Fault-Tolerant Computing Based on Mach*

Ozalp Babaoglu

TR 89-1032
August 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* This work was supported in part by the Department of Defense Advanced Research Projects Agency (DARPA) under grant N00140-87C-8904. The views, opinions, and findings contained in this report are those of the author and should not be contrued as an official Department of Defense position, policy, or decision.

Fault-Tolerant Computing Based on Mach*

Özalp Babaoğlu†

August 29, 1989

Abstract

We consider the problem of providing automatic and transparent fault tolerance to arbitrary user computations based on the Mach operating system. Among the several alternatives for structuring such a system, we pursue the "task-pair backup" paradigm in detail and outline how it might be supported by Mach. Some of the new system calls and protocols that need to be incorporated into the Mach kernel and server tasks are sketched.

*This work was supported in part by the Department of Defense Advanced Research Projects Agency (DARPA) under grant N00140-87C-8904. The views, opinions, and findings contained in this report are those of the author and should not be construed as an official Department of Defense position, policy, or decision.

†Author's permanent address: Dipartimento di Matematica, Università di Bologna, Piazza Porta S. Donato 5, 40127 Bologna, Italy.

1 Introduction

Mach [1] is an example of the kernel-based technology for constructing operating system environments on multiprocessor and distributed systems. Other systems that subscribe to the same philosophy for structuring computing environments include the V kernel [8], Chorus [17] and Amoeba [16]. Among these systems, Mach has emerged as a particularly popular foundation for UNIX™ development on a wide range of interesting architectures.

The properties that make Mach appropriate for distributed computing (e.g., message-based interactions, many services being performed by user-level tasks, etc.) would appear to make it appropriate for fault-tolerant computing as well. In this work, we pursue this possibility and propose an architecture to support fault-tolerant user computations based on Mach.

The following section outlines the requirements for the hardware environment as it relates to fault tolerance. Section 3 gives an overview of Mach and considers the problem of structuring it on the hardware base. In Section 4 we identify the goals for fault tolerance that are desirable and feasible in the context of Mach. In light of these goals, various architectures for fault tolerance are discussed in Section 5. The paradigm of backing up each task with an inactive secondary is explored in detail in Section 6. We conclude the paper by listing problems that remain to be solved before the architecture can be realized.

2 The Hardware Environment

We consider a generic multiprocessor system as the hardware base. The system consists of multiple autonomous processing units called *nodes* that may be tightly coupled through a parallel bus or losely coupled through a network. There may or may not be support for shared memory at the hardware level.

We make the following assumptions about the failure characteristics of the hardware:

1. Processors fail by stopping. In other words, a fault processor never makes incorrect state transitions that are externally visible.

2. The hardware exhibits failure independence in the sense that the failure of one node does not affect the operation of other nodes.

3. There are no critical components whose failure would render large parts of the system inoperative. Examples of such components include power supplies, interconnection structures and communication links. If present, we assume that these components are either sufficiently reliable or are replicated a sufficient number of times so as not to be considered critical.

4. The communication subsystem may lose messages. Message delivery times are bounded and can be used to detect processor failures. Processor and communication failures never partition the system.

A large number of commercial multiprocessor systems have been constructed that satisfy the above assumptions to varying degrees. In particular, a conventional geographically distributed, losely-coupled network architecture comes very close to meeting all of the assumptions. In such a system, the communication channel (e.g., Ethernet) is typically the only critical component and it could be easily replicated to achieve the desired failure modes of the hardware.

3 The Software

3.1 Mach Overview

Mach is a communication-oriented operating system that is particularly well suited for multiprocessor and distributed architectures. It consists of a small, extensible kernel and a collection of user-level tasks that provide (perhaps several) operating system support environments. The abstractions supported by the Mach kernel are the following:

Tasks are the basic units of resource allocation and define an environment for execution.

Threads are the basic units of execution. Threads within the same task share its resources (including virtual memory). A thread contains the minimal state information to guarantee independent execution.

Ports are simplex communication channels that are protected by the kernel using capabilities. All references to objects in Mach are performed by sending messages to ports representing them.

Messages are typed collection of data that are communicated between threads.

Paging Objects are secondary storage objects associated with a a region of a task's virtual memory.

The Mach kernel itself can be considered a multi-threaded task that accepts messages corresponding to the kernel operations from user tasks. The procedural system call interfaces of the Mach kernel and servers are written in a remote procedure call language called MIG.

In addition to the primitives supported directly by the kernel, Mach provides complete 4.3bsd UNIX emulation. Currently, much of this emulation is supported directly in the Mach kernel. The so-called "de-kernelization" effort of moving all of the 4.3bsd emulation out of the kernel and into server tasks is ongoing but far from complete. In anticipation of an eventual separation of 4.3bsd emulation from the Mach kernel, we consider only the questions regarding the fault tolerance of the abstractions supported by the Mach kernel—tasks, threads, ports, messages and paging objects. Issues related to the fault tolerance of abstractions supported by server tasks (network server, memory server, file server, 4.3bsd emulation servers, etc.) can be addressed only when these server tasks are well defined. Furthermore, having fault-tolerant kernel objects at their disposal, many servers should attain fault tolerance at little or no additional effort.

3.2 Mach on the Hardware

In deciding a rational strategy for structuring Mach on a multiprocessor, it is useful to make the following distinctions based on the memory access characteristics of the architecture:

Uniform Memory Access (UMA) machines are shared memory multiprocessors where there is no difference in memory access costs with respect to regions of the physical address space.

Non-Uniform Memory Access (NUMA) machines, on the other hand, also share a common physical memory, but memory access costs may be different for different physical addresses (e.g., local vs. remote memory).

No Remote Memory Access (NORMA) machines classify physical memory as "local" and "remote" with respect to each processor and do not permit direct access of remote memory. This property is typical of distributed systems where processors can only communicate via message exchange.

Regardless of the actual underlying multiprocessor hardware, the fault tolerance considerations discussed in Section 2 require us to view the system as a NORMA model for structuring Mach. In other words, any multiprocessor hardware has to be seen as a losely-coupled collection of nodes with no shared resources. Given that there can be single failures that crash an entire node, there must be at least one independent Mach kernel per node. The same observation leads us to conclude that threads belonging to a single task should all be scheduled within the same node.

This distributed system view is similar to those of Tandem [2], Stratus [21] and Nixdorf [6] commercial fault-tolerant systems, which are all based on loosely coupled multiprocessor architectures. The Sequoia system [3], however, chooses to retain the NUMA architecture at the cost of having to develop a custom operating system of great complexity. Both the Stratus and Sequoia systems employ special hardware (processor pairs with hardware comparators) for hardware failure detection.

4 Goals for Fault Tolerance

With respect to fault tolerance, applications can be broadly classified as belonging to two domains:

- Process control
- Transaction processing.

The factor differentiating the two domains is the acceptable time for recovery after a failure. In the case of process control, real time constraints typically require immediate recovery. This, in turn, implies sufficient continual redundancy in the computation such that failures can be masked as soon as they occur. Resulting system cost is high—increased computational resources must be dedicated to replicate a single task, leaving reduced system capacity for other applications. Transaction processing applications, on the other hand, typically have less severe recovery deadlines—all that is required is that the system eventually recover from a failure. Resulting system costs can be kept sufficiently low in the sense that most of the hardware resources available are utilized for useful computation. Obviously, there is a continuum of such system designs that trade off recovery time for system overhead.

The fault tolerance goals we have for the Mach system are of the eventual recovery type after any single node failure. Note that a single node failure admits multiple failures as long as all of them are in the same node. This goal is consistent with our desire not to have to modify the hardware in any way and to obtain a system with high productive processing power in the absence of failures. Another goal of our design is the automatic and transparent integration of fault tolerance into user tasks. In other words, unmodified user programs that are desired to be fault tolerant should be automatically backed up and recovered when failures occur. This is in contrast to systems such as Tandem and Tolerant [22] that require reprogramming or preprocessing user programs to render them fault tolerant.

Our goals could be summarized as follows: Automatic and transparent tolerance of any single node failure with reasonable recovery delays and small failure-free operation overheads. In this sense, our goals are very similar to those of the Nixdorf TARGON/32 system.

5 Models for Fault Tolerance

Within the context of eventual recovery, computations can be structured in several different ways to achieve fault tolerance. The various approaches differ essentially in the manner in which they guarantee global system state consistency when parts of the computation have to be recovered from past states due to failures. Here we outline some of the principal models that have been proposed.

5.1 Transactions

In the transaction model [11], computation is divided into units of work called *transactions*. The system guarantees three properties for transactions: atomicity, serializability and permanence. Atomicity is with respect to failures in the sense that the execution of a transaction must be all or nothing—failures should never leave intermediate states of the transaction computation visible to other transaction. Serializability, on the other hand, requires that the concurrent execution of a several transactions should be as if they executed (in some arbitrary order) one after the other. Permanence guarantees that the computation makes finite progress despite failures.

The basic transaction model has been extended to distributed data models and nested transactions [15]. By definition, transaction boundaries always define consistent system states from which a computation can recover from. The major drawback of the transaction model is that fault tolerance cannot be integrated transparently to applications. Programs must explicitly use the transaction paradigm by announcing the beginning and end of transactions within programs at opportune points. Furthermore, while the serializability requirement of transaction-based recovery is appropriate for database applications, it can be overly restrictive for general distributed computations.

Modern systems that adopt the transaction model as the basis for fault tolerance in distributed computations include Arjuna [18], Argus [13] and Camelot [19]. Mach does not support the transaction abstraction at the kernel level. Efficient distributed transaction management systems can be built on top of Mach as a collection of server tasks. Camelot is one such system.

5.2 Checkpoint-Rollback

An arbitrary distributed computation could be made fault tolerant without having to structure it as a collection of transactions. All that is required is a mechanism whereby computations can be restarted from some past state in response to failures. To prevent having to restart computations always from the very beginning and thus guarantee forward progress, the state of the failure-free execution is periodically saved to stable storage. The saved past states are called *checkpoints*. The act of restoring a computation to a past state is called *rolling back* and the interval during which recovery is taking place *rolling forward.* The interval with which checkpoints are taken is a system tuning parameter and establishes the relative costs of the failure-free execution overhead and recovery delays.

In a system in which computations interact by exchanging messages or sharing data, rolling back a failed computation to an arbitrary checkpoint may result in an inconsistent global system state [7]. Intuitively, rollback should never result in a system state in which there are computations that appear to have received messages (read data values) that have not yet been sent (written). The manner in which global system state consistency is guaranteed results in two distinct strategies.

5.2.1 Optimistic Recovery

The general strategy is to design the failure-free execution of the system by "gambling" that failures will not occur. If indeed they don't, all is fine. If, however, the system encounters a failure, it must have collected sufficient information along the way to roll back those computations necessary to establish a consistent system state for recovery. For example, the scheme proposed by Strom and Yemini [20] allows checkpointing and message logging to stable storage occur asynchronously with computations but maintains the causality information necessary to allow the recovery algorithm to construct the global system state to roll back to. In a variant of the scheme [23], messages are logged in the nonvolatile memory of the sender rather than the receiver, resulting in even further asynchrony of stable storage writes with respect to computation.

An assumption common to both [20] and [23] is that computations are deterministic—an initial state and a sequence of messages to be received uniquely establishes the final computation state and the sequence of messages it sends. At the cost of additional complexity, Koo and Toueg present checkpointing and rollback algorithms that are suitable even when computations are not deterministic [12]. By storing two checkpoints per computation, their algorithms can also tolerate failures during checkpoint and rollback operations. An unfortunate consequence of optimistic strategies is that recovery time is difficult to bound since, in addition to the failed computation, an arbitrary number of others may need to roll back.

5.2.2 Pessimistic Recovery

A reasonable alternative to the above strategy is to structure the checkpointing mechanism in a manner such that recovery only involves the computations affected by the failure. This has the desirable consequence that recovery is both simple and more predictable in the delays it introduces to the system. The cost, obviously, is shifted from recovery to checkpointing.

To prevent arbitrary computations from having to be rolled back due to a failure, pessimistic schemes synchronize checkpointing with global interactions of computations. For example, if computations were to be checkpointed after each send operation, rolling back only the failed ones to their most recent checkpoint would always guarantee global state consistency. To avoid the substantial delays associated with checkpointing to a stable store, pessimistic schemes typically contain checkpoints in the context of a *backup* computation on another processor.

Notable systems that employ pessimistic recovery include Tandem [2] and Auragen [5] (while the Auragen approach was later adapted and further refined by Nixdorf for their TARGON/32 System [6], we continue to refer to it as the "Auragen approach"). Tandem uses process pairs to achieve fault-tolerant computations. Whenever the primary process engages in an activity that may induce global interaction (perhaps indirectly), it checkpoints its state to the backup. The responsibility for identifying these interactions and invoking the appropriate operating system primitives to do the checkpointing belongs to the programmer. The Auragen system, on the other hand, achieves application-independent fault tolerance automatically by performing checkpoints at fixed intervals but keeping the backup process always in a state where it can take over from the primary upon a failure. As with the schemes in [20] and [23], the Auragen method works only for deterministic computations. We elaborate on this scheme in the next section.

6 A Proposal for Fault-Tolerant Mach

In this section we outline a structure to render Mach tasks fault tolerant in a manner consistent with the goals of Section 4. The design we propose is essentially hardware independent—most distributed systems that have access to a fast broadcast communication medium could form the architectural foundation for the design.

Given the close harmony among our goals and the close match between the two computational models, we base our design on the Auragen approach. In this scheme, only user-level tasks can be made tolerant of single failures by backing them up on a processor with an independent failure mode than the primary. The operating system kernel is not backed up in the sense that a crash of a node could cause the kernel state to be lost. Those services of the operating system that need to be fault-tolerant (because they contain state information that needs to survive crashes), have to be moved out of the kernel and into server tasks. This is exactly the philosophy adopted by Mach.

6.1 Overall Structure

Each node of the system executes an independent Mach kernel that manages the resources local to that node (thread scheduling on processors, physical memory, ports, etc.). The units of replication and, thus fault tolerance, are Mach tasks. Threads are implicitly replicated within the backup task. The basic idea is to selectively back up tasks on different nodes. The backup task remains inactive but is kept in a state not to far behind the primary by periodically checkpoints and presenting it all of the messages that are received by the primary. In this manner, should the primary fail, the backup becomes active and rolls forward by processing the messages enqueued for it and takes over as the primary when recovery is complete. Given the assumption that all tasks (more correctly, threads, since tasks do not execute) are deterministic, the final state reached by the recovered backup and the messages it sends are guaranteed to be identical to those of the primary. The scheme also includes a mechanism to suppress resending of messages already sent by the primary.

In the following sections we outline the extensions and changes necessary to the Mach kernel to realize this scheme.

6.2 Task and Thread Creation

The kernel primitive to create a new task is extended to be

```
task_create(parent_task, inherit_memory,
            child_task, child_data, backup_type)
```

where the last parameter indicates how the child task is to be backed up. We discuss the various choices for backup type in Section 6.5. In case a backup is desired, it is created on a node different from the child (primary) task. In case the child is to inherit the parent's memory, it needs to be created on the same node as the parent task. The backup task is inactive in the sense that its threads are not selected for scheduling on the backup node. In Mach terminology, the task is created in the *suspended* state and as if a `task_wait()` call has been made. The kernel maintains a table that establishes the correspondence between the primary and the backup for the `child_task` and `child_data` ports. In other words, in trying to deliver a message to any port associated with the primary task, the kernel should have sufficient information to locate (perhaps with the help of a network server) the corresponding port at the backup.

Thread creation requires no changes as far as fault tolerance is concerned—the new thread is implicitly replicated within the backup task. The kernel, however, needs to record an association between the ports of the new thread and the backup task. This will be used in making copies of future messages available to the backup thread that were sent to this thread.

6.3 Messages and Communication

In Mach, communication ports are associated with tasks. A message sent to a task port can be read by any thread within the task. Furthermore, ports can be declared to be **unrestricted** in which case a thread can choose to receive *any* message among those enqueued at the unrestricted ports. This possibility represents a potential violation of our assumption that all executions are deterministic. Two tasks may end up in different states even if they start in the same initial state and are presented the same message sequence if the two make different choices with respect to delivering messages to threads that have been received at unrestricted ports.

To remedy this problem and eliminate non-determinism, we propose that the

```
msg_receive(header, option, timeout)
```

kernel primitive be modified such that when invoked with the `msg_local_port` field of the message header set to `PORT_DEFAULT`, the first message from the port with the *smallest* identifier [1] is delivered to the thread. We also need to guarantee that port identifiers are assigned the same relative ordering at both the primary and the backup.

The major new mechanism that needs to be included in the kernel is a 3-way multicast facility. To keep the backups up-to-date with respect to primaries, a copy of each message sent from one task to another needs to be also delivered to the two backups (if they exist). In the most general case, a message sent from task A to task B needs to be delivered to the following three destinations:

- task B (the primary destination)
- task B' (the backup destination)
- task A' (the backup sender).

The situation is illustrated in Figure 1. The 3-way multicast needs to be atomic in that either all three or none of the destinations receive the message and that two concurrent broadcasts are received in the same (arbitrary) order at all destinations.

Note that in the case the message is destined to a control port associated with a thread, we treat it as if it were destined to the task containing the thread as far as the multicast is concerned. Another issue to be dealt with in the case of messages to thread ports is the situation where the corresponding port (and thus the thread) at the backup does not yet exist. In this case, the message to the backup has to contain sufficient information (for example the relative ordering of the port identifier discussed above) for the message to be enqueued in a manner such that it will be read by the correct thread (yet to be created) if and when the backup has to recover.

The arriving message is treated differently at the three destinations. At the primary destination it is enqueued to be received and processed normally. At the backup destination it is simply enqueued on the corresponding port, if it exists, or on a fictitious port to be created in the future, as explained above. At the backup sender, the message itself is discarded

[1] Actually, any totally ordered field associated with the port is sufficient.

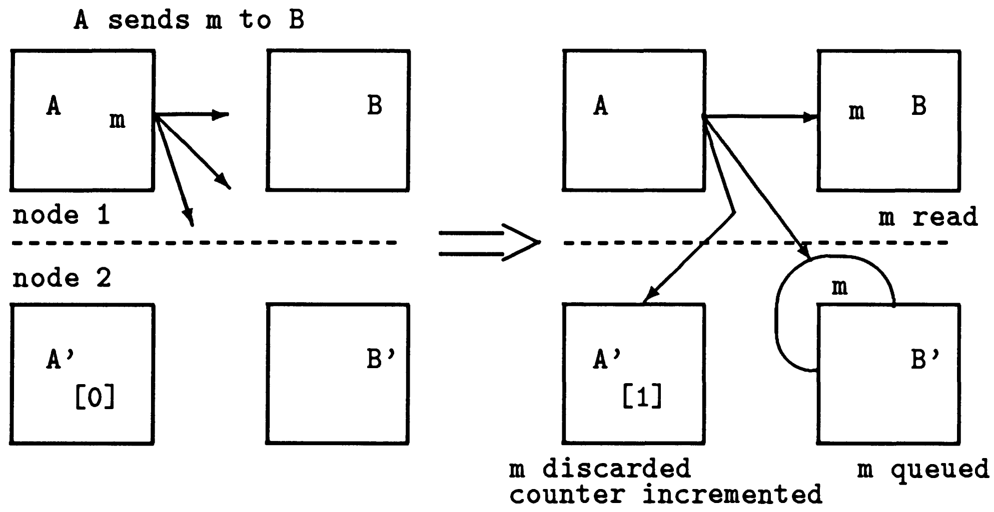

Figure 1: Communication Between Primary and Backup Tasks

but its arrival is used to increment a counter that represents the number of messages sent since the last checkpoint from the primary to the backup. We call this the *msslc* counter. The kernel primitives

```
msg_send(header, option, timeout)
msg_rpc(header, option, rcv_size, send_timeout, rcv_timeout)
```

need to be modified such that the *msslc* counter is decremented on each invocation and the specified message not sent as long as the counter value is greater than zero. In this manner, duplicate messages from the primary and its backup are suppressed during recovery.

Since each communication between fault-tolerant tasks involves a 3-way multicast, the overall efficiency of this scheme is critically dependent on the efficiency with which the multicast is realized. In systems such as Auragen and Nixdorf, the protocols necessary for atomic 3-way broadcast are incorporated in the bus arbitration hardware and low-level communication software. Since our system does not necessarily include any such support at the hardware level, it must be realized entirely in software. The obvious place to include an atomic 3-way multicast support in our system is in the network server of Mach. It is this server that extends the Mach IPC abstractions over a network between Mach kernels.

Typically, software solutions to the atomic 3-way multicast problem are complex and expensive in terms of both messages and delay [9]. In its most general formulation, the problem is equivalent to the Byzantine Agreement problem. One property of the hardware we can exploit in achieving efficient atomic multicasts is the existence of a fast broadcast network. The solution we propose is a protocol called *Trans* due to Melliar-Smith and Moser [14].

The basic idea of the *Trans* protocol is the following. Acknowledgements for broadcast messages are placed in messages that are themselves broadcast and are therefore seen by all other nodes. This avoids the necessity of each recipient acknowledging the message separately. In the failure-free case, a broadcast generates two messages—the broadcast message itself and the single (broadcast) acknowledgement.

To understand the protocol, consider the following example where node A broadcasts a message[2].

- Node B, among others, receives the message uncorrupted.
- Node B includes a positive acknowledgement for A's message in B's next broadcast. If B does not have a message to broadcast immediately, it broadcasts just the acknowledgement.
- Node C on receiving B's message is aware that A's message has been acknowledged and that there is no need for C to acknowledge it again. Node C, however, may decide to acknowledge B's message if no other node has acknowledged it yet.
- Suppose a fourth node, D, had not received the message broadcast by A. The message from B alerts D of its loss of A's message causing it to include a negative acknowledgement for A's message in its next broadcast.

The protocol generates an extremely small number of acknowledgement messages in practice and has small delays as well. On a high-speed and reliable bus it should deliver satisfactory performance. The protocol can

[2] More correctly, the network server task of the node where sender A lives broadcasts a message

also be extended to guarantee the total ordering requirement of atomic broadcasts as well. The current algorithms for achieving this additional property are computationally intensive and may generate high overhead for the network servers. We feel that this choice of the 3-way multicast protocol needs to be further evaluated and compared with alternatives such as the process group management and broadcast facilities of the ISIS toolkit [4].

6.4 Checkpointing

Periodically, the state of the backup task is made to coincide with that of its primary. This checkpointing action is initiated automatically by the kernel of the primary node whenever the primary task has read a certain number of messages. There is also an upper-bound on the real-time interval that can elapse between checkpoints, regardless of the number of messages read. These two system parameters determine the recovery delay characteristics of the system.

Checkpointing requires the cooperation of the Mach kernels (and perhaps the external pager tasks) at the nodes corresponding to the primary and the backup. The kernel of the primary task needs to keep track of the pages that have been modified since the last checkpoint and send them to the pager of the backup when asked to checkpoint.

The primary kernel is responsible for bringing the execution state of the backup task up-to-date. To accomplish this, the Mach kernel interface needs to be extended to include a

```
checkpoint(header, option, timeout, target_task)
```

primitive to be invoked by one kernel on another. The effect is to establish the execution state of `target_task` managed by the kernel indicated by port `msg_remote_port` (the Mach kernel at the secondary node) to be identical to that specified in the message. The message contains the states (registers, program counter, stack) of all of the threads contained within the task and the number of messages read by the threads in the primary from each of the ports associated with the task since the last checkpoint. The receiving kernel installs the state by creating the ports that do not already exist in the backup, deleting those that have been deallocated, removing the number of messages from the head of each port queue corresponding to the messages already read by the primary and resetting the *msslc* counter to zero.

A prudent policy would be to suspend the primary task until the checkpoint operation is complete. In this manner we have the guarantee that the backup can cleanly take over in case of an eventual failure of the primary. Certain properties of the message delivery system can be exploited to allow checkpointing to go on asynchronously with the primary execution.

6.5 Recovery

A failure can affect either a single task (e.g., failure of the CPU executing a thread of the task) or many tasks in a node (e.g., memory module or power failure). In the first case, the Mach kernel at the node where the failed primary was executing sends a `task_resume()` request to the kernel of the backup node to activate the backup. In handling this request, the backup kernel examines the `backup_type` parameter that was specified when the primary-backup task pair was created. If the type is `ANY`, the kernel creates a new backup for the backup (which is about to become the new primary) on any other node that is available. If the type is `HERE`, a new backup will be created on the same node as the failed primary (perhaps after waiting for the node to come up if it had failed completely). This may be dictated by tasks that are associated with the physical resources particular to that node and can only be restarted there[3]. Finally, if the type is `ONCE` no new backup is created once a primary fails.

In all cases, the kernel completes the resume operation by making the threads of the task executable. The network servers of the entire system are notified of the recovery so as to be able to construct the new members of the 3-way multicasts involving this task. The backup recovers simply by executing. Given that it starts from a checkpoint state and reads the same messages that the primary received, it eventually reaches the same state as the primary before its failure. Recall that the *msslc* counter prevents the recovering backup to resend messages already sent by the primary.

In case of a node failure involving multiple tasks, the entire node is declared down and each Mach kernel examines if it contains any tasks that are backups for primaries on the failed node. For each such task the above recovery procedure is performed.

[3] In [6], the authors note that since certain system server tasks have to be type `HERE`, it makes little sense to provide type `ANY` as the overall system availability cannot exceed those of the server tasks

Failure detection and notification is carried out by a collaboration between the nodes using a periodic polling mechanism.

6.6 Other Sources of Non-Determinism

In addition to the non-determinism of the message system discussed in Section 6.3, the fact that Mach tasks can be multi-threaded with shared memory can also lead to non-deterministic executions. When a task has multiple runnable threads, the scheduling among them can be arbitrary. Consequently, two initially identical tasks may have very different final states depending on the order in which their threads executed. The Mach scheduler must be modified to make thread scheduling deterministic. In a manner analogous to our solution for the message receive non-determinism, we propose that the scheduler discriminate between otherwise "equal" threads by picking the one with the smaller port identifier. Note that the scheduler can use whatever other criterion (e.g., priority) to partially order threads—the above rule is invoked only at the very end to break remaining ties.

Typical operating system services such as `time_of_day()` and `get_pid()` can be sources for non-deterministic computations. In Mach, all global services including time-of-day are handled by server tasks through the usual message communication mechanism. Consequently, a request for the time-of-day is guaranteed to return the same value to both the primary and the backup through the mechanisms outlined above. In Mach, all references to objects are done indirectly through (protected) ports. Thus, the equivalent of the "pid" in Mach is task port which can only be used indirectly in referencing and its value cannot be examined.

Finally, interactions between Mach tasks and the external world can lead to non-deterministic computations. These interactions can be explicit as in the case of signals being generated by a user from the keyboard, or implicit as in the case of page faults resulting from memory contention. Coping with these sources of non-determinism as well as evaluating the consequences of rendering scheduling deterministic remain open questions. In the worst case, we will have to confront non-deterministic computations squarely. The costs associated with this alternative are high. One possibility, as advocated by the Clouds system [10], is to actively replicate computations (thus consume processing power even in the absence of failures) and invoke an algorithm to elect one of the (correct but possibly different) outputs to commit as "the" output of the computation. Alternatively, we could adopt the algorithms of [12] and pay the price as complex checkpointing and rollback operations. Not only are the primitives more complex, rollback involves more computations since we can no longer rely on "replaying" recorded messages to generate the same sequence of messages.

References

[1] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of Summer Usenix*, pp. 93–112, July 1986.

[2] J. F. Bartlett. A NonStop Kernel. *Proc. Eighth Symposium on Operating Systems Principles*, pp. 22–29, Asilomar, California, December 1981.

[3] P. A. Bernstein. Sequioa: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing. *IEEE Computer*, 21(2), pp. 37–45, February 1988.

[4] K. P. Birman. Replication and Fault-Tolerance in the ISIS System. In *Proc. Tenth Symposium on Operating System Principles*, pp. 79–86, Orcas Island, Washington, December 1985.

[5] A. Borg, J. Baumbach and S. Glazer. A Message System for Supporting Fault Tolerance. In *Proc. Ninth Symposium on Operating System Principles*, pp. 90–99, Bretton Woods, N. H., October 1983.

[6] A. Borg, W. Blau, W. Graetsch, F. Herrmann and W. Oberle. Fault Tolerance Under UNIX. *ACM Transaction on Computer Systems*, 7(1), pp. 1–24, February 1989.

[7] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States in a Distributed System. *ACM Transaction on Computer Systems*, 3(1), pp. 63–75, February 1985.

[8] D. R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3), pp. 314-333, March 1988.

[9] F. Cristian, H. Aghili and R. Strong. Atomic Broadcasts: From Simple Message Diffusion to Byzantine Agreement. In *Proc. 15th International Symposium on Fault-Tolerant Computing*, pp. 200–206, July 1985.

[10] P. Dasgupta, R. J. LeBlanc and E. Spafford. The Clouds Project: Design and Implementation of a Fault-Tolerant Distributed Operating System. Technical Report, Georgia Institute of Technology, Atlanta, 1985.

[11] J. Gray, P. McJones, M. Blasgen, B. Linsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2), pp. 223–242, June 1981.

[12] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transaction on Software Engineering*, SE-13(1), pp. 23–31, January 1987.

[13] B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler and W. Weihl. Argus Reference Manual. Technical Report MIT/LCS/TR-400, Massachusetts Institute of Technology, Cambridge, Massachusetts, November 1987.

[14] P. M. Melliar-Smith and L. E. Moser. Trans: A Broadcast Protocol for Distributed Systems. Technical Report TRCS88-28, Department of Computer Science, University of California, Santa Barbara, December 1988.

[15] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing* MIT Press, Boston, Massachusetts, 1985.

[16] S. J. Mullender and A. S. Tanenbaum. The Design of a Capability-Based Distributed Operating System. *The Computer Journal*, 29(4), pp. 289-300, 1986.

[17] M. Rozier, *et al.* Chorus Distributed Operating System. In *Computing Systems*, 1(4), 1988.

[18] S. K. Shrivastava, G. N. Dixon, G. D. Parrington, F. Hedayati, S. M. Wheater and M. C. Little. The Design and Implementation of

Arjuna. Technical Report, Computing Laboratory, University of Newcastle upon Tyne, March 1989.

[19] A. Z. Spector, D. S. Daniels, D. J. Duchamp, J. L. Eppinger and R. Pausch. Distributed Transactions for Reliable Systems. In *Proc. Tenth Symposium on Operating System Principles*, pp. 127–146, Orcas Island, Washington, December 1985.

[20] R. E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transaction on Computer Systems*, 3(3), pp. 204–226, August 1985.

[21] D. Taylor and G. Wilson. Stratus. In *Dependability of Resilient Computers*, pp. 222–236, T. Anderson (Ed.), BSP Professional Books, Oxford, England, 1989.

[22] Tolerant Systems. Inc. Eternity Series: Technology Brief. Internal publication, Tolerant Systems, San Jose, California, July 1988.

[23] D. B. Johnson and W. Zwaenepoel. Sender-Based Message Logging. In *Proc. 17th International Symposium on Fault-Tolerant Computing*, pp. 14–19, Pittsburgh, Pennsylvania, July 1987.