Optimistic Recovery in Distributed Systems

ROBERT E. STROM and SHAULA YEMINI
IBM Thomas J. Watson Research Center

Optimistic Recovery is a new technique supporting application-independent transparent recovery from processor failures in distributed systems. In optimistic recovery communication, computation and checkpointing proceed asynchronously. Synchronization is replaced by *causal dependency tracking*, which enables a posteriori reconstruction of a consistent distributed system state following a failure using *process rollback* and *message replay*.

Because there is no synchronization among computation, communication, and checkpointing, optimistic recovery can tolerate the failure of an arbitrary number of processors and yields better throughput and response time than other general recovery techniques whenever failures are infrequent.

CR Categories Subject Descriptors: **[Operating Systems]**: reliability; D.4.7 **[Operating Systems]**: Organization and Design—*distributed systems*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Distributed algorithms, fault-tolerance message replay, recovery, optimistic algorithms, orphans transparent recovery

1. INTRODUCTION

Distributed multiprocessor configurations are replacing centralized processors as a result of an increasing demand for both higher throughput and higher availability. However, achieving high availability is more difficult in multiprocessor configurations because of the more complicated failure modes of such systems. This paper addresses the problem of restoring a consistent state of a distributed system following the failure of one or more of its processors.

We consider distributed systems that are constructed from *processes*, each of which maintains private state information and communicates with other processes by exchanging messages. As a result of communication, individual process states will become dependent on one another. A set of process states in which each pair of processes agrees on communication between them has taken place and which has not is called a *consistent* system state. If the state of a process that has sent a message is ever lost, then in order for the system state to be

Authors' address: IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.

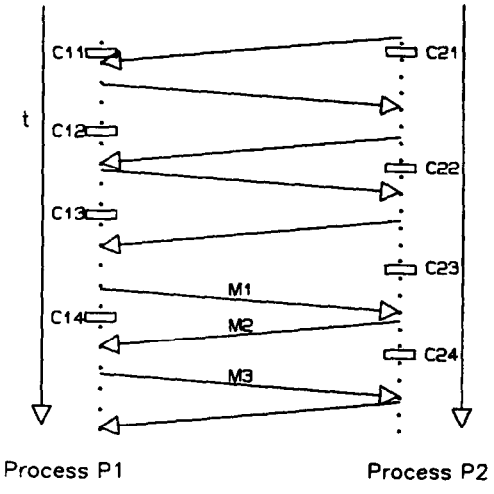

Fig. 1. Domino effect: If process P_1 fails and is restored to checkpoint C_{14}, it loses its memory of having received M_2 and having sent M_3. Thus P_2 will have received M_3 which was "never sent." If P_2 rolls back to C_{24}, it will have sent message M_2 which P_1 "never received." If P_2 rolls back to C_{23}, P_1 will have sent message M_1 not received by P_2, necessitating a further rollback of P_1.

consistent, the state change resulting from the receipt of that message in the receiving process must be undone; that is, the process must be *rolled back.*

A processor failure will cause the states of some of the processes executing on that processor to be lost. *Recovery* from a failure involves restoring a consistent system state. Recovery mechanisms typically recover from the loss of a process's state by retrieving a saved snapshot of an earlier state of that process, called a *checkpoint.* Since it is infeasible to take a checkpoint of an entire distributed system "at once," an attempt to recover may result in the unbounded cascading of rollbacks in an attempt to find a consistent set of individual process checkpoints. This problem is called *the domino effect* [14] and is depicted in Figure 1.

The domino effect is typically avoided by synchronizing checkpointing with communication and computation (see, e.g., [2], [4], [7], [11], and [12]).

This paper describes *optimistic recovery*, a new application-independent, transparent recovery technique based on *dependency tracking*, which avoids the domino effect while allowing computation, communication, checkpointing, and "committing" to proceed *asynchronously.* Because there are no synchronization delays during normal operation, optimistic recovery can make use of *stable storage*, that is, storage that persists beyond processor failures [10], thereby supporting recovery from failure of an arbitrary number of processors. The elimination of synchronization delays additionally yields improved response time over other transparent recovery mechanisms.

In this paper we describe the optimistic recovery protocols for recovering a consistent systemwide state following a failure of one or more processors in a distributed system. We do not discuss (1) means for detecting failures, (2) mechanisms for determining the new system configuration after a failure, (3) mechanisms for implementing a stable store, (4) mechanisms for providing reliable communication within the distributed system. Solutions to these problems are orthogonal to our recovery technique. The reader is referred to [1], [10], [13], and [19] for further discussion of these issues.

1.1 Goals for Distributed System Recovery

Our approach to distributed recovery has the following goals:

—*Application-independence.* The recovery technique should be applicable to arbitrary programs.

—*Application-transparency.* The recovery technique should be transparent to the programs being made recoverable. Application-transparency (1) simplifies programming; (2) allows both applications and recovery protocols to evolve independently, thereby avoiding the risk that the software becomes obsolete as a result of small changes in either the application or the underlying hardware; (3) enables preexisting programs to become recoverable without modification.

—*High throughput.* The CPU resources of all processors should be available for productive work when there are no failures.

—*Maximal fault-tolerance.* The recovery mechanism should provide recovery from failure of *any number* of processors of the system.

2. OPTIMISTIC RECOVERY

2.1 Overview

In optimistic recovery, computation, communication, and checkpointing proceed *asynchronously.* Instead of consistent checkpoints being maintained at all times, enough information is saved to reconstruct a consistent state *after* a failure. When reconstructing a consistent state, we face the following problem: A process P_1 may receive and process a message M from a process P_2, but P_2 may fail before having recorded enough information on stable storage to enable restoring the state from which it sent M.

Optimistic recovery solves this problem by having each process *track its dependency* on the states of other processes with which it communicates. As a result of dependency tracking, it is possible for P_1 to detect that it has performed computations that causally depend on states that the failed process P_2 has lost. Such computations are sometimes called *orphans.* If such computations have been performed, they will be *undone* by restoring an earlier state of P_1 that does not depend on lost states.

A state is restored by first restoring an earlier checkpoint from stable storage and then *replaying logged messages*—that is, reexecuting the process by driving it from the sequence of input messages saved in stable storage. Because we control the extent of rollback by replaying the correct number of messages, the system never rolls back "too far" and hence avoids the domino effect.

The optimistic recovery protocols ensure that the externally visible behavior of a distributed system incorporating these protocols is equivalent to some failure-free execution. By "equivalent," we mean that all messages sent outside the distributed system in the failure-free execution that would be sent in the same order during actual execution and that no other messages will be sent. Despite the existence of processor failures that result in the loss of the recent state of some processes, we meet the above correctness criterion by (a) restoring an earlier possible state of the failed processes using rollback and replay, (b) rolling back

other processes whenever these have been determined by dependency tracking to depend on lost states, and (c) committing messages to the outside as soon as it is determined from dependency information that the states that generated the messages will never need to be rolled back.

2.2 Concepts and Definitions

2.2.1 *Logical Machine and Recovery Units.* A cluster of machines incorporating optimistic recovery appears to applications executing on it and to other systems communicating with it as a single *logical machine*, as shown in Figure 2.

The logical machine is partitioned into a fixed number of *recovery units* (RUs), which communicate with one another through message passing. Application processes may be created and destroyed dynamically and are assigned to particular recovery units. In optimistic recovery, recovery units, rather than individual processes, fail and are recovered.

By allowing each recovery unit to schedule multiple processes, and by holding the number of recovery units fixed, we simplify the recovery algorithms without restricting the dynamic variability of the system workload. In addition, it becomes a system design parameter whether to have many "small" recovery units or a few "large" ones.

We make the following assumptions about the logical machine:

—*Reliable, FIFO channels between recovery units.* These can be implemented by any of a number of communication protocols (see, e.g., [19]). We assume nothing about the arrival order of messages sent to a recovery unit from two different sources.

—*Fail-stop* [15]. All failures are detected immediately and result in halting the failed recovery units and initiating recovery action. (We later weaken this assumption. It suffices to require that failures be detected before any event resulting from them is made visible ("committed") outside the logical machine.)

—*Independence.* Failures will not recur if the recovery unit is reexecuted on another processor. (We later weaken this assumption as well in our discussion of recovery from software faults.)

—*Stable storage.* Recovery units store their current state in volatile storage, which is lost upon their failure. Information needed to reconstruct volatile storage is maintained in stable storage and persists across failures [10].

—*Spare processing capacity.* It is always possible to relocate a failed recovery unit to some working processor, which will be able to access the previously logged recovery information on stable storage. We assume that physical processors will multiplex the workload of several recovery units. Relocating a recovery unit to another processor may degrade performance but will have no other visible effect.

2.2.2 *State Intervals.* We assume the behavior of each recovery unit to be *repeatable* and *message driven.* That is, the state of the recovery unit can be regenerated by restoring an earlier state, restoring the subsequent input message queue in its original order (using the message log described below), and replaying the processing of the recovery unit. Thus, we can identify a state of a recovery unit by the ordinal number of the last input message that it processed.

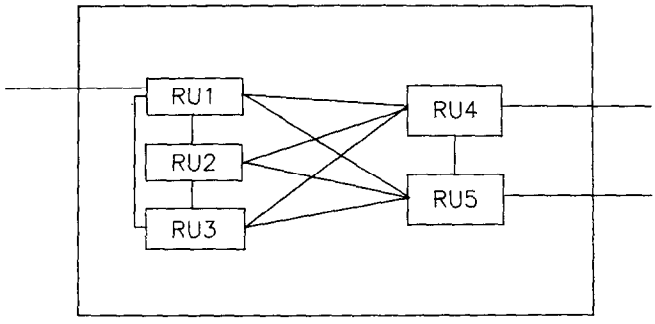

Fig. 2. Logical Machine: The Logical Machine is seen by the external world as a single machine communicating through channels, depicted by double lines. Internally, it contains multiple communicating recovery units.

Suppose RU_i has already processed its first $n-1$ input messages and is ready to process its nth input message $M_i(n)$. Its volatile storage is in some state, which we shall call $S_i(n)$. Given state $S_i(n)$ and message $M_i(n)$, RU_i's processing component will execute a series of computations, which we call *state interval* $I_i(n)$ of RU_i. During state interval $I_i(n)$, RU_i may conceivably generate output messages destined to other recovery units or to outside the logical machine. When RU_i's processing unit is ready to dequeue message $M_i(n+1)$, state interval $I_i(n+1)$ is started (Figure 3).

The property of repeatability implies that an arbitrary state $S_i(n)$ of RU_j can always be restored, provided we can recover an earlier state $S_i(n-d)$ and the subsequent messages $M_i(n-d)$ through $M_i(n-1)$. $S_i(n)$ is restored by *replaying* the processing of these messages in order, starting at $S_i(n-d)$.

2.2.3 *Incarnation Numbers and State Indices.* Because RU_i may roll back and then resume processing either as a result of its own failure or in response to failure of another recovery unit RU_j which is unable to reconstruct states that have affected RU; some input message ordinal numbers (and the corresponding state interval numbers) may be reused. In order to continue to have a unique way of identifying state intervals, we designate each input message of a given recovery unit and its corresponding state interval, by a *message* or *state index*, which is a pair $[\iota, \mu]$, where μ is a *message number* and ι is an *incarnation number.* The incarnation number of a recovery unit is incremented each time a recovery unit resumes processing after having rolled back (see Figure 4.)

2.2.4 *Live History.* A state interval of a recovery unit RU_i is *live* if it has not been rolled back. The *live history* of a recovery unit is the sequence of state intervals of that recovery unit that have not been rolled back. The live history constitutes a sequence of state intervals that could have arisen during a failure-free execution of the recovery unit. For example, in Figure 4, the live history consists of state intervals [1, 1] through [1, 5], [2, 6] through [2, 8], and [3, 9] through [3, 13].

A state interval $[\iota, \mu]$ of RU_i is a *live predecessor* of a state interval $[\iota', \mu']$ iff $[\iota, \mu]$ precedes $[\iota', \mu']$ within the live history of RU_i. We use the symbol $\prec$ to

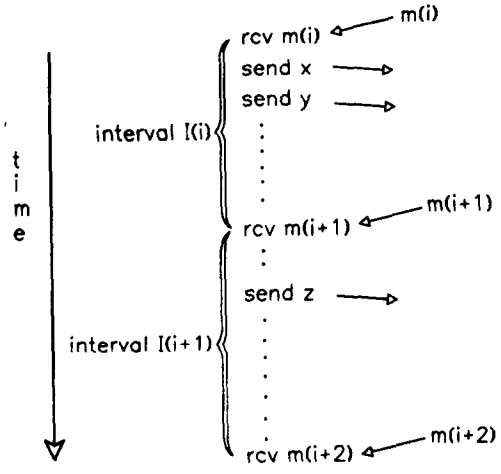

Fig. 3. Numbering messages and intervals.

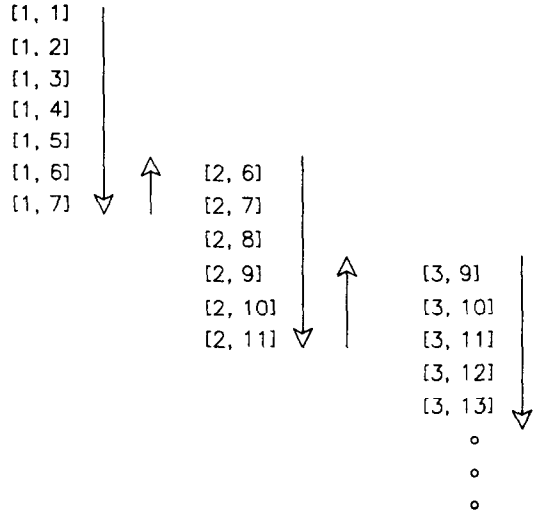

Fig. 4. Numbering messages in the presence of rollbacks.

denote the live predecessor relation. Thus, while the actual execution order follows the lexicographical order, $[\iota, \mu] \prec [\iota', \mu']$ means that $[\iota, \mu] < [\iota', \mu']$ according to the lexicographical order, and that no other interval that precedes $[\iota', \mu']$ supersedes $[\iota, \mu]$. For example, $[1, 5] \prec [2, 10]$, whereas $[1, 6] \not\prec [2, 10]$.

In a correct implementation of optimistic recovery, all messages committed to outside the logical machine depend only on live histories.

2.2.5 *Causal Precedence.* State intervals in a logical machine are partially ordered by a *causality relation.* Within a recovery unit, the order of the state intervals in the live history, that is $\prec$, determines the causality order: Each interval is caused by its live predecessor interval. Between recovery units a partial order is induced by the sending and receiving of messages. State interval $[\iota_i, \mu_i]$ of RU_i *immediately causes* interval $[\iota_j, \mu_j]$ of RU_j whenever a message sent from

RU_i during interval $[\iota_i, \mu_i]$ is dequeued by RU_j to begin interval $[\iota_j, \mu_j]$. The transitive closure of the relations "immediately causes" and $\prec$ results in a partial ordering over the set of state intervals in the distributed system, which we call *causal precedence* or *dependency.*

2.2.6 *Possible States.* A state interval is said to be *impossible* iff it depends on two state intervals of the same recovery unit that cannot both be live; that is, the two state intervals have identical message numbers and different incarnation numbers. Otherwise, it is said to be *possible.* For example, suppose a state interval depended on both states [3, 13] and [2, 10] within the recovery unit shown in Figure 4. Since [3, 10] $\prec$ [3, 13], the state interval also depends on interval [3, 10] and is therefore impossible.

If a state interval is possible, its dependencies can be encoded by a single index into the live history of each of the recovery units. This index denotes the "latest" state interval of each recovery unit on which the possible state interval depends and indicates a causal dependency on that state interval and all its live predecessors.

2.2.7 *Logging.* Each recovery unit periodically logs information to stable storage in order to support recovery. Logging to stable storage is *not* synchronized with communication. There are two kinds of information saved on stable storage: checkpoints and input message logs.

Checkpoints are snapshots of the complete state of a recovery unit. Checkpoint frequency is a tuning option: More frequent checkpoints may imply more disk activity; less frequent checkpoints may result in recovery taking a longer time.

An input message is said to be "logged" whenever both its data and the ordinal position in which it is processed can be obtained on demand during recovery. (There exist optimizations in which it is not necessary to actually write some or all of the message to stable store in order to "log" it, because the message is known to be reconstructible from other stable information in the system.)

2.2.8 *Lost Messages, Lost States, and Orphans.* Messages processed but not yet logged by a recovery unit at the time of a failure are *lost messages*; the corresponding state intervals are called *lost state intervals.* Messages and state intervals that are either lost or causally dependent on lost state intervals are called *orphans.* State intervals that will never become orphans are called *committable.*

Note that a message can be considered "lost," even when its data value is completely recoverable, if the ordinal position the message occupied in the input message stream of the receiving recovery unit is unrecoverable. This is because the relative order in which messages sent by different recovery units are merged is not deterministic, and, therefore, upon recovery the message might be merged in a different order, and the computation may be different.

2.3 Components and Data Structures

A recovery unit consists of (1) a set of input and output *half-sessions*, (2) a *merge component*, (3) a *processing component*, and (4) a *recovery manager component.* Each recovery unit maintains (1) a *dependency vector*, (2) a *log vector*, and (3) *an incarnation start table*, as well as checkpoint and message logs on stable storage (Figure 5).

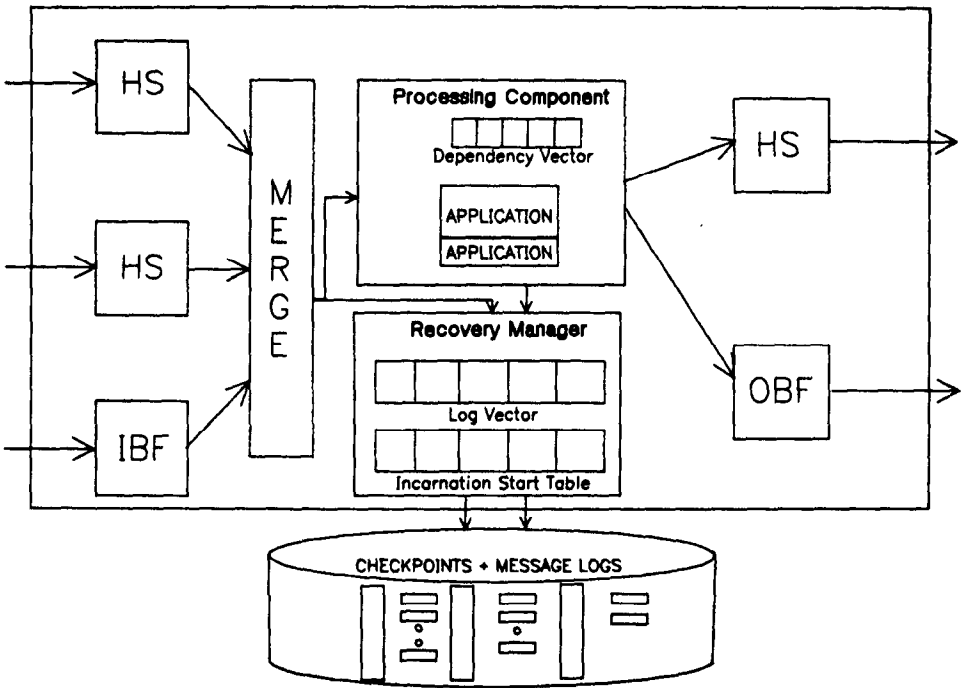

Fig. 5. Structure of a recovery unit.

2.3.1 *Sessions and Boundary Functions.* Each recovery unit may receive messages from other recovery units and from input devices external to the logical machine. Each recovery unit may also send messages to other recovery units and to output devices external to the logical machine.

The protocol by which a recovery unit receives data from an external device is called the *input boundary function*; the protocol by which a recovery unit sends data to an external device is called the *output boundary function.* Between each pair of recovery units, there is a pair of unidirectional *sessions.* A session consists of a pair of *half-session* protocols—an *output half-session* in the sender and an *input half-session* in the receiver. Figure 6 shows a session between recovery units in the dashed box.

Session protocols serve the purpose of detecting lost and duplicate messages resulting from failures. Boundary function protocols are mostly device specific; however, output boundary functions additionally delay messages intended to be sent outside the system until they are committable.

2.3.2 *Merge Component.* The merge component combines all the input message streams from the input half-sessions and the input boundary functions into a *merged input stream.* This component assigns each message in the merged input stream an *ordinal position number*, which corresponds to the order in which messages will be processed by the processing component. The merge component may be implemented very simply—for example, first-come, first-served—or it may be more sophisticated. For example, it may take advantage of the fact that

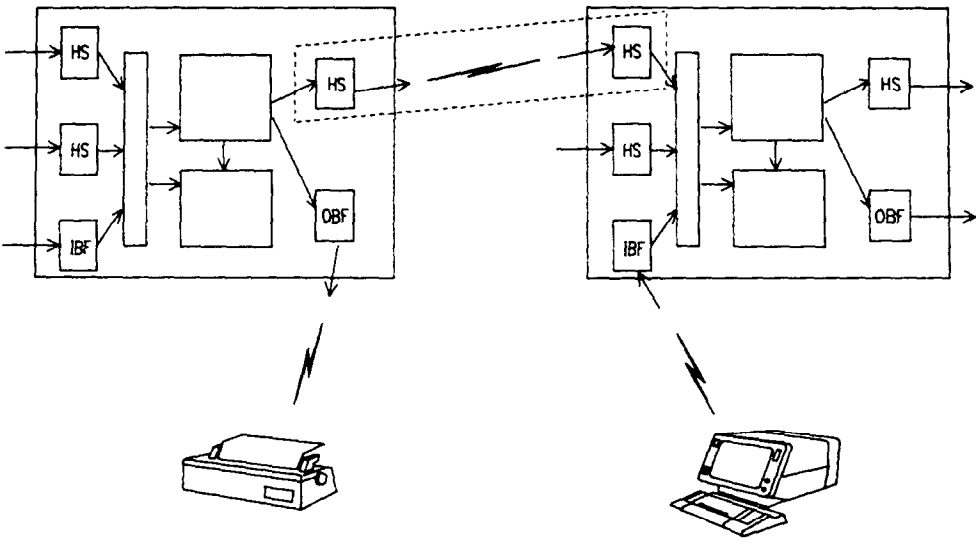

Fig. 6. Sessions between recovery units.

the recovery unit schedules many processes, and it may delay inserting a message into the input stream if it is destined for a process that is not ready to receive it. The merge component is *not* required to be deterministic, because its output is logged, and the log is used to reconstruct the merged sequence if replay is required.

2.3.3 *Processing Component.* The processing component is that part of a recovery unit where the application processes execute. Application processes running in the processing component keep their state in volatile storage. The application processes are unaware of the recovery protocols. If several application processes are allocated to a single recovery unit, the processing component is responsible for scheduling these processes.

The processing component is driven by the merged input stream. The processing component must be deterministic.

2.3.4 *Recovery Manager Component.* Each recovery unit's recovery manager is responsible for maintaining recovery information on stable storage. This includes scheduling checkpointing, logging messages, and reclaiming obsolete checkpoints and messages. The recovery manager is also responsible for recovering its recovery unit following a failure. Recovery consists of restoring the recovery unit's *earliest* checkpoint and replaying the subsequent message log. When the message log is exhausted, the recovery manager is responsible for broadcasting an appropriate *recovery message.*

2.3.5 *Dependency Vector.* In a logical machine with m recovery units, the causal predecessors of a possible state interval I of a recovery unit RU_i can be

encoded by a vector of state indices, $\langle d_1, d_2, \ldots, d_m \rangle$, provided that RU_i never enters an impossible state. The causal predecessors of I in each RU_j are the set of state intervals d, such that $d \leq d_i$. We call this vector a *dependency vector.*

As part of its internal state, each recovery unit RU_i maintains a dependency vector DV_i, which identifies the causal predecessors of RU_i's *current* state interval. The ith component of DV_i, $DV_i(i)$, is the index of RU_i's current state interval.

If state interval I of RU_i depends on intervals $d_1, d_2, \ldots, d_m$ of the m recovery units of a logical machine, then if *any one* of these intervals cannot be recovered after a failure, I must be rolled back.

2.3.6 *Incarnation Start Table.* To be able to identify rolled back states, each recovery unit keeps an *incarnation start table*, which records the earliest message number in each incarnation of each recovery unit. We use the notation $IST(\iota, k)$ to denote the earliest message number in incarnation ι of RU_k. For example, if RU_k is the recovery unit shown in Figure 4, then $IST(1, k) = 1$, $IST(2, k) = 6$, and $IST(3, k) = 9$. The incarnation start table enables the processing component to determine whether a given state interval is part of the live history of a recovery unit, or whether it has been rolled back.

An interval $[\iota, \mu]$ is part of the live history of RU_k iff

$$\nexists \iota' \ni \iota' > \iota \wedge IST(\iota', k) \leq \mu,$$

that is, that there does not exist a later incarnation ι' of RU_k that starts at message number μ or less.

The incarnation start table entry for an incarnation ι is needed only as long as messages depending on incarnation numbers less than ι still exist in the logical machine. In practice, if the logical machine has been failure free for a long enough time, there is only one relevant incarnation number for each recovery unit, and the incarnation start tables can be empty.

2.3.7 *Log Vector.* Each recovery unit logs its input messages in the background—the more messages logged, the more computations become committable.

In order that a recovery unit be able to determine which of its computations are committable, it must know both the status of its own logging progress and the logging progress of the other recovery units in the logical machine. This information is recorded in a *log vector* LV, maintained in each recovery unit. The ith component of LV_k is a state index l_i of a state in RU_i such that it and all its live predecessors have been logged. LV_k reflects the current logging status of the recovery units in the logical machine as perceived by RU_k.

The actual logging status of a logical machine may be further ahead than indicated by any of the local log vectors. Log vectors are updated by periodically and asynchronously broadcasting local log vectors to other recovery units. Since messages once logged remain logged forever, log vectors are strictly monotonically increasing. Log vectors are used to determine the committability of state intervals.

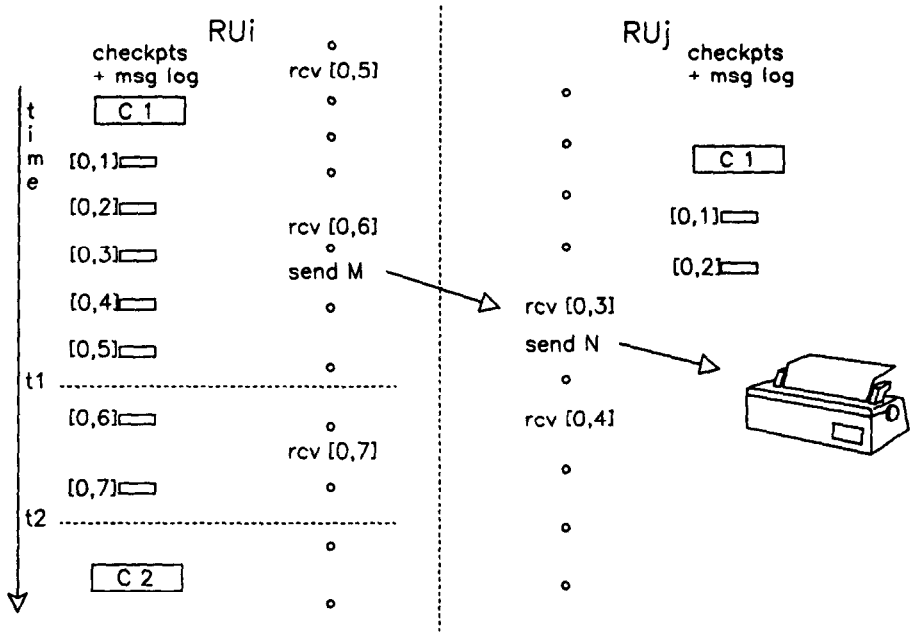

Fig. 7. Sample history of a logical machine: Message M depends on interval [0, 6] of RU_i. The receipt of message M begins interval [0, 3] of RU_j, which now depends on interval [0, 6] of RU_i. Message N depends on both interval [0, 3] of RU_j and interval [0, 6] of RU_i. A failure of RU_i at time t_1 will necessitate a rollback of RU_j, but a failure at time t_2 will not.

2.4 Algorithms

2.4.1 *Example.* We illustrate the algorithms of optimistic recovery with an example (Figure 7) depicting two recovery units RU_i and RU_j. In this example, the sixth message received by RU_i gives rise to state interval [0, 6]. During interval [0, 6] RU_i sends a message M to RU_j. The receipt of M begins state interval [0, 3] of RU_j. Thus, [0, 6] of RU_i is a causal predecessor of [0, 3] of RU_j. As a result of M, RU_j sends message N to a printer outside of the logical machine. We illustrate the operation of the algorithms in the failure-free case and in several possible failure cases.

2.4.2 *Processing Component Algorithms.* In addition to performing computations, the processing component has the responsibility of avoiding impossible states, maintaining the current dependency vector, and labeling the dependencies of each output message.

2.4.2.1 Checking Possible States and Maintaining the Dependency Vector. When the processing component of RU_k dequeues an input message M, thereby beginning a new state interval, it is necessary to check that accepting M does not lead to an impossible state, because the dependency vector encoding is only meaningful for possible states.

The processing component of RU_k maintains a dependency vector $DV_k = \langle [I_1, M_1], [I_2, M_2], \ldots, [I_m, M_m] \rangle$ subject to the following invariant:

—The current state is a possible state.
—The current state is not currently known to be an orphan, that is, for each i,

$$\nexists i \ni \iota > I_i \wedge \mathrm{IST}(\iota, i) \leq M_i.$$

When a new message M with dependency vector $\langle [\iota_1, \mu_1], [\iota_2, \mu_2], \ldots, [\iota_m, \mu_m] \rangle$ is dequeued, it is checked that processing M preserves the above invariant. There are three possible cases:

(1) *Usual case.* For each i,

$$\nexists \iota \ni (I_i < \iota \leq \iota_i \wedge \mathrm{IST}(\iota, i) \leq M_i)$$

and

$$\nexists i \ni (\iota_i < \iota \wedge \mathrm{IST}(\iota, i) \leq \mu_i).$$

The first condition guarantees that if $[I_i, M_i]$ lexicographically precedes $[\iota_i, \mu_i]$, it is also a *live* predecessor and, therefore, that dequeuing the message will lead to a possible state. The second condition guarantees that the new message is not itself known to be an orphan. (If the system has been failure free for some time, then $\iota_i = I_i$, in which case the first test is not needed, and ι_i will be the most recent incarnation of RU_i, in which case the second test is trivially satisfied.)

In the normal case, M is accepted, and DV_k must be updated to reflect the new state dependency, as follows:

$$DV_k(i) \leftarrow \begin{cases} [I_i, M_i + 1] & \text{for } i = k, \\ \max([I_i, M_i], [\iota_i, \mu_i]) & \text{for } i \neq k, \end{cases}$$

where max is defined on pairs using the usual lexicographical ordering.

(2) *M depends on a new incarnation of some recovery unit RU_i with an as yet unknown start number*, that is, such that $\mathrm{IST}(\iota_i, i)$ is undefined. (The message has arrived at RU_k before the incarnation start table has been updated.) Since a new incarnation is known to exist, but its start number is not yet known, it is possible that the current state of RU_k depends on a state of RU_i that has been rolled back. In this case the processing of M is delayed until the recovery message announcing the new incarnation's start number has been received by the recovery manager.

Notice that this situation, which results in a delay of a recovery unit, only occurs in the case in which another recovery unit has failed and is restarting, and does not arise in failure-free operation.

In Figure 7, if RU_i fails at time t_1, it will restore checkpointed state C_1 and replay its first five messages. RU_i will then begin a new incarnation, number 1. If RU_i subsequently sends a message to RU_j, the dependency vector on that message will contain incarnation number 1 in the ith component. If this message arrives at RU_j prior to the arrival of the recovery message containing incarnation 1's start message number, then the processing of the message will be delayed.

(3) *The new message is an orphan.* That is,

$$\exists(\iota, i) \ni \iota_i < \iota \wedge \mathrm{IST}(\iota, i) \leq \mu_i.$$

In this case, the orphan message is discarded.

Orphans will be detected in this way whenever the sending recovery unit has not yet learned that it depends on lost states, but the receiving recovery unit has received a recovery message identifying these states as lost.

Since logging is not synchronized with computation, the orphan may have already been logged. If the message has been logged, the log must be undone. If no other recovery unit has been informed that the message was logged, the message may simply be erased. Otherwise, the recovery unit responds as if it had crashed itself, in order to undo the effects of logging this message; that is, it must begin a new incarnation and send a recovery message.

2.4.2.2 SENDING MESSAGES. The processing component appends the *current value* of the dependency vector to the header of each message it sends.

The overhead associated with tracking dependency is (in a naive implementation) m cells in each message header, where m is the number of recovery units in the logical machine. For logical machines having large numbers of recovery units, there are space-saving optimizations that lower this overhead, such as sending only those dependency vector values that have changed since the last message on this session.

2.4.3 *Half-Session Algorithms.* The half-session of the sender:

(1) appends successive session sequence numbers (SSNs) to each message it sends on the channel. Note that these sequence numbers are relative to a particular channel between two recovery units and are unrelated to message indices, which are relative to a particular recovery unit.
(2) "saves" (i.e., is responsible for reconstructing) each sent message until notified by the receiver that the message has been logged.

It is safe to use volatile storage to save these messages, since, if the sender fails, messages needed by the receiver will be recreated by replay. (As detailed below, the sender always resumes from its *earliest* checkpoint, and no checkpoint is discarded unless it is no longer needed for purposes of regenerating messages not yet logged by the receiver.) Any messages that would not be recreated by replay are orphans, and so are not needed, since the receiver would have had to back them out had it not failed.

The half-session of the receiver:

(1) Notifies the sender of the messages it has logged so that the sender may give up recovery responsibility.
(2) Maintains an "expected" session sequence number and an "expected" sender incarnation number. The receiver compares the actual and expected session sequence numbers on each message it receives. There are three possible cases:

    (a) *Normal case.* The actual and expected session sequence numbers are equal. In this case the half-session passes the message to the merge function and increments its expected session sequence number.

(b) *Sender failed.* The sender's session sequence number is lower than expected. There are two possible actions, depending on whether the sender's incarnation number (which is stored in the message as part of the dependency vector) is greater than the incarnation number expected by the receiver:

(b1) The message's incarnation number is lower than or equal to the expected incarnation number. In this case the message is a duplicate, sent by the recovering unit during replay. Such a message is discarded by the input half-session.

(b2) The message's incarnation number is higher than the expected incarnation number. In this case the sender has completed recovery, and the receiver accepts the message and modifies its expected session sequence number and incarnation number to be the next higher sequence number of the new incarnation.

In the example (Figure 7), suppose that RU_i fails at time t_1. Suppose that just prior to the failure it had sent two messages to RU_j: a message from state [0, 2] with session sequence number 101, and message M from state [0, 6], with session sequence number 102. After RU_i restores checkpoint C_1, it replays state intervals [0, 1] through [0, 5], thereby resending the message with session sequence number 101. Since RU_j expects sequence number 103, and since message 101 arrives from the earlier incarnation, the message is discarded. After replaying, RU_i starts a new incarnation, number 1. The next message sent to RU_j will carry sequence number 102, but incarnation number 1. RU_j will accept this message and reset its expected sequence and incarnation number.

Receiver failed. In this case the session sequence number of the received message will be higher than the expected message. The receiver will retrieve the missing messages by obtaining any logged messages from the log and any unlogged messages from the sender.

The actions of the receiver half-session as a result of comparing session sequence numbers are summarized in Figure 8.

2.4.4 *Output Boundary Function Algorithm.* The *outside world*, that is, any entity outside of the logical machine, is not guaranteed to be able to participate in the optimistic recovery algorithms, since it may be unable to back out computations. Therefore, all output messages with destinations outside the logical machine are buffered in output boundary functions (OBFs), until the states from which they were sent became committable. The following theorem allows us to ascertain when a state interval can be determined to be committable:

THEOREM 1. *Let* $\langle d_1, d_2, \ldots, d_m \rangle$ *be the dependency vector of an arbitrary possible state interval* $[\iota, \mu]$. *Then, if there is a log vector LV such that for each i,* $d_i \leq LV(i)$ *then* $[\iota, \mu]$ *is a committable state.*

PROOF. Suppose that interval $[\iota, \mu]$ were not committable. Then by definition of "committable," at some time in the future, there will exist a lost state interval of one of the recovery units, say, RU_i, on which $[\iota, \mu]$ depends. Call that state

SSN test	Incarnation number	Meaning	Action
actual = expected		normal	accept message
actual > expected		receiver failed	obtain missing messages from log and/or sender
actual < expected	actual = expected	sender failed; message is a duplicate	ignore message
	actual > expected	sender failed; message begins a new incarnation	accept message; modify expected SSN and incarnation number

Fig. 8. Possible actions of a receiver half-session.

interval s_i. By the definition of the dependency vector for possible state intervals, $s_i \preceq d_i$. But, by the definition of the log vector, d_i and all its live predecessors have been logged. Therefore s_i must have been logged, so it cannot have been lost. □

The output boundary function of RU_k may release any messages whose dependency vectors satisfy the condition

$$d_i \preceq LV_k(i) \qquad \text{for all} \quad i.$$

Note that committing requires only local information—the current log vector of the recovery unit committing the message and the message's dependency vector.

In Figure 7, message N is not committable until both message [0, 3] of RU_j and message [0, 6] of RU_i have been logged. Theorem 1 asserts that the commitment can be guaranteed whenever RU_j's message [0, 3] and RU_i's message [0, 6] are both logged.

Data destined for outside the logical machine are subject to a commitment delay that can be affected by (a) the speed with which units log their inputs, and (b) the delay between logging and communicating the log vector.

If the system designer knows that the message is destined for an external entity which itself has rollback capability (e.g., a transaction processing system), then it is possible to improve response time by sending uncommitted messages for early processing by the external entity and later sending it "commit" or "abort" messages, allowing the transaction processing to proceed in parallel with the period of uncertainty about the message. This, in effect, "extends the logical machine" to include the entity with rollback capability.

2.4.5 *Recovery Manager*

2.4.5.1 LOGGING. From time to time, the recovery manager takes a checkpoint of the entire volatile state of the recovery unit. (In practice, there exist optimizations for incremental checkpointing.)

The recovery manager logs all input messages on stable storage in the order in which the merge function determines that they are to be dequeued by the processing component.

As a result of buffering, queuing, and I/O delays, messages may be logged either before or after they are processed by the processing component; *there is no synchronization between logging and processing.* Thus in Figure 7, by time t_1, RU_i has logged its input messages only up to the fifth, but is already processing its sixth input message. Because there is no synchronization, it is possible to perform optimizations in the stable storage I/O that are not possible with synchronous logs, such as blocking several log entries into a single disk track before writing them out.

2.4.5.2 MAINTAINING THE LOG VECTOR. It is necessary to let other recovery units know what has been logged, since this information is used to commit computations to outside the logical machine and to reclaim obsolete checkpoint and log storage. Whereas dependency information must be communicated on every transmission between recovery units in order for the algorithm to work correctly, log vector information (provided it is eventually transmitted) can be sent more infrequently, if desired. However, the sooner the log vector is broadcast to other recovery units, the sooner these recovery units can commit computations, and hence the better the response time observed outside the logical machine.

The log vector is maintained as follows. Each recovery unit, upon a receipt of a new log vector $\langle [\iota_1, M_1], \ldots, [\iota_m, M_m] \rangle$, computes the union of the sets of logged messages indicated by taking the pointwise maximum of the components of the old and new log vectors. Let $LV_k = [I_1, M_1], [I_2, M_2], \ldots, [I_m, M_m]$. Then

$$LV_k(i) \leftarrow max([I_i, M_i], [\iota_i, \mu_i]) \qquad \text{for} \quad i = 1 \cdots m,$$

where max is defined on pairs using lexicographical ordering.

A typical protocol for propagating the log vector would be to have the output half-session piggyback the current log vector on each data message sent out of a recovery unit. In this case, in order to ensure progress, it may be necessary to periodically send a message containing only a log vector on any channel that has remained idle for a sufficiently long time.

2.4.5.3 RECOVERY AFTER FAILURE. After a failure or rollback, a recovery unit restores its earliest checkpoint and replays its log until either an orphan or the end of the log is reached. It then begins a new incarnation by (1) increasing the incarnation number and (2) sending a *recovery message* to other recovery units in the system informing them of the starting message number of the new incarnation. Each input half-session is given an expected session sequence number and incarnation number based on the last processed message received from that half-session.

In our example, if RU_i fails at time t_1, it will restore checkpoint C_1, replay logged messages through [0, 5], and then begin a new incarnation starting at

state [1, 6]. It will send a recovery message informing other recovery units (e.g., RU_j) that the new incarnation has begun, and that states 6 or greater of earlier incarnations (in this case, incarnation number 0) are lost.

When a recovery unit RU_k receives a recovery message announcing the start of a new incarnation ι of some recovery unit RU_i, it updates its incarnation start table by adding a new entry $IST(\iota, i)$. It then examines its current dependency vector to see whether its current state is still a possible state. If the current state of the recovery unit depends on a state that is no longer live, that is,

$$DV_k(i) = [I_i, M_i] \quad \text{and} \quad IST(\iota, i) \leq M_i,$$

then the recovery unit must roll back to an earlier state that does not depend on the no longer live messages of RU_i,

In our example (Figure 7), suppose RU_i fails at time t_1, after RU_j has processed message M, but before RU_i has logged its message [0, 6] on which M depends. After the failure, RU_i will replay through messages [0, 5], and begin a new incarnation with interval [1, 6]. When RU_j receives the recovery message announcing that RU_i has begun incarnation 1 with message number 6, it will examine its dependency vector, which will show a dependency on RU_i's interval [0, 6]. Since this interval is now known to be lost, RU_j will roll back and act exactly as if it had failed. It will restore its checkpoint C_1, replay messages [0, 1] and [0, 2], and then begin a new incarnation of its own, sending other recovery units a recovery message with the new incarnation start number.

2.4.5.4 RECLAIMING CHECKPOINTS AND LOGS. During normal operation, each recovery unit accumulates checkpoint and log records in stable storage.

A recovery unit may discard a checkpoint C_i whenever it knows that it will never be required to recover any interval between C_i and the following checkpoint C_{i+1} either (a) to roll back for the purposes of undoing the effects of orphan messages (*state backout*), or (b) to roll back for the purpose of resending messages lost by a receiving recovery unit that has failed (*message recovery*).

A recovery unit can determine that no interval between checkpoints C_i and C_{i+1} will ever need to be recovered for state backout when C_{i+1}'s state is committable, as determined by its dependency vector and current log vector.

A recovery unit can determine that an interval is no longer needed for message recovery when all messages sent from that interval have been logged by their receiving recovery units. If no interval between C_i and C_{i+1} is needed for message recovery, then C_i is not needed for message recovery.

Whenever C_i may be discarded, all the subsequent log entries up to C_{i+1} can also be discarded.

THEOREM 2. *If a state interval I of RU_k is committable, and if all messages sent by RU_k in intervals preceding I are recoverable, then a systemwide consistent state can thereafter always be found without having to back out interval I.*

PROOF. (1) By definition, state interval I does not depend on any orphan messages. Therefore, it is possible to recover all other recovery units to a point where they have sent any message that RU_k has received from them (i.e., RU_k will never have to be backed out to undo orphans). (2) All messages sent by RU_k to other recovery units are recoverable, so they will eventually all be received

(i.e., RU_k will never have to be backed out for the purpose of message recovery). □

THEOREM 3. *Provided that no recovery unit indefinitely delays* (1) *logging its input,* (2) *transmitting its log vector, and* (3) *taking another checkpoint, then each recovery unit will eventually be able to safely discard its oldest checkpoint.*

PROOF. Any given recovery unit will eventually take a second checkpoint (third assumption). That second checkpoint will depend on some set of states of each of the other recovery units. But the log vector of each of the recovery units will continue to increase monotonically (first assumption), and that fact will be transmitted to the given recovery unit (second assumption). Therefore the log vector will eventually encompass all the states on which the second checkpoint depends, making the state at that second checkpoint committable. Any messages sent between the first and second checkpoint will also eventually be logged. Therefore, eventually both conditions will be met for discarding the oldest checkpoint. □

COROLLARY. *There is no domino effect. This follows from the fact that backout is bounded by the earliest checkpoint that has not been discarded and from the fact that checkpoints are continuously being discarded.*

2.4.5.5 RECLAIMING INCARNATION START TABLES. The recovery manager also reclaims obsolete entries in the incarnation start table. Information for a given incarnation ι of a recovery unit can be discarded when there are no more messages from incarnation $\iota - 1$ left in any of the recovery units or in the channels between them. This fact can be determined by means of a periodic broadcast from each recovery unit of the oldest (smallest) incarnation number for every other recovery unit that resides in its log or in any output message queue for which it has message recovery responsibility. The minimum of these incarnation numbers can be used to determine a bound below which any incarnation number is obsolete.

Since increments of incarnation numbers are rare and the extra storage in the incarnation start tables is negligible, reclaiming incarnation start table entries can be given extremely low priority and requires negligible overhead in normal operation.

2.5 Recovery from Non-Fail-Stop Failures

The fail-stop and independence assumptions are not always met in practice, because (1) some errors are not immediately detected, and processing will continue until the faulty state results in a failure; (2) failures resulting from faulty operating system software will repeat themselves if the identical conditions are restored.

Although the repeatability of software failures is advantageous for debugging in that it is easier to locate the fault by replaying the crash than by examining a postmortem dump, recovery entails *avoiding* the replay of the events leading to the error.

To maximize the chance of full recovery from a software failure, it is necessary to discard as much of recent past history as possible. To do so, we compute for

each RU_i the earliest state index s_i such that no output boundary function committed any message dependent on interval s_i. It is then safe for each recovery unit to roll back, acting as if all messages with indices s_i and later had not been merged. Those of the unmerged messages that depend only on committed states will be retained and remerged, while the rest will be discarded completely. To reduce further the likelihood of repeating the software failure, the nondeterministic merge algorithm can be perturbed, so that remerged messages will be processed in a different order.

Similarly, if a failure is detected after a lapse of time and it is determined that several recent state intervals are invalid, recovery is still possible, provided that no output boundary function has yet committed output that depends on the invalid state intervals. These intervals can be backed out, together with any computations in other recovery units that depend on them.

Although optimistic recovery cannot recover from failures that are detected too late, or from operating system software failures that repeat themselves despite perturbation of the system state, the fact that dependency tracking makes it possible to reconstruct several past consistent states gives optimistic recovery an advantage over recovery systems such as those in [2] and [4], which can reconstruct only the latest consistent state.

3. RELATED WORK

We distinguish the following categories of recovery schemes:
—application-specific recovery,
—transaction-based recovery,
—pessimistic recovery.

3.1 Application-Specific Recovery

In application-specific recovery, recovery is explicitly programmed as part of the application. The recovery code may involve intimate knowledge of both the application domain and of the underlying hardware (see, e.g., [16] for a survey of some such techniques). Small changes in either the application or the underlying hardware may entail substantial redesign of the recovery algorithms.

3.2 Transaction Recovery

In the transaction model [3, 6, 7, 11] computation is divided into units of work called *transactions*. A transaction system is expected to behave as if individual transactions were executed in some serial order (*serializability*), although, to reduce response time, the transactions are actually executed in parallel through multiprogramming or multiprocessing.

Transactions terminate by either *aborting*, or by *committing* their updates to stable storage. In distributed environments, transaction commitment involves synchronous checkpointing ("force-writing") to stable storage by each of the processors at each transaction boundary [12].

The protocols supporting the committing and aborting of transactions are easily extended to handle recovery from machine failures by treating failures as

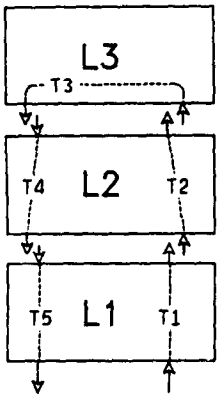

Fig. 9. Short transactions: Commits are frequent, and little concurrency is achievable, since within a given layer each message modifies the same data.

aborts. These protocols can be built into an operating system and can therefore be made transparent to applications.

Unfortunately, not all programs can be expressed as transactions. Transaction systems are based on several assumptions:

—Serializability is required.

—The probability that different transactions contend for the same data is low.

—Transactions are "long enough," that is, involve enough computation to amortize the I/O delays of synchronous commits at each transaction boundary.

For applications that are structured as transactions and that satisfy the above assumptions, transaction-based recovery is likely to be cost effective, since there is little additional overhead involved. However, transactions are only a special case of the more general model of communicating asynchronous processes, for which serializability may be an unnecessary restriction.

Consider, for example, a layered communication protocol, consisting of three layers: L_1, L_2, and L_3. Each layer consists of a process that maintains state information. Messages passing through such a layered protocol typically change the same state information in each layer; for example, the layer sequence number is updated by *each* message. Thus the assumption of low conflict is not satisfied.

We consider two possibilities for defining transaction boundaries in such a system:

(1) *Short transactions.* Passing through a single layer is considered a complete transaction. Since upon completion transactions synchronously force-write information to stable storage, the resulting synchronization delays due to I/O to stable storage at each commit point would be unacceptably high. See Figure 9.

(2) *Long transactions.* A single transaction consists of passing through multiple layers (see Figure 10). Each layer supports messages going out into the network (down) and messages coming in from the network (up). In this case the serializability requirement of the transaction model is overly restrictive. The following sequence of updates is perfectly acceptable for the communication protocol: L_1 (up); L_2 (down); L_2 (up); L_1 (down). However, this sequence is not serializable, because L_2's data sees the transactions in the order "down–up," and

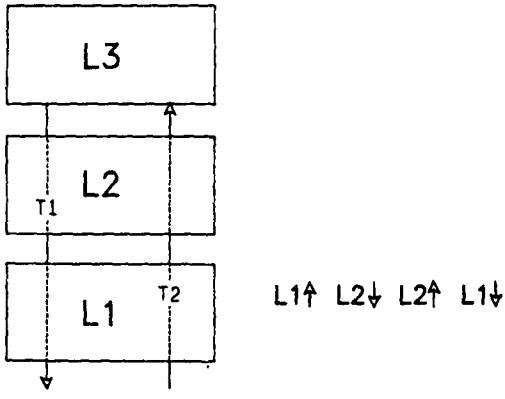

Fig. 10. Long transactions: Transactions span multiple layers in order to reduce the frequency of commits. Serialization imposes delays because interleaving conflicts with serialization.

L_1's data sees the order "up–down." Serialization would substantially reduce the level of concurrency in such a system.

In our view, transactions are best viewed as a technique for efficiently implementing a *single* logically serial process, such as a database manager, by executing its noninterfering parts in parallel and not as a general recovery technique.

3.3 Pessimistic Recovery

Both Tandem NonStop™ [2], and Auragen™ [4]) are systems that support transparent, application-independent recovery. Unlike optimistic recovery, these systems synchronize communication and computation with checkpointing. We call such systems *pessimistic*, since they delay processing each message until both the state of sender and the state of the receiver have been checkpointed to avoid an inconsistency in the rare case of a failure. To avoid the substantial delays associated with checkpointing onto a stable storage medium such as mirrored disks, pessimistic systems typically use a *backup process* on another processor to hold checkpoints.

All communication then requires a multiway synchronization of the primary and backup of both sender and receiver, and multiple failures can no longer be tolerated: If both the primary's processor and the backup's processor fail, recovery is no longer possible.

4. SUMMARY AND CONCLUSIONS

Optimistic recovery is a *transparent* recovery mechanism. That is, applications can be written as if they were to be executed on an ideal failure-free machine. Transparency is important for any system that does not have a single, permanent, hard-wired application. If the application is to be modified, or if new applications are to be written, or if old applications written without recovery in mind are to be run on the system, then transparent recovery will save programming effort and reduce the risk of introducing errors.

Optimistic recovery applies to any system that can be viewed as a collection of recovery units communicating by message passing. It is not restricted, as transaction-based recovery is, to applications that can be structured as units of work accessing a global database to which concurrency control is applied.

We believe that optimistic recovery pays a low price for transparent recovery. Unlike pessimistic recovery techniques, there is no synchronization required upon communication. Therefore, as long as the I/O bandwidth to the disk is sufficiently high, logging delays do not slow down computation. Because logging may be asynchronous, several log entries may be blocked into a buffer and written out in a single I/O operation. Provided that an input message is logged by the time the computation it engendered has completed and is ready to return a result to the external user, there is no response time delay.

Optimistic recovery has the further advantages over pessimistic recovery that a backup processor is not required for checkpointing, that recovery is possible even after the temporary loss of *all* processors, and that some failures not satisfying the fail-stop condition can be recovered.

Optimistic recovery is a special case of *optimistic algorithms* [18]. An optimistic algorithm is one that guesses that an uncertain but highly probable event will happen, and executes "guarded" computations dependent on that assumption. If the assumption should prove true, the optimistic algorithm commits the guarded computations; otherwise, it rolls them back. Optimistic algorithms perform better than their pessimistic counterparts whenever the net gain (the performance improvement when the guess succeeds less the performance loss when the guess fails, weighted by the respective probabilities) is greater than the fixed overhead of tracking dependencies and maintaining rollback capability.

The "guess" in optimistic recovery is that the set of state intervals named in the dependency vector of an input message will be made recoverable before the next failure. The fixed overhead during failure-free operation is

—appending a session sequence number to each message traveling between recovery units and checking it upon arrival;
—maintaining a dependency vector in each recovery unit, copying the vector to the header of each message sent, and updating the dependency vector on each message received;
—periodically checkpointing the full state of each recovery unit and incrementally logging input messages;
—periodically transmitting and updating the log vector;
—buffering messages in the output boundary function until they are committable.

Because the optimistic recovery algorithms gamble that failures will not occur, we expect optimistic recovery to recover somewhat more slowly when failures occur. However, since in most distributed systems failures are very infrequent, we expect optimistic recovery to perform significantly better overall than other recovery techniques.

ACKNOWLEDGMENTS

The authors are indebted to David Jefferson for valuable discussions of our research. Jefferson's own work on virtual time and the time warp mechanism [8] proved to be inspirational for turning our thoughts into precise algorithms. Jefferson's work, while designed for concurrency control and distributed simulation rather than for recovery, has a number of points in common with ours. Our work differs from Jefferson's in our use of a separate time-line for each

recovery unit, together with a partial ordering based on causality rather than a single global time scale as in [9], and in our use of rollback to recover from processor failures.

We wish to thank Marc Auslander and Irving Traiger for useful discussions regarding the implementability of our technique, and Giacomo Cioffi, Arthur Goldberg and Yehuda Afek for valuable criticisms of earlier drafts of this paper.

REFERENCES

Note: References [5], and [17] are not cited in text.

1. AGHILI, H., KIM, W., MCPHERSON, J., SCHKOLNICK, M, AND STRONG, R. A highly available database system. IBM Research Rep. RJ 3755, IBM, Jan. 1983.
2. BARTLETT, J. F. A 'nonstop' operating system. In *11th Hawaii International Conference on System Sciences*. University of Hawaii, 1978.
3. BJORK, L. Recovery scenario for a DB/DC system. In *Proceedings of the ACM Annual Conference* (Atlanta, Ga., Aug. 24–29). ACM, New York, 1973, pp. 142–146.
4. BORG, A., BAUMBACH, J., AND GLAZER, S. A message system supporting fault tolerance. In 9th ACM Symposium on Operating Systems Principles (Bretton Woods, N.H., Oct. 11–13). *Oper. Syst. Rev. 17*, 5 (Oct. 1983), pp. 90–99.
5. CHANDY, K. M., AND LAMPORT L. Distributed snapshots: Determining global states in distributed systems. *ACM Trans. Computer Syst. 3*, 1 (Feb. 1985), 63–75.
6. DAVIES, C. T. Recovery semantics for a DB/DC system. In *Proceedings of the ACM Annual Conference* (Atlanta, Ga., Aug., 24–29). ACM, New York, 1973, pp. 136–141.
7. GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The recovery manager of the System R Database Manager. *ACM Computing Sur. 13*, 2, (June 1981), 223–242.
8. JEFFERSON, D. Virtual time. USC Tech. Rep. TR-83-213, Univ. of Southern California, Los Angeles, May 1983.
9. LAMPORT, L. Time clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, (July 1978), 558–564.
10. LAMPSON, B., AND STURGIS, H. Crash recovery in a distributed storage system. Xerox PARC Tech. Rep., Xerox Palo Alto Research Center, Palo Alto, Calif., Apr. 1979.
11. LISKOV, B., AND SCHEIFLER R., Guardians and actions: Linguistic support for robust distributed programs. In *The 9th Annual Symposium on Principles of Programming Languages* (Albuquerque, New Mex., Jan. 25–27). ACM, New York, 1982, pp. 7–19.
12. MOHAN, C., AND LINDSAY, B. Efficient commit protocols for the tree of processes model of distributed transactions. In *Proceedings of the 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing* (Montreal, Canada, Aug.), 1983, pp. 76–80.
13. MOHAN, C., STRONG, H. R., AND FINKELSTEIN, S. Method for distributed transaction commit and recovery using Byzantine agreement within clusters of processors. IBM Res. Rep. RJ 3882, IBM, San Jose, Calif., June 1983.
14. RUSSELL, D. L. State restoration in systems of communicating processes. *IEEE Trans. Softw. Eng. SE-6*, (2), (Mar. 1980), 193–194.
15. SCHNEIDER, F. B. Fail-stop processors. In *Digest of Papers from Spring Compcon '83* (Mar.). IEEE Computer Society, San Francisco, 1983.
16. SCOTT, R. K., GAULT, J. W., MCALLISTER, D. G., AND WIGGS, J. Experimental validation of six fault-tolerant software reliability models. In *Proceedings of 14th Annual Symposium on Fault-Tolerant Computer Systems* (Kissimmee, Fla., June 20–22). 1984.
17. STROM, R. E., AND YEMINI, S. Optimistic recovery: An asynchronous approach to fault tolerance in distributed systems. *Proceedings of the 14th Annual Symposium on Fault Tolerant Computer Systems* (June 20–22, 1984).
18. STROM, R., AND YEMINI, S. Synthesizing distributed and parallel programs through optimistic transformations. IBM Res. Rep. RC 10797, IBM, 1984.
19. TANNENBAUM, A. S. *Computer Networks*. Prentice-Hall, Englewood Cliffs, N.J., 1981.

Received December 1983; revised February 1985; accepted April 1985