

Transparent Recovery of Mach Applications

Arthur Goldberg* Ajei Gopal† Kong Li‡ Rob Strom§
David F. Bacon¶

Abstract

We have built a software layer on top of Mach 2.5 that recovers multi-task Mach applications from fail-stop failures. The layer implements Optimistic Recovery (OR), a mechanism for transparent recovery from failing tasks and processors, based on asynchronous checkpointing and logging of inter-task messages. OR recovers from failure by restoring a checkpoint and replaying the logged messages.

The current prototype supports message communication via sends and receives, simple port operations, and task interactions through the environment manager.

This paper discusses the issues Mach raised for this implementation, the structure of the OR layer, the design of future enhancements, and comparisons with other recovery techniques.

1 Introduction

During the lifetime of a distributed system one or more of its tasks may fail. Rather than abort the entire system due to a single failure, it is preferable to make the computation fault-tolerant—that is, the computation should continue to function correctly despite the failure of processors or individual tasks.

There are two approaches to fault-tolerance. Either fault-tolerance can be explicitly written as part of an application, or it can be written as a general

*IBM T.J.Watson

†Cornell University. This research was begun when this author was visiting IBM. Also partially supported by an IBM Graduate Fellowship.

‡UCLA. This research was begun when this author was visiting IBM.

§IBM T.J.Watson

¶UC Berkeley

purpose function available to many applications. The second approach—the one we choose—is based on transparent fault-tolerance.

Transparency implies two things. First, to make an application recoverable, no application code must be changed—at worst, the application may have to be recompiled or relinked. Second, the fault-tolerant system must produce the same output¹, even when failures occur, as the original non-recoverable system.

We have built a layer on top of Mach [3, 21] that makes multi-task Mach applications transparently recoverable from fail-stop processor failures. The layer implements Optimistic Recovery (OR) ([18]), an ‘optimistic’ mechanism for fault-tolerance.

A transparent recovery mechanism for Mach applications is desirable for several reasons. First, Mach is oriented towards distributed systems, so Mach applications are likely to be multi-task programs that run on multiple machines. Such applications are ideally suited for recovery since they suffer the failure of *some* component more frequently than a program running on a single machine.

Second, Mach applications are likely to be long-lived systems like computation and communication servers which users rely on, and therefore need to be automatically recovered.

We choose to build OR on top of Mach, rather than another operating system, for several reasons. First, while not a perfect match, the Mach interface does present the basic communication primitives of the OR model. Second, Mach is a young operating system with a small community of users, which we hope to impact with our experience. We hope that future implementations of the OR layer can be more deeply incorporated into Mach, so that applications can be made recoverable as a routine matter.

This paper describes the design and implementation of the optimistic recovery layer, of which we have a running prototype (called Optimistically Recoverable Mach or **ORM**). We describe the particular challenges posed by Mach and how we overcame them. In some cases, we suggest alternative or additional facilities to make solving these problems easier, particularly where the facilities we require are likely to be required by other services layered on top of Mach, such as distributed debugging and process migration.

¹Failures may cause stuttering—see Section 5.4.1.

2 Introduction to Optimistic Recovery

Optimistic Recovery is an efficient technique for recovering a recent consistent state of a distributed system after the failure of any number of system components. We will present an abstract system model and will then precisely define what we mean by “failure”, “recovery”, “consistent”, and “recent”. We then summarize the algorithm.

2.1 Abstract Model

A distributed system or *logical machine* is composed of a number of *recovery units* (RUs) connected by one-way communications links. Each recovery unit consists of a processor, a local *volatile storage* used by programs running within the recovery unit, and a local *stable storage* used by the recovery mechanism. Computations running within an RU read and write volatile storage, and send and receive messages over the communication links. A communication link connects the recovery unit either with another recovery unit in the logical machine, or with the *outside world* external to the logical machine. We assume that writing to stable storage can be asynchronous and can overlap computation. The recovery mechanism initiates the transfer of data to stable storage, and later receives an acknowledgement that the data has been written.

A *failure* of an RU consists of the following events: (1) the processor halts—no further messages are sent or received, and no further writing to stable storage occurs; (2) the contents of volatile storage is lost; (3) messages in transit on links to or from the RU are lost; (4) sometime later the RU starts up again. This is a *fail-stop* failure[16]. In particular, a failure does not cause any corrupted data to be written to stable storage or corrupted messages to be sent.

A failure in a distributed system consists of the failure of one or more (possibly all) of its RUs.

A *correct execution* is a trace of external inputs and outputs which would be observable in the absence of failure. Prior to a failure, the behavior of a system is some *prefix* of a correct execution. The state restored by the recovery mechanism after a failure is *consistent* if the future behavior is a *suffix* of that same correct execution.

Ideally, the behavior obtained by splicing the behavior prior to failure together with the behavior after recovery should be equal to the original failure-free execution. However, we cannot avoid a small amount of overlap or *stutter*—

external inputs received in the recent past may be lost and may have to be resent; external outputs sent in the recent past may be delivered again. A consistent state is *recent* if there is a small, bounded amount of stutter. In particular, there is no *domino effect*—an unbounded cascade of rollbacks to the global start state.

A *transparent* recovery mechanism, such as OR, is an extension to the operating system in each recovery unit such that any application program which behaves correctly in the absence of failure will behave identically (except for stutter) in the presence of failure without having to be rewritten.

2.2 Overview of Algorithm

OR is described in detail in Strom and Yemini’s paper[18]. Optimizations to reduce the cost of checkpointing and logging are described in other papers[17, 1, 19, 20].

OR is based on backwards error recovery[15]. A recent consistent state of a system is obtained by restoring recent states of each recovery unit such that for any communication link from RU A to RU B , (1) there are no *missing* messages—messages sent by A but not received by B , and (2) there are no *orphans*—messages received by B but not sent by A .

OR uses *checkpointing*, *logging*, and *replay* to restore states. Each recovery unit periodically takes a checkpoint by writing the state of its volatile memory to stable storage. Additionally, each input message processed by the recovery unit is logged to stable storage. We require each RU to be *piecewise deterministic*. This means that the state of any computation can be reconstructed by first restoring a prior checkpoint and then re-executing the computation using the logged messages rather than the communication links as the input source.

OR introduces a *session* protocol on each communication link. The sender *half-session* consecutively numbers each message, and saves a copy of messages not yet logged by the receiver. The receiver half-session checks for duplicate and missing messages. Duplicates arise as a result of replay; missing messages arise when a receiver fails after physically receiving a message but before logging it.

What makes OR “optimistic” is the fact that instead of avoiding orphans, we allow orphans to occur and subsequently roll them back. If A sends B a message while A ’s current state is not recoverable because an earlier input message has not yet reached stable storage, the message may become an orphan, because due to a failure, that earlier message may never be logged. Rather than delaying the sender by requiring logs to be “force-written”, optimistic recov-

ery instead tags each message with the identifiers of the unlogged antecedent messages and proceeds with sending the message, hoping that orphans are rare. These tags are called *dependency vectors*[18] or *dependency maps*. Each RU updates its local dependency map upon receiving a tagged message. The dependency map defines the set of unlogged messages on which the current state depends.

Most of the time no failure will occur—the antecedent message will eventually be logged, and a *log message* will be sent notifying other recovery units that the dependency tags may be removed from the dependency map. If all dependency tags are removed from a state or a message, the current state or message becomes *committable* and will never be rolled back. If a failure does occur, the dependent recovery units will be notified of which messages are lost. Any recovery unit containing a dependency on a lost message is known to have received an orphan message. The RU is rolled back to an earlier state which does not depend upon any lost message. All messages with receive sequence numbers greater than the sequence number of the last logged message before a failure are assumed lost. A new *incarnation* is begun starting at the next higher receive sequence number. Incarnation numbering is used to distinguish descendants of the message with this number from possible descendants of messages with the same receive sequence number in a previous incarnation.

The correctness requirement forbids us to send an uncommitted (potential orphan) message to the *outside world*. Otherwise one might observe a certain output prior to failure even though the restored state has a future inconsistent with the delivery of that output. Therefore, messages destined for the outside world are buffered by an *output boundary function* until all their causal antecedents are logged.

3 Issues in Implementing ORM

Implementing ORM required dealing with several issues: (1) deciding how to map the abstractions of the OR model, such as recovery unit, log, and communications link, into specific Mach entities in ways which satisfy OR's assumptions, such as piecewise determinism; (2) coping with features of Mach which make efficient implementation difficult.

We will summarize these issues separately:

3.1 Mapping the Abstract OR Model

Certain abstractions map fairly obviously to physical reality. For example, the “volatile storage” of OR corresponds to virtual memory of Mach applications; “stable storage” corresponds to disk; a “failure” is a halt and reboot of Mach, e.g. after shutting off power.² Less obvious are the mappings for the concepts “recovery unit,” “communication,” and “logical machine.”

3.1.1 Recovery Unit Granularity

We chose to map an RU to a Mach *task*. We thought this a good conservative decision for a first implementation. Each Mach task has an independent address space which can be viewed as the RU’s volatile memory. Mach tasks do not usually share memory. On the other hand, if we were to map an RU to a Mach thread, we would be faced with the problem of identifying which volatile memory belonged to which RU, and of mapping the shared-memory communication to message communication. Additionally, the smaller the RU, the more communication becomes inter-RU communication, which requires logging and dependency tracking. Finally, larger RU’s, e.g. clusters of tasks, are harder to make piecewise deterministic and would require enforcement of synchronization between tasks.

3.1.2 Communication

Communications links between RUs are Mach ports; OR messages are Mach “messages”; sending and receiving correspond to *msg_send* and *msg_receive*. For the most part the Mach model is a close fit to the OR model. However there are some differences, some of which necessitated careful design of **ORM**, others of which were temporarily ignored.

Mach ports differ from OR links in that in OR the sender knows the destination RU at the time a message is sent. In Mach, one has to do extra work to be able to determine the destination RU of a port for which one has send rights. Even then any such knowledge is only approximate because a port owner may pass receive rights at any time, even when the message is in transit. Additionally, after a failure, it is necessary to re-create ports, and Mach may assign different

²For a recoverable system which is embedded within another system, even the notion of failure may admit multiple interpretations. For example, one can conceive of an implementation in which a single user’s program is a “logical machine”. The cancelling of the program by action of the system operator may be considered a “failure” in that environment. However, in this paper we will assume failures are hardware failures or power failures.

port identifiers. Similarly, after a failure Mach may assign a port identifier which happen to be identical to one which previously designated a different port. Because of these problems, it was necessary to implement the concept of a “reliable port”. Reliable ports are to Mach ports as virtual addresses are to physical addresses. The applications programmer deals only with reliable ports; these ports never fail even when the communication link breaks or when recovery occurs. The recovery system tracks which Mach port corresponds to a particular reliable port. If no Mach port exists and one is needed, it is rebuilt.

In OR, there are two separate “receive” events: the physical arrival of a message, and the processing of the message by the program. Messages can be logged by the receiver as soon as they physically arrive. This can improve response time since if a message is logged sooner any external output which depends on the message can be released sooner. But there are two disadvantages: (1) we would have to dequeue the message from the Mach port in order to log it and then requeue it until the application program was ready to dequeue it; (2) if a message to port x arrives and is logged before another message to port y , it will be viewed as causally earlier, even if the program dequeues the second message first—this could result in a bogus dependency and an unnecessary delay in committing output. It is not clear whether it is better to log messages by arrival order or by dequeue order, but using dequeue order is easier to implement. Therefore in the current implementation, we pretend that messages are still in transit until they are actually dequeued.

When an application uses the Mach feature that associates message limits with ports, the mapping of OR “send” to Mach *msg_send* is not straightforward. The success or failure of a *msg_send* depends not only on the state of the sender, but also on the state of the receiver; therefore in principle dependencies flow from receiver to sender.

In the current implementation, we assume that all communication between tasks is via Mach messages. Mach actually provides several other means of communication. Tasks with appropriate rights can schedule other tasks. Memory can be shared between tasks as well as between threads within tasks. Finally, the current Mach implementation includes UNIX communication—signals, files, pipes, which eventually will be implemented using Mach, but which currently is not. Our current prototype ignores these complications, thereby restricting the scope of recovery. Our next plans call for the elimination of these restrictions in favor of increasingly “realistic” environments.

3.1.3 Boundary Function

OR assumes a strict boundary between the “inside” and the “outside” of a logical machine. Events on the inside are presumed unobservable and are allowed to be based on uncommitted (or even incorrect) computations which may be rolled back. Events on the outside are presumed observable, may not be rolled back, but may be delayed until they are committable.

Decisions on where to draw the boundary affect performance, ease of implementation, and the extent of the recovery. For example, ideally the window manager is *inside* the logical machine. Then if the machine is shut down or the power fails, on restart all windows will appear just where they were “recently” before the failure. In the current prototype, we chose not to modify the Mach kernel, but instead to recompile and relink recoverable programs with a modified Mach library and `mach.h`. It was not feasible at the time to remake the window manager, and therefore the windows are currently *outside* the logical machine. In our test system, we make an arbitrary assumption that each RU owns a single `xterm` window. The window is rebuilt when the RU recovers. In a future version of our prototype, we expect to locate the window manager *inside* the logical machine.

For performance reasons we must relax the strict output boundary function. If every key stroke and mouse motion is an input event, and every echo of a key stroke and every position change of the mouse cursor is an output event, there would be a need to delay the echo of a key until its causal antecedent (the keystroke) was logged, which is impractical, even with OR. Even in failure-free environments, the end user is accustomed to seeing “uncommitted” states of a screen—for example, intermediate bit patterns while windows are being moved around—and only trusts what is on the screen once it is “stabilized”. We therefore decided that screen output was not considered to have been “observed” unless it was on the screen for k milliseconds without being undone by a failure or recovery action.

Another component where there is a choice of where to draw the boundary is the file system. The file system could be viewed as an output device, receiving only committed data, but it is also possible to view the file system as the internal state (volatile storage) of an RU corresponding to the file server. There are optimizations to avoid logging all messages to a file server[1], and to avoid copying the whole file system in order to take a checkpoint[20]. The current implementation does not support recovering the state of the UNIX file system.

3.1.4 Piecewise Determinism

OR presupposes that each RU is piecewise deterministic. For a single-threaded Mach task without shared memory, this is straightforward. The computation and most of the system calls are deterministic. In certain cases the Mach documentation is not clear on whether a system call has non-deterministic results—e.g. *vm_allocate* for a nonspecific address. Those system calls with non-deterministic results must log their results, although if there is a usual result (e.g. *msg_send* completing without timeout), it suffices to log only the unusual results.

For a multithreaded Mach task, there can be nondeterminism unless the order of access to shared resources can be made deterministic, or it can be logged, or some determiner of this order can be logged.

Multithreading can be made approximately deterministic using the C-threads coroutine package, since there is actually only one thread which includes a C-threads scheduler. However, this implementation uses timeouts, and it would be necessary to log not only input messages, but timeouts as well.

If multiple Mach threads are run in parallel, then they could access shared memory and other shared resources such as ports in a non-deterministic order. Since our recovery layer intercepts system calls to operate on ports, it would be easy to log the order of these operations. However, it is difficult to trap the change of “ownership” of virtual memory without getting into the kernel and obtaining the assistance of the memory mapping hardware. We view this as impractical in the absence of further knowledge of the application structure such as knowing that the application was written in a high-level language with particular disciplined use of memory.

However, for an important disciplined use of shared memory (the concurrent reader, exclusive writer assumption), we *can* make execution piecewise deterministic provided that we log access to operations such as mutual exclusion (mutex) which the application needs to issue to guarantee correct usage. On replay we will force threads to obtain mutex locks in the order shown in the log, by blocking out-of-order operations. This guarantees that the shared resources will be accessed in the same order. Since Mach does not guarantee any particular coherence property to shared memory, (e.g. atomicity), it is unlikely that any program making undisciplined use of shared memory will be portable. Hence we expect that most applications will access shared memory through packages like C-threads which enforce some discipline such as concurrent reader exclusive writer. In particular, the Ada language requires its shared memory applications to obey these assumptions.

Our implementation is piecewise deterministic provided these assumptions about shared memory utilization within a task are met. How severe a restriction this is, how violations can be detected, and whether this restriction can be relaxed is a subject for further study.

3.2 Implementation Problems

Our work is in a preliminary stage, so we can offer only tentative conclusions about the problems of implementing transparent optimistic recovery on Mach.

On the one hand, the Mach philosophy of supporting a thin kernel with minimal state, with most system services implemented on top of Mach, and with a “small” set of primitives based on message passing is a good fit for the assumptions of optimistic recovery. In fact most of our problems are side effects of Mach’s not having gone far enough in exploiting the Mach philosophy.

Current problem areas are:

- **Undefined semantics:** The current Mach documentation is not always mathematically rigorous. Often we have to determine the semantics experimentally, and there is always the risk that we have interpreted a bug or release dependent feature as part of the definition. In particular, it is not clear what Mach promises or does not promise to be true of non-failing systems which are connected to failing systems.
- **State in the kernel:** The ideal operating system for OR is one with a completely stateless kernel. Any kernel state other than state used purely for performance enhancement needs to be made recoverable and needs to be “assigned” to a single RU.
- **Source vs. Object Compatibility:** We currently need to recompile Mach programs with a different include library and link them with a different run-time library. If we were able to intercept Mach kernel calls, we would be able to be Mach-compatible at the object code level. Unfortunately, because application programs actually build Mach message headers, we would introduce some inefficiency in replacing these with our different and slightly larger headers.
- **Duplication:** We have chosen to build our prototype on top of Mach. We would make kernel modifications only as a last resort. Unfortunately this often causes us to duplicate some work which Mach is also doing transparently to us. For example, we need to track when receive rights change ownership, so that we can associate messages with the appropriate OR

link. However Mach needs to do approximately the same work itself. Another example: OR “sessions” duplicate some of the protocol which Mach already provides to support reliable message delivery.

- Security: Because we have chosen not to modify the kernel, we are storing data structures needed for recovery in the virtual memory of the RU being recovered. A buggy program could modify out these data structures, cause OR to checkpoint or log bad data, and thereby prevent recovery.

4 ORM Prototype

ORM supports the following subset of the Mach interface:

- Ports: *port_allocate*, *port_deallocate*, *task_notify*, *task_self*, *environment_port*
- Messages: *msg_send*, *msg_receive*, *msg_rpc*
- Environment manager: *env_set_port*, *env_get_port*
- Miscellaneous: *scanf*, *printf*, *exit*

This subset is enough to write fairly complicated distributed applications.

A programmer writes code as though programming for a vanilla Mach system, with the following additions:

- The file *OR.h* must be included in each C file—this file contains a list of macros that replaces all occurrences of the calls given above by the corresponding OR calls. For example *port_allocate* is replaced by *PortAllocate*, where the latter is a routine we provide. In addition, it includes the data structures and globals that the user program must use. In particular, it defines an expanded message header structure, *MsgHeader_t*.
- A call to *ORInit* must be made in *main* before anything else is done; this call is discussed below. In the future we can avoid this by having the standard startup code make the call.

4.1 Overview of ORM

Suppose a programmer writes a multi-task Mach application, and wishes to make it recoverable. How does the programmer use our OR layer?

Each RU consists of two parts—user code, and OR code. Although the user sees no difference in the functioning of the program (save that the program becomes recoverable), several things are actually going on. First, OR creates several Mach threads in the task. Each of these threads is responsible for a particular piece of work—for example, asynchronous logging of messages, asynchronous checkpointing, updating log maps etc. Furthermore, all user calls to the supported subset of the Mach interface are replaced by calls to equivalent OR procedures.

Before running the application, the programmer must ensure that two special tasks, the *Recovery Manager* (RM) and the *Port Authority* are running.

4.2 Reliable Ports

Informally, a *reliable port* is a port that survives task and communications failures.

For example, suppose a Mach task holding receive rights to a Mach port p fails. Recovering the task is not sufficient to mask the failure, since other tasks holding send rights to the port can no longer send to the task.

However, when a failing task t represents an RU r , the above behavior is undesirable. We want recovery of t to replace it by another task t' , and for t' to acquire all the port rights that t owned. Furthermore, the acquisition must be transparent to the user code in any RU.

The reliable port abstraction is our solution to the above problem. A Mach application using OR manipulates only *reliable* ports, instead of Mach ports. Using OR replaces the type of a Mach port with the type of a reliable port so all user code that handles Mach ports actually handles reliable ports. The OR code maps reliable ports into Mach ports and manages the Mach ports required for actual communication.

Our design is based on the use of an additional name-server, called the *Port Authority* (PA). The PA maintains a table of mappings from reliable port to Mach port. In addition, it maintains a *dependency map* associated with each entry.

The operations we support include the creation of reliable ports, the deletion of reliable ports, the passing of send rights and the passing of receive rights. Furthermore, we ensure that all the information stored in the PA is treated only as a *hint*. There are two major advantages to this. First, we avoid any data consistency problems that might arise when the PA is replicated

across different machines. Second, there is no need to reliably log any of the information in the PA.

In this section we will deal with that part of the design that has actually been implemented in **ORM** including reliable port allocation and deallocation, passing send rights and recovery from failure. In Section 4.2.2 we will talk about the aspects of the design that are not present in **ORM**; primarily the passing of receive rights.

4.2.1 The Port Authority

In principle, a recovering RU can bind its reliable port rights to the Mach ports held by other RUs by interacting directly with the other RUs. The PA serves as an intermediary for these interactions, caching bindings from reliable port identifiers to Mach ports. Therefore, the PA is simply a convenience for the implementation of reliable ports, and need not be made reliable itself.

The **ORM** Port Authority interface supports three operations:

- void

```
PA_Advertise(r_port, m_port)
ReliablePort_t r_port;
port_t m_port;
```

An RU makes this call to bind the reliable port to the Mach port. It passes the send rights for the Mach port `m_port` to the PA.

- port_t

```
PA_Lookup(r_port)
ReliablePort_t r_port;
```

An RU makes this call to obtain send rights to the reliable port, `r_port`. The PA passes send rights for the Mach port it currently believes is bound to `r_port` back to the RU which makes the call.

- void

```
PA_Free(r_port)
ReliablePort_t r_port;
```

An RU makes this call to cancel any binding the PA has for the reliable port `r_port`.

Although the PA is stateless, for **ORM** to function, the PA must always be present.

4.2.2 Receive Rights Transfer

Recall that **ORM** does not support the transfer of receive rights. In order to do so, we require the following additions:

- The PA must associate a dependency map with each of its bindings. The same port may be owned by RU *A* for part of its lifetime and by RU *B* for another part of its lifetime. The PA will try to cache the latest ownership.
- It may be necessary for senders to retransmit missing messages. It is important not to retransmit messages which have in fact been successfully received by a previous owner. Therefore each RU needs to track the session sequence number of the last received message prior to acquiring the port.
- OR in an RU that owns the receive rights to a port must keep track of all the active senders to that port. This information must be passed when the receive rights are passed.

4.3 *ORInit*

ORInit performs initialization for a new RU. It performs several functions:

- Assigns the RU a unique identification.
- Creates reliable ports and bindings for *task_self*, *task_notify*, *environment_port*. These bindings are saved locally. They do not need to be registered with the PA.
- Initializes the other subsystems—half sessions, checkpointing, logging, interfaces with the PA and the RM.
- Creates the following threads:

ORMain Respond to OR messages and requests

Log Daemon Log messages asynchronously

Retransmission Daemon Retransmit unacknowledged messages

Checkpoint Daemon Checkpoint asynchronously

Reinitialization Daemon Restore a task's state from disk on recovery

4.3.1 *task_self, task_notify, environment_port*

All these calls return the reliable port bound to the appropriate port.

4.3.2 *port_allocate, port_deallocate*

```
kern_return_t
port_allocate(r_task, r_port)
Reliable_Task_t r_task;
Reliable_Port_t *r_port;
```

```
kern_return_t
port_deallocate(r_port)
Reliable_Port_t r_port;
```

In response to a *port_allocate*, OR first assigns a globally unique name *rp*. It then allocates a Mach port *mp*, and creates a local binding between *mp* and *rp*. Next, it registers the binding between *mp* and *rp* with the PA, via the call *PA_Advertise*. Finally, it returns *rp* as the reliable port *r_port* to be used by the application.

In **ORM**, a task can only allocate a port in itself, and not in another Mach task.

To reclaim resources on a *port_deallocate* of reliable port *rp*, OR removes the local binding, gets rid of the Mach port bound to *rp* and asks the PA to free *rp*.

4.3.3 `MsgHeader_t`

We have added several fields to the Mach message header structure `msg_header_t`:

- The session sequence number for the half session protocol
- The unique identifier of a local reliable port
- The unique identifier of a remote reliable port
- The unique name of the sending RU
- A dependency map

Application code compiled with `OR.h` will automatically use our expanded message header, and not the original Mach message header. OR also renames the `msg_local_port` and `msg_remote_port` fields so the programmer is hidden from the substitution of Mach ports by reliable ports, and also there is no need for reformatting of the header by OR.

4.3.4 Message passing

We support message passing via `msg_send`, `msg_rpc` and `msg_receive`. Furthermore, we support the transfer of send rights to a reliable port via a message header.

Suppose that a task wants to send some data to another task, listening to some port, *remote*. Suppose also that the task wants to give the other task send rights to a port *local*. In the Mach paradigm, the task initializes a message header—in particular, the fields `msg_local_port` to *local* and `msg_remote_port` to *remote*—and performs either a `msg_send` or a `msg_rpc`.

Now consider **ORM**, and recall that we have modified the definition of a message header. The user proceeds as before—initializing the message header and making a call to the message send mechanism. However, the call is intercepted by OR, which first performs a translation based on locally cached information, of the user supplied reliable ports into Mach ports—say the reliable port named *remote* into *mach_remote* and *local* into *mach_local*. Then, additional information, such as a sequence number, the RU name, and a dependency map are added to the header. Finally, the message is sent using a Mach call.

Suppose that the Mach call fails because port *mach_remote* is no longer valid. There are two reasons—either the port has been deallocated by the remote RU, or the remote RU has failed, and is being recovered. These cases can be distinguished by checking with other RUs. If no other RU is receiving on the port then it has been deallocated. Otherwise, eventually some RU will advertise the port with the PA.

When receiving a message, a Mach task must fill in a message header, including the port on which the message is to be received. This is followed by a call to the `msg_receive`. When the task is made recoverable, the `msg_receive` call is intercepted by OR, which translated the reliable port inserted by the user, into a Mach port.

Note that timeouts are possible in both message sends and receives. To permit deterministic playback after a failure, this timeout information is logged.

4.3.5 Summary of reliable ports in ORM

Any reliable port has a globally unique name. When a reliable port is allocated, the PA is given send rights to a newly allocated Mach port that is bound to that reliable port. Send rights to this Mach port are passed whenever send rights to the reliable port are passed. When an RU obtains rights to a reliable port, it caches the rights to the associated Mach port.

When an RU fails and recovers, it allocates a new Mach port for each reliable port for which it has receive rights. This new binding is sent to the PA (like when the reliable port was first allocated).

When an RU deallocates a receive reliable port, it informs the PA to delete any binding for that reliable port.

When an RU wants to send a message along a reliable port, it uses the locally cached Mach port (bound to that reliable port). If this port is bad, the RU continues to perform lookups on the reliable port in the PA, until one of the following:

- either the PA is able to return a good Mach port—corresponding to the new port installed by a recovered receiver, or
- the PA is unable to find a binding for that reliable port—corresponding to a deallocation of the port.

Thus, the overhead incurred in the absence of a failure is the registration of a reliable port with the PA. Subsequent message exchanges require only a local translation between reliable port to (a cached) Mach port. In the case of failure however, the receiver will have to register the new Mach port, and each sender will have to obtain send rights to that new port from the PA.

4.3.6 The ORM Recovery Manager

The **ORM** Recovery Manager is a task that checks to see if an RU has failed. If so, it tries to restart the RU from the appropriate checkpoint.

Note that the RM is stateless, as all the information required about running tasks is already available in the file system. Thus, there is no need to ensure that the RM checkpoints itself. RM is needed for automatic failure detection. If RM is absent, failure detection and RU restart must be performed manually.

4.3.7 Information logged

In **ORM**, the following information is asynchronously logged:

- Messages received—this causes an update of the log map. Additionally, an acknowledgement is sent to the RU that originated the message to prevent retransmissions.
- Inputs (via *scanfs*)—this causes an update of the log map.
- Timeouts on message sends/receives—this is required for deterministic playback on failure.

4.4 Saving and Restoring Checkpoints

Implementing rollback requires the preservation of the full state of a task. Our initial implementation checkpoints a task state by forking a child whose state is written out by an OR thread to disk while the parent continues execution.

In order to recover from a failure, the OR mechanism must take checkpoints of the RU states periodically. The states should include

- Executing states of all threads.
- The virtual memory image of the task.
- Mach ports
- Other Unix-related state: signal, file system, pipe, . . . etc.

The first two are necessary to restore a failed task. They can be retrieved through Mach kernel calls. We do not save the kernel state associated with Mach ports. Instead, the reliable port abstraction that we have done will handle the re-creation of Mach ports transparently. We save the name of a reliable port if the port for which this task has receive rights. When the task is restored, this information is used to generate a new Mach port and to advertise it to the PA. For those reliable ports to which the task has send rights, the recreation of Mach ports is not done until the restored task attempts to send messages on these reliable ports. At that time the PA can supply the necessary Mach ports and rebind them with reliable ports. For the Unix states, they are maintained by the kernel and there is no general way to extract them from the kernel, other than intercepting all Unix calls, recording its state, and

forwarding the call to the Mach kernel. Currently we do not save these Unix states.

In our implementation of **ORM**, each task has several OR daemon threads and user threads. In particular, the Checkpoint Daemon checkpoints its own task and saves necessary states, including thread states and virtual memory states, into disk by the following algorithm, the necessary Mach kernel calls are enclosed in parenthesis:

- Suspend all user threads and other daemon threads (`thread_suspend`)
- Allocate a thread buffer to store all thread states (`task_threads`, `vm_allocate`)
- For each threads, abort the thread (`thread_abort`), and get its states into the buffer (`thread_get_state`)
- Do a *unix_fork*

The *unix_fork* copies the current task into a child task with a single thread. The child task serves as a virtual memory image of this task, while the Checkpoint Daemon in the parent task takes the checkpoint. The reason to use a *unix_fork* instead of a *fork* is to get the virtual memory image of *all* threads; a Mach *fork* deallocates the virtual memory of all but the calling thread.

After a *unix_fork*, the child task suspends itself. The Checkpoint Daemon thread will

- Resume all user threads and other daemon threads (`thread_resume`)
- Write each virtual memory region of the child task into disk (`vm_region`, `vm_read`)
- Write out the thread state buffer and deallocate the buffer (`vm_deallocate`)
- Close the checkpoint file and terminate the child task

Instead of resuming all other threads after the checkpoint is written out, OR resumes them immediately after the *unix_fork* to minimize the freezing time. Note the Checkpoint Daemon can not checkpoint itself. Thus it must be recreated when OR restores a task.

Note that the current implementation even checkpoints OR daemon threads, which is not necessary since they can be recreated after a restore.

To restore a task is rather straightforward. First the restorer does a *unix_fork* to create a clone task which serves as a template for the new restored task. The clone task simply suspends itself. Then the parent task performs the following actions in the context of the clone task:

- Terminate all threads (`thread_terminate`)
- Deallocate each memory region (`vm_region`, `vm_deallocate`)
- Create and load memory regions of the restored task (`vm_allocate`, `vm_write`, `vm_protection`)
- Create each thread (`thread_create`) and set their state (`thread_set_state`)
- Resume the reinitialization thread—Reinitialization Daemon (`thread_resume`)

The reinitialization thread reinitializes all OR and user related states, creates the Checkpoint Daemon, and resumes all other threads.

5 Dependency Tracking

Dependencies are stored as a *dependency map* (DM) which is a table mapping RU i to incarnation, interval pair, expressed as $[\iota, \mu]$, where ι is the incarnation and μ the sequence number of the latest message received by RU i that causally proceeds this state. Each user task has a current DM (Task_DM) indicating the latest interval of each task that it depends on. The optimistic computation tracks dependencies by inheriting them in states and messages. When application code sends a message, OR transmits a copy of Task_DM with the message. Similarly, when an RU takes a checkpoint or application code outputs external data through the output boundary function, OR copies the Task_DM into the checkpoint and the output message.³

When an RU receives a message either from another RU or from the external input boundary function, OR increments the Interval number for the RU's entry in its Task_DM. When an RU receives a message from another RU, OR merges the message's DM into the RU's Task_DM, setting the entry in Task_DM to the most recent antecedent of the DM in the message and its Task_DM. If both maps contain RU X , one as (ι_1, μ_1) and the other as (ι_2, μ_2) , then the most recent antecedent is given by

³We currently store the DM in an enlarged header, `MsgHeader_t`, in an in-line Mach message. In the future we plan to support out-of-line messages, and store the DM in an out-of-line buffer. Reducing the number of message formats from 3 to 2 would simplify this work.

- (ι_1, μ_1) if $\iota_1 > \iota_2$
- (ι_2, μ_2) if $\iota_1 < \iota_2$
- $(\iota_1, \max(\mu_1, \mu_2))$ if $\iota_1 = \iota_2$

Absence of an RU entry in a DM means that the computation at that point does not depend on the RU. External addition of another RU to an OR system simply adds another entry to the DMs that depend on it.

5.1 Incarnation Start Table

The Incarnation Start Table (IST) is a function from an RU, incarnation pair to interval. $IST(R, \iota) = \mu$ means that RU R 's first action in incarnation ι was to receive the message that started state interval μ . By convention, $IST(R, 1) = 0$ since an RU can send messages when it starts up.

The IST is used to identify orphan messages. A message is an orphan if it depends on a discarded, or orphan, message as indicated by the IST. For example, if $IST(X, 2) = 4$ and $IST(X, 3) = 10$ then a message depending on state $(2, \iota)$ of RU X with $i \geq 10$ is an orphan. In general, an object depending on state (ι, μ) of RU X is an orphan if there exists $IST(X, \iota)$ such that $i > \iota$ and $IST(X, \iota) \leq \mu$. Note that this is a one way implication—a message that is not now an orphan can become an orphan in the future.

A running RU always knows its own IST. The RU stores its IST on stable storage.

5.2 Recovery

An RU starts a new incarnation after each rollback. On a rollback in incarnation ι RU X :

1. restores its oldest checkpoint
2. re-receives input messages from its log, stopping before the first orphan
3. if a failure occurred X reads the IST from stable storage
4. writes $IST(X, \iota+1) = \text{current receive sequence number} + 1$ to stable storage

5. sends RECOVER($X, \iota+1$, current receive sequence number + 1) to all other RUs
6. re-receive the remaining non-orphan messages from the log
7. begins communicating with other RUs

An RU also knows the IST function for other RUs although this may be out of date and incomplete. The IST is communicated in RECOVER messages, sent during recovery (above) and on request from an RU missing IST data.

5.3 Rollback

Since the IST determines whether a message is an orphan only two events can cause a rollback:

1. a failure
2. receipt of a RECOVER message that updates the IST

The IST for X at RU Y can be incomplete. A single failure of X should not cause this since Y will receive X 's RECOVER message, but a multiple failure may lose RECOVER messages. RU Y detects an incomplete IST when it receives a message that depends on incarnation ι at X and finds that $\text{IST}(X, \iota) = \text{UNDEFINED}$ for some $i \leq \iota$. Y must request a RECOVER(X, ι) message from X for all values of $i \leq \iota$.

A message or state with a particular dependency map becomes “committable” when all the messages it depends on have been logged. We assume an RU logs messages in receive sequence number order. A log map maps each element of a set of RUs to an incarnation, interval pair indicating the extent of the RU's logging. When an RU logs a set of input messages it updates its local log map and broadcasts the map to other RUs. Note that a logged message can later become an orphan.

To determine whether a dependency map is committable we compare it with the local log map. If RU X is only in the dependency map then it is not committable; if it is only in the log map then ignore it. If RU X is in the dependency map as (ι_D, μ_D) and the log map as (ι_L, μ_L) , then the dependency map is not committable if $(\iota_D > \iota_L)$ or $(\iota_D = \iota_L \text{ and } \mu_D > \mu_L)$. (We assume that the IST for X is complete up through ι_D so that orphans have been already eliminated.) Otherwise it is committable.

An RU cannot execute a message that depends on an interval whose IST is undefined. When it discovers the IST is incomplete, it delays execution of the message until the requested RECOVER message is received. When a RECOVER message arrives, the RU compares its dependency map with the IST update. If the dependency map depends on an orphan then the RU rolls back. In this way an RU effects any rollbacks for incarnation C before it starts depending on incarnations greater than C.

5.4 Output Boundary Function

OR produces output as soon as possible after it is committed. The Output Boundary Function (OBF) buffers output until it has been committed.

The current prototype replaces the *printf* with a function that converts the output into a string and passes the string to the OBF. Whenever an RU's log map is updated OBF attempts to output results that have been newly committed. In the future we plan to support the full stdio package by relinking the write calls stdio makes to a write function that buffers output until it is committed.

5.4.1 Stuttering

OR can stutter because logging a message that commits output and producing the output must be separate events. To minimize stuttering the prototype stably stores a count of the messages output by the OBF, which we update after completing the output. During the replay after recovery we discard all output with sequence numbers less than or equal to the stably stored count. Stuttering is still possible, since a crash between some output and incrementing the stable count will reproduce the output during the replay after recovery, but it is unlikely.

5.5 Logging Messages and Non-deterministic Events

A key challenge in implementing optimistic recovery, or indeed any system relying on rollback and replay, is logging all events that determine a task's behavior. The most obvious determinant of a task's behavior are the messages it receives. A message can be viewed as two parts: its content, and its position in the receive sequence at the task. The content can be logged either by the receiving task, as in **ORM**, or by the sending task, as discussed in [17, 11].

The merge position depends on unpredictable message communication delays, so must be determined at the receiver and should probably be logged there.

Another source of non-determinism in Mach is memory sharing between tasks, or between concurrent threads within an address space. On a uniprocessor, a deterministic thread scheduler (such as coroutine threads [4]) can be used to make the execution deterministic. We do not have a way of addressing this problem on a true multiprocessor, so **ORM** will initially be restricted to tasks running on a single cpu and without shared memory between tasks.

There is a third source of non-determinism in Mach which we had not expected, namely the ability of one task to modify the state of another task through a system call [3]. In some cases, such as the de-allocation of a port by another task, the effect is only seen when the former owner of the port tries to perform an operation on the port – in this case, logging of the non-deterministic event can be deferred until its effects become visible for the first time. In other cases, such as one task mapping a page into the memory of another task, there is no system call to use as a handle for logging. This is similar to the shared memory problem, and will not be supported in the first version of **ORM**.

6 Comparison with other systems

6.1 Transaction Systems

The original recoverable systems were database systems, and it remains true that most work on recovery assumes a transaction model. For many, the very word “recovery” carries the implication that the user has specified an atomic unit of work. Many experimental recoverable systems, such as Argus[13], Quicksilver[8], and Avalon/C++[9] assume a transaction model.

In transaction models the transactions or units of work must be explicitly defined by the programmer. Transactions run in parallel and share the database; typically the “recovery” mechanism enforces both atomicity and recovery.

The major differences in functionality between OR and transaction systems are: (1) transaction systems recover only some of the data, e.g. the database but not the program counter and local variables of long-running actions; (2) programmers must explicitly define units of work, and hence a conventional program cannot be transparently made recoverable; (3) at the end of a distributed transaction a two-phase commit is initiated and at least one record must be force-written, whereas OR continuously logs information to stable

storage and continuously commits computations whose dependencies are all logged.

6.2 ISIS

ISIS [12] is a distributed programming environment that has fault-tolerance as one of its goals. It provides the user with a variety of tools, including ones that allow the replication of a user-defined part of the process state, the maintenance of process groups, and so on. Using these tools, a programmer can create fault-tolerant applications. However, an efficient use of the ISIS tools calls for sophisticated knowledge and system design ability, and may be beyond most programmers.

In contrast to ISIS, in Optimistic Recovery, the programmer can be oblivious to the possibility of failure, and can write an application for the Mach interface, as though failures do not occur. This represents a considerable simplification over having to explicitly code in fault-tolerance.

7 Future Plans

One can classify our future plans into *realism* enhancements and *performance* enhancements. The realism enhancements will enable the OR layer to support a larger set of Mach and Unix primitives. This work includes:

- Link the stdio package with reads that call the input boundary function and writes that call the output boundary function.
- Support communication via out-of-line Mach messages.
- Support passing receive rights to reliable ports.
- Support dynamic creation of RUs (`fork()` or `task_create()`) and destruction of RUs (`exit()`). These operations must be made undoable, so that rollback can ‘un’create or ‘un’destroy an RU.
- Implement the Mach port set operations and the virtual memory operations.
- Recover the state of a task’s user interface, by incorporating the state of the window system, such as X, in the RU.

- Make an object-level compatible Mach/OR system which will automatically recover applications .
- Support more Unix functionality, so that signals, pipes and file connections are recoverable. This work's architecture will depend on the implementation of the Unix outside the mach kernel.

Before making performance optimizations, we plan to measure the performance of **ORM** and tune it appropriately. Many performance optimizations will be based on the work in the earlier papers, [17, 1, 19, 20].

- Garbage collect checkpoints and message logs
- Experiment with different logging mechanisms, such as sender-based logging, volatile logging, and null logging.
- Optimize file system access.

We also intend to use the general rollback capability of **ORM** to examine other optimistic systems, such as Time Warp ([10]), optimistic process replication ([7]) and optimistic parallelization ([2]). Other applications for generalized rollback are also inviting, such as distributed debugging.

8 Conclusion

We have built a prototype software layer on top of Mach that recovers failed multi-task Mach applications. The layer implements Optimistic Recovery (OR), which asynchronously logs inter-task messages to disk, and occasionally checkpoints each Mach task's state. OR recovers from failure by restoring a checkpoint and replaying the logged messages. Dependencies are transmitted with each Mach message and logging progress is broadcast. OR prepares output as early as possible, but delays writing to the screen until the messages the output depends on have been logged.

Our extensions to Mach for rollback and replay have other applications besides recovery. For example, rollback, or the ability to checkpoint a task and continue its execution in a different operating environment, effectively migrates a process from a machine, to a disk and then back to a machine, whereas normal process migration skips the disk. Another application that uses rollback are distributed debuggers of non-deterministic programs [5, 14, 6].

We hope to convince the Mach community of the need for services that allow straightforward checkpointing, since we believe it is a critical service for fault-tolerance, migration, and debugging. In particular, state in the kernel must be accessible in a form that can be used independent of the running kernel. We have addressed a major portion of this problem in Mach with our reliable port mechanism. A similar degree of effort will be needed to provide reliable pipes, reliable sockets, etc.

We hope to create an awareness that sources of non-determinism should be limited, and where feasible implemented in a way that they can easily be recorded. For programs that communicate only by Mach message, we have demonstrated that this can be done.

References

- [1] BACON, D. F. How to log all filesystem operations (while only writing a few to disk). Tech. Rep. RC, IBM T.J. Watson Research Center, 1990.
- [2] BACON, D. F., AND STROM, R. E. Optimistic parallelization of communicating sequential processes. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (April 1991).
- [3] BARON, R. V., BLACK, D., BOLOSKY, W., CHEW, J., DRAVES, R. P., GOLUB, D. B., RASHID, R. F., AVADIS TEVANIAN, J., AND YOUNG, M. W. Mach kernel interface manual. Tech. rep., CS Department, CMU, April 1990.
- [4] COOPER, E. C., AND DRAVES, R. P. C threads. Tech. rep., CS Department, CMU, October 1988.
- [5] FELDMAN, S. I., AND BROWN, C. B. IGOR: A system for program debugging via reversible execution. In *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* (May 1988), pp. 112–123.
- [6] FORIN, A. Debugging of heterogeneous parallel systems. In *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* (May 1988), pp. 130–140.
- [7] GOLDBERG, A. P., AND JEFFERSON, D. Transparent process cloning: A tool for load management of distributed systems. In *Proceedings of 1987 International Conference on Parallel Processing* (August 1987), pp. 728 – 734.

- [8] HASKIN, R., MALACHI, Y., SAWDON, W., AND CHAN, G. Recovery management in quicksilver. *Transactions on Computer Systems* 6, 1 (1988).
- [9] HERLIHY, M. P., AND WING, J. M. Avalon: Language support for reliable distributed systems. Tech. Rep. CMU-CS-86-164, Carnegie Mellon University, 1986.
- [10] JEFFERSON, D. Virtual time. *ACM Transactions on Programming Languages and Systems* 7, 3 (July 1985), 404.
- [11] JOHNSON, D. B., AND ZWAENEPOEL, W. Sender-based message logging. In *The Seventeenth International Symposium on Fault-Tolerant Computing* (June 1987), IEEE Computer Society, pp. 14–19.
- [12] KENNETH P. BIRMAN, R. C., ET AL. The ISIS system manual, version 2.0. Tech. rep., CS Department, Cornell, March 1990.
- [13] LISKOV, B., CURTIS, D., JOHNSON, P., AND SCHEIFLER, R. Implementation of argus. In *The Eleventh Symposium on Operating Systems Principles* (1987), ACM Special Interest Group on Operating Systems.
- [14] PAN, D. Z., AND LINTON, M. A. Supporting reverse execution of parallel programs. In *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* (May 1988), pp. 124–129.
- [15] RANDELL, B. System structure for software fault tolerance. *IEEE Transactions on Software Engineering SE-1*, 2 (June 1975), 220–232.
- [16] SCHNEIDER, F. B. Fail-stop processors. In *Digest of Papers from Spring Compton* (March 1983), IEEE Computer Society.
- [17] STROM, R. E., BACON, D. F., AND YEMINI, S. A. Volatile logging in n-fault-tolerant distributed systems. In *The Eighteenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers* (June 1988), pp. 44–49.
- [18] STROM, R. E., AND YEMINI, S. A. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems* 3, 3 (August 1985), 204–226.
- [19] STROM, R. E., YEMINI, S. A., AND BACON, D. F. Toward self-recovering operating systems. In *The International Conference on Parallel Processing and Applications* (L'Aquila, Italy, Sept. 1987), North-Holland, pp. 475–483.
- [20] STROM, R. E., YEMINI, S. A., AND BACON, D. F. A recoverable object store. In *Hawaii International Conference on System Sciences* (Kailua-Kona, HI, Jan. 1988), vol. II, pp. 215–221.

- [21] WALMER, L. R., AND THOMPSON, M. R. A programmer's guide to the mach system calls. Tech. rep., CS Department, CMU, December 1989.