required to support fault tolerance.

8.1 Multiple Message Handling

Although most messages go to three destinations, they are transmitted just once across the intercluster bus. An efficient low level protocol assures that each target cluster is listening when the message is transmitted. It is true that at the receiving ends, three messages must be read into the cluster and distributed. However, message transmission, receipt, and distribution are all handled by the executive processor. Processes running on the work processors are not affected by the delivery of the two backup copies.

8.2 Backup Creation

By deferring the creation of backup processes for as long as possible and by making it the responsibility of the executive processor, we assure that the overhead is limited. In fact, the executive is responsible for all maintenance of backup processes, leaving the work processors free to do normal work unrelated to the support of fault tolerance.

8.3 Synchronization of Primary and Backup

The synchronization of a primary process and its backup is a potentially costly operation. However, we have minimized the extent to which it delays the pri-

mary. The primary interrupts its normal execution for only as long as it takes to place its dirty pages and the sync message on the outgoing queue. The process need not await completion of the operation by either the page server or the backup processor.

The interval between syncs is tunable with one exception. The current implementation requires a primary to sync prior to handling any asynchronous signal which is not ignored. We are looking into ways to avoid such forced syncs.

8.4 Crash Handling and Recovery

Crash handling was expected to be the area in which most overhead is incurred. However, we have attempted to minimize the slowdown. Processes unaffected by the crash, or only minimally affected (their correspondents have crashed) may begin to execute before all crash handling has been completed. Crash handling which requires that no regular user processes be executing is done by special high priority user processes. Remaining tasks, such as bringing up new backups is done by the executive processor, creating minimal interference with the unaffected processes.

9. Project Status

The development team has successfully transformed UNIX* into a distributed message-based operating system. We are currently integrating and testing process synchronization and crash handling. We expect to have a working two-cluster fault tolerant system in the early fall of 1983.

The difficulties we have encountered during two years of design, development, and implementation have resulted from the complexity of the task. The two major parts of the project, conversion of UNIX to a distributed system and the addition of fault tolerance, complicated each other and were difficult to divide into managable and independent pieces. Providing sufficient organization, without limiting creativity with over management, has been a formidable task.

Two factors have made our task easier. First, the simplicity of the underlying idea provided a commonly understandable and unifying base from which to begin. Secondly, we began with an existing operating system which defined our interface with the world and provided a common framework in which to build.

10. Future Developments

In the very near future Auros™'s fault tolerant capabilities will be extended in the following areas. Hardware failures which do not affect all processes in a cluster will not cause the cluster to crash, but will cause individual backups to be brought up for the affected processes. An efficient method for allowing asynchronous reads and writes and intracluster shared memory to be backed up will be implemented. The difficulty lies in the nondeterminism of asynchronous IO and of shared memory reference which would seem to require notifying the backup for each action of the primary. We plan to avoid the overhead of continually notifying the backup by waiting until the primary has an ordinary messge to send and including information about the results of any nondeterministic events with this message, a copy of which will be seen by the backup. A crash of the primary after this message can be handled by recreating the "nondeterministic" events deterministically in the backup. A crash before this message would wipe out any evidence of the events (since no message escaped the cluster prior to crash) and thus could be repeated in the backup without inconsistency.

11. Conclusion

This paper has described the design and initial implementation of an operating system in which message-based communication is used to assure that processes survive hardware failures. A message system, embedded in the operating system kernel, provides each inactive backup process with all information necessary to take over execution should its primary fail. Periodic synchronization of the backup with its primary limits the amount of recomputation required for the backup to catch up during recovery.

Fault tolerant operation is automatic and transparent to the user. This, together with UNIX compatibility, allows much existing software to be run fault tolerantly without modification.

The system is designed for use in on-line transaction processing environments where short delays during recovery are acceptable. Maximal productive use of resources in the absence of failure, at the expense of short delays during recovery, is of primary importance.

The separation of the operating system into a kernel for local management and resource control, and backed up servers for global resource management, provides many of the benefits of distributed operation without loss of central control.

References

(1) Bartlett, J.F., A NonStop Kernel, Proceedings of the Eighth Symposium on Operating Systems Principles, December 1981, pp 22-29.

(2) Baumbach, J., Memory Management in a Fault Tolerant System, In preparation.

(3) Denning, P., Fault Tolerant Operating Systems, Computing Surveys, Vol.8, No.4, December 1976.

(4) Ritchie, D.M. and K. Thompson, The UNIX Time-sharing System, Bell System Technical Journal 57, 6, July 1978.

(5) Russell, D.L., Process Backup in Producer-Consumer Systems, Proceedings of the Sixth SOSP, November 1977, pp 151-157.

(6) Stratus/32, VOS Reference Manual, October 1982.

*UNIX is a trademark of Bell Laboratories.