Chapter Three
Message Logging Techniques

Message logging techniques checkpoint process states and (log) interprocess messages onto stable storage. When a process fails and is rolled back, its *message log* is played back to it, so that when the message log has been depleted, the process is in a state consistent with the non-failed processes in the system. Message logging techniques require application processes to be *deterministic*: given a process state and a sequence of inputs (message log), the process will generate the same outputs. Processes can be checkpointed independently, thus minimizing disruptions to the entire system, and a single failed process can be recovered, potentially without interfering with the operation of other processes.

Many papers have been published which use the *message logging* technique as the basis for application-transparent distributed error recovery algorithms [Borg83, Powe83, Stro85, John87, Stro88, John90a, Sist89, Jalo89, Elno92a]. In this chapter a new asynchronous (optimistic) message logging algorithm based on [Stro85] is presented. The computational model described in the previous chapter provides the foundation for a complete proof of correctness of the presented algorithm. This model is also used to thoroughly explain the technical details of asynchronous message logging and to make corrections to previously published algorithms. The new algorithm uses both stable *and* volatile message logs to recover more of the failed process(es)' lost computation. The use of volatile message logs also reduces the number of non-failed processes (orphans) that are forced to roll back in order to recover a consistent system state.

Previous work on message logging algorithms is discussed in Section 3.1. Section 3.2 lists data structures required by asynchronous message logging. Section 3.3 illustrates the basic approach of asynchronous message logging and summarizes the major ideas presented in [Stro85]. In Section 3.4 we show how the presented correctness model is applied to asynchronous message logging. The new algorithm is presented in Section 3.5. The algorithm is discussed and its correctness proven and summarized in Section 3.6. Some relevant details are discussed in Section 3.7. Section 3.8 concludes this chapter. The performance analysis of the algorithm, and some optimizations, using a scalable, execution-driven simulator, running real parallel applications and assuming a failure-free system can be found in Chapter 6.

3.1. Previous Work

Previous work on distributed message logging algorithms can be divided into pessimistic (atomic [Borg83, Powe83], synchronous [John87, Jalo89]) and optimistic (asynchronous [Stro85, Stro88, Sist89]) protocols. In *pessimistic* message logging, the computation of an application and the logging of its messages to stable storage are synchronized. Algorithms relying on *atomic* logging use hardware (e.g. single bus) or software (e.g. reliable multicast) to ensure that both the receiver and stable storage receive the message. *Synchronous* algorithms do not require atomicity but block the sending process from sending any further messages until it has been confirmed that the message sent has been received and logged. *Optimistic* message logging algorithms asynchronously log messages to stable storage without delaying process execution. The principal advantage of

pessimistic schemes is a quicker recovery due to the avoidance of *orphans*. However, any synchronization can significantly degrade the performance of a distributed application.

The synchronous algorithm presented in [John87] logs messages in the volatile memory of the sender. Synchronization on every message is required. When a process sends some message *m* it is not allowed to send any further messages until the corresponding acknowledgment, containing sequencing information, has been received and the sequencing information has been logged (in volatile memory). Due to the use of *volatile message logs* this algorithm can only tolerate the failure of a single process.

The ‘‘pseudo’’-synchronous algorithm in [Jalo89] takes a message checkpoint (some time) *prior* to each send. This message checkpoint contains a copy of all messages received after the previous send and is sent to a backup process on another node in the system. If message delays can be arbitrarily long then a process may have to wait an arbitrarily long time before it can send its next message. In the worst case a receive always occurs just before a send and the full synchronization penalty is incurred. In the best case the ‘‘last’’ receive before the next send occurs far enough ahead of that send that no synchronization penalty is incurred. Thus, ‘‘minimizing’’ synchronization penalties requires that one either correctly guess which receive is the *last* receive or that one take a message checkpoint after *every* receive.

The asynchronous message logging algorithm presented in this thesis is based on the algorithm presented in[Stro85] but is extended to use volatile message logs in addition to stable logs in order to recover as much of the failed process(es)’ pre-

failure execution as possible. As discussed in Sections 3.3 and 3.6.3 there is a critical flaw (omission) in their presented algorithm.

In [John87] volatile logs are used to hold *all* messages since the target machine is assumed to be a collection of diskless workstations. As previously discussed this (exclusive) use of volatile message logs only tolerates the failure of a single process. In [Stro88] volatile logs are used to reduce the amount of messages that must be logged to stable storage. Messages are stored in volatile memory until they are removed or ‘‘spooled’’ to disk. The additional use of volatile logs in this way is orthogonal to the issues discussed in this chapter. Additionally, neither a complete algorithm nor a complete proof was included in [Stro88].

In [John87] the pessimistic algorithm was also extended to be an optimistic algorithm by requiring the sending of dependency information on each message[Stro85]. To be able to keep a single checkpoint per process an orphan-detection algorithm must be run prior to taking a new checkpoint to ensure that the old checkpoint is no longer needed[John87]. This implies that a new checkpoint would be taken only if no orphans were detected. In [John90a] this optimistic algorithm was further extended to keep multiple checkpoints per process. Although a complete optimistic algorithm was not provided in either paper, it appears that this extended algorithm is identical to Strom and Yemini’s optimistic algorithm except that messages are stored in the sender’s volatile memory instead of on stable storage.

In [Sist89] correctness proofs were given for a simplified asynchronous message logging algorithm which requires the rollback of all processes in the system, assumes the loss of all volatile memory, and assumes that at most one

failure occurs. This simplified algorithm does not exhibit the principal advantages of message logging which, in general, requires the checkpointing and rollback of only one process at a time. A second asynchronous algorithm is presented and proven in [Sist89] where the recovery procedure is *always* able to recover the maximal (most recent) global state, but still requires the rollback of the entire system. Unlike other message logging algorithms which require the transmission of $O(n)$ information with every message, this algorithm only requires $O(1)$ information. However, a process must have the ability to take a checkpoint of a *past* state in order to ensure that the maximal (consistent) global state exists on stable storage. An implementation of such a checkpointing scheme requires, for example, 1) the calculation of the consistent state to be checkpointed (requiring $O(n^3)$ messages [Sist89]), 2) the restoration of an older checkpoint, 3) the replay of the stable message log until the chosen state is reached, and 4) the checkpointing of that chosen state. This requires significant computing resources and is more complex and time-consuming than simply saving the *present* process state to stable storage.

A new message logging algorithm, called Manetho[Elno92a], addresses applications that require periodic, and timely, commitment of messages to the external world. In this algorithm dependency tracking involves the (more costly) manipulation and transmission of antecedence (dependency) graphs, instead of vectors, but an external (output) message can be committed after a single synchronous disk operation.

3.2. Data Structures for Asynchronous Message Logging

The following data structures are required to support asynchronous message logging and are maintained for each application process[Stro85]:

- *Incarnation Number, Inc_s* — incremented each time a process s is rolled back.
- *Send Sequence Number, $SSN_s[r]$* — incremented when a message is sent from process s to some process r.
- *Expected Incarnation Number, $eInc_r[s]$* — the incarnation number that a process r expects to receive from s.
- *Expected Send Sequence Number, $eSSN_r[s]$* — the send sequence number that a process r expects to receive from s. This number is incremented after the arrival and acceptance of a message with the expected $SSN_s[r]$.

  Inc_s, $SSN_s[r]$, $eInc_r[s]$, and $eSSN_r[s]$ are used to ensure FIFO ordering of messages across process incarnations.
- *Receive Sequence Number, RSN_r* — incremented and then assigned to a new incoming message.

  The start of a new *state interval* in a process is indicated by the receipt of a message. A state interval is identified by the following triple: (r, Inc_r, RSN_r). RSN_r combined with Inc_r is used as an ‘‘ordering filter’’ that forces determinism upon the nondeterministic arrival of messages. *RSN order* refers to the lexicographical ordering of (Incarnation,RSN) pairs.
- *Dependency Vector, DV_s* — each entry $DV_s[n]$ contains an incarnation

number and RSN ($DV_s[n].Inc$, $DV_s[n].RSN$) which indicates the lexicographically latest state interval of each process n that s's current state interval *depends on* (as defined in Section 2.1), if any.

- *Incarnation Start Table*, IST_s — contains an entry, $IST_s(Inc_n, n) = firstRSN$ for each incarnation of process n as long as any trace of that incarnation remains in the system. Each entry indicates the RSN of the first message that will be (or has been) received from that incarnation.

  This table is used to determine if a process' current state is orphaned and if incoming messages are orphan messages.

- *Log Vector*, LV_s — each entry $LV_s[n]$ contains an incarnation number and RSN ($LV_s[n].Inc$, $LV_s[n].RSN$) which indicates how far process s has been informed that process n has logged.

  Every process periodically broadcasts its log vector entry which indicates the last message it has logged to stable storage.

3.3. Basic Approach of Message Logging

In [Stro85] Strom and Yemini proposed an *optimistic* message logging technique for non-shared-bus architectures where logging proceeds asynchronous of process computation. The recovery of a process' current state depends on its earliest checkpoint and on the disk's log of the messages which the process has received since that checkpoint. When a process fails some messages might not yet be logged to disk. To achieve a consistent system state additional processes (*orphans*) may have to be rolled back. Thus this approach is termed ''optimistic''

because it does not incur the overhead of synchronization during normal operation in order to avoid orphans.

In order to maintain correctness despite the loss of recovery information (messages) sending a message involves the following steps:

1) A message M from sender S to receiver R is sent along with S's current *dependency vector* and $SSN_s[r]$. S saves a copy of M until R notifies S that the message has been logged (step 4).

2) If the SSN of M matches the *expected* SSN ($eSSN_r[s]$ and $eInc_r[s]$) maintained by R, *and* the message is determined not to be an *orphan* then R updates its dependency vector with the dependency vector sent with M since R's state now depends on the state interval in which S sent M.

3) R sends a copy of the message to disk.

4) R ''eventually'' receives notice that the message has been logged and then ''eventually'' notifies S that the message has been logged.

The state of each process q is periodically checkpointed. A checkpoint C_q^i is reclaimed (discarded) when every message upon which the checkpointed state C_q^{i+1} depends has been logged to stable storage *and* when all messages that were sent by q prior to C_q^{i+1} have been logged to stable storage. When a checkpoint C_q^i is reclaimed the log of all messages received prior to and including the checkpoint state C_q^{i+1} can be discarded. In this chapter, *earliest checkpoint* refers to the earliest checkpoint that has not yet been reclaimed.

Messages can be committed to the outside world when *every* message upon

which they depend has been logged. This is determined by comparing the local log vector to the message's dependency vector.

When a process q fails the earliest checkpoint of that failed process is restored. The restored process then plays back its stable log until an orphan or the end of the log is reached. If the last message replayed from the log has RSN order (Inc_q, X), then a new incarnation entry, $IST(Inc_q+1, q) = X+1$, is added to IST_q and is broadcast to the rest of the system in a *Recovery* message. This informs the other processes that any state intervals greater than (Inc_q, X) and less than $(Inc_q+1, X+1)$ are lost. A process p receiving a *Recovery* message updates its IST and determines if it is an orphan by examining its dependency vector and IST to see if its current state depends on any unrecoverable state interval.

This summarizes the main ideas presented in[Stro85]. Although this algorithm detects orphans (processes and messages) and performs *state backout*, it does not always recover/replay all the messages necessary for a correct recovery. The basic flaw is that they do not properly deal with non-orphan messages which are not replayed during the ordered replay phase. Specifically, they do not *guarantee* that such messages will be delivered to the receiver following recovery and this can lead to a lost message inconsistency (the sender state indicates that the message has been sent while the intended receiver never receives the message).

Consider the following example (see Figure 3.1). A process Q fails and is rolled back. There are two messages in Q's stable log but the first one is determined to be an orphan (it depends on a state interval that is known to be lost). The second message is not explicitly replayed from the stable log. When P receives a *Recovery* message from Q, P determines that it is not an orphan (does

not depend on any state intervals known to be lost) and that it has no messages in volatile memory for *Q* (the message had been reclaimed when it was successfully logged to stable storage — steps 1) and 4) above). This is an inconsistent state due to a lost message.

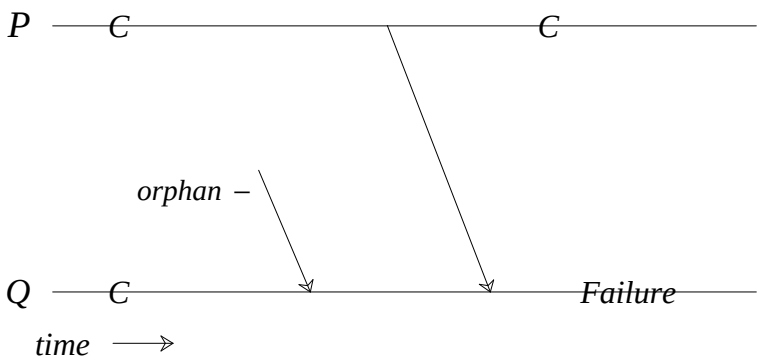

Figure 3.1: An example where a message needed for correct recovery is not retrieved.

It should be noted that this flaw causes inconsistencies only for those sessions where the receiver is the recovering process and where the sender does not send any additional messages to the receiver following recovery. If the sender does send a message to the receiver following recovery, the sender and receiver half-session protocols discussed in[Stro85] will (eventually) correctly deliver the messages to the receiver. Alternatives for solving this problem are briefly discussed in Section 3.6.3. This problem was also overlooked in[Sist89].

3.4. Recovering a Consistent Global State

As described in Section 2.1, a consistent global state is composed entirely of consistent process and channel states such that every pair of processes and the connecting channels ‘‘agree’’ on the number and content of messages which have been exchanged. Ordering information (the depends-on relation) is used to determine if two process states are inconsistent (due to lost or orphan messages) as per Figure 2.1. Since the checkpointing of processes does not affect the ordering of events in the system history, only recovery needs to be examined to show correctness.

Recovery consists of:

a) Restoring the earliest (unreclaimed) checkpoint of the failed process(es), and

b) *Reenacting* the failed process’ previous (pre-failure) behavior as thoroughly as possible. Ideally, enough information exists to reenact the process’ pre-failure execution up to the point at which it had previously failed. If so, no other processes are adversely affected by the failure since no messages have been lost or orphaned.

   If some recovery information has been lost then recovery must also include,

c) *Purging* the effects of any behavior that could not be successfully reenacted due to incomplete recovery information.

With message logging schemes there are no *lost* messages. A non-orphan input message can *always* be retrieved from either — the stable message log, the volatile message log of a non-failed sender, or from the replay of a failed sender

(case 1 in Figure 2.1) [Stro88]. An input message is an *orphan* if there exists a process in which a receive event has occurred for that message and no corresponding send event (exists or) can be reconstructed in some other process (case 2 in Figure 2.1). Orphaned (inconsistent) processes occur due to *incomplete recovery information* which manifests itself through:

a) *Unordered messages:* any message that does not have receive sequence information. This includes any message that has been (or will be) sent that 1) has not been logged to stable storage, *and* 2) has not had its receive sequence information logged in the sender's volatile memory. A message that is logged to stable storage can *become* unordered if, during replay, a logged message that precedes it in RSN order is determined to be an orphan (see b) below). *Ordered messages* are messages that have been successfully logged to stable storage *or* have been logged in the (non-failed) sender's volatile memory along with receive sequence information.

During recovery unordered messages occur when either 1) the sender and receiver fail and the message is not logged, 2) the message was never received in the pre-failure execution, or 3) the message was received but the sender didn't receive ordering information from the receiver. (Note: new messages sent in the post-failure execution which were not sent in the pre-failure execution fall under the definition of unordered messages). Since the ordering information of these messages has been ''lost'' (or never established) it can not be guaranteed that the post-failure execution (recovery) will continue to be identical to the pre-failure execution if these messages are replayed. In other words, the inputs may be processed in a different order and subsequent send events can not be guaranteed to

be reconstructed.

b) *Orphan messages:* a message that depends on some message that is 1) unordered (lost) as the result of a failure(s) or 2) an orphan message. The orphan message itself might be retrievable but since it transitively depends on some message whose ordering information could not be determined it can not be guaranteed that an identical message (or any message at all) will be resent. A process that has received an orphan message is an *orphan process*.

Logging the receive order at the sender *may* result in fewer unordered messages than if the receive order is not sent to the sender as in[Stro85]. This is especially the case if the sending of received messages with their RSNs to stable storage can be delayed at the receiver and lost when the receiver fails. This may be the case if the logging of messages to stable storage is a low-priority task or if messages are blocked together to be logged as one unit. Fewer unordered messages in turn result in fewer orphan messages and orphan processes.

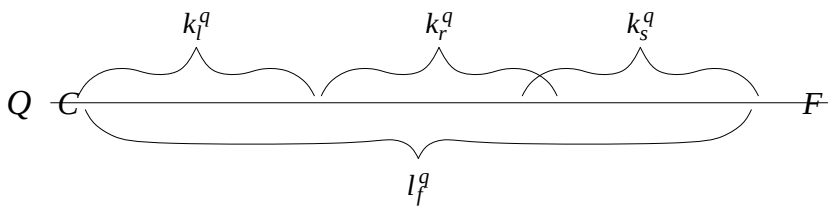

Figure 3.2: $k_f^q=\{k_l^q,k_r^q,k_s^q\}$.

At the time of failure of a process q, k_f^q are the messages that have been *sent to* the failed process q since its earliest checkpoint and l_f^q messages have been *sent from* q.

Messages in k_f^q can be broken down into disjoint groups (see Figure 3.2):

a) *Logged* messages (k_l^q): these messages have been received, assigned an

RSN, and logged to stable storage. As such, it is *guaranteed* that it is possible to obtain copies of the messages and their (pre-failure) RSNs.

b) *Received but not logged* messages (k_r^q): these messages have been received and it *might* be possible to obtain copies of the messages with their (pre-failure) RSNs from somewhere, but there is no guarantee that this is possible (they have not been logged to stable storage).

c) *Sent but not received* messages (k_s^q): these messages have been sent but either no RSNs have been assigned to them yet or, the RSN cannot be obtained from anywhere.

Note: any or all of these groups may be empty.

Once a message is received, an RSN is sent back to the sender. Once a message is sent to receiver q, it becomes a *potential* member (if q fails) of k_f^q and k_s^q. When the sender receives the RSN, if the stable log has not yet received the message with its RSN, the message becomes a potential member of k_r^q. When a copy of the message with its RSN arrives at the stable log, the message becomes a potential member of k_l^q.

If there are several senders, the order in which RSNs are received by the senders may be different from the order in which messages arrive at the receiver (due to concurrency). Hence it is possible for messages to become members of k_r^q while some messages with *lower* RSNs are still in k_s^q. Thus, there may be ‘‘gaps’’ in the RSNs of messages in k_r^q, forcing the end of the deterministic replay phase of recovery.

According to the correctness model presented in Section 2.1, for each

message $m \in k_f^q$ (that was sent at, prior, or even after, the failure of q) either:

1) m is not an orphan and its receipt must be reflected in the post-failure execution of q, or

2) m is an orphan and its sending must be undone (i.e. the sender must be rolled back to some state interval prior to the one in which the message was sent).

Also, messages must be received in the order they were sent, within each pair of processes in spite of the failure (to satisfy FIFO channel assumption).

For each message $m \in l_f^q$ (that was sent by q prior to its failure) either:

1) the message must be regenerated during q's replay. This is guaranteed by the deterministic assumption *as long as* q's inputs are identical to q's pre-failure behavior. These duplicates must not be processed a second time (i.e. must be detected and deleted). or,

2) each message that is not regenerated during q's replay must not be received. This occurs when the input stream diverges from the pre-failure behavior due to an unordered or orphan message. The remaining messages in l_f^q are *orphan messages* and their receipt must be undone.

In other words, every pair of processes must agree which messages have been sent and which have not (such that no messages are lost or duplicated) — thus resulting in a consistent post-failure global state. Ordering information (the depends-on relation) is used to determine if the process states are consistent.

3.5. Algorithms

There are three main differences between the algorithm described in[Stro85] and the new algorithm provided in this chapter. The differences are as follows:

1. As in [Stro85] receive sequence information is logged to stable storage with the message. Unlike[Stro85] this information is also sent back to the sender to be logged in its volatile memory.
2. During recovery, the *entire* stable log is replayed, except for any orphans that are detected. The stable log is then followed by messages logged at the senders and new messages. (In [Stro85] the stable log is explicitly replayed only up to the first orphan that is encountered, as discussed in Section 3.3.)
3. Unlike in[Stro85] steps are included to ensure that the incarnation start table (IST) is robustly maintained.

The first difference is an optimization to reduce the number of orphans (additional processes that will have to roll back). The second is a correction to previous asynchronous algorithms and the third has not been previously specified, but must be included in any asynchronous message logging algorithm.

3.5.1. Normal Operation and Checkpointing

Much of this part of the algorithm is simply reworded from[Stro85], using the data structures defined in Section 3.2. The major change is the sending of the receive sequence information back to the sender. Several minor modifications have been made to either clarify or expand the original description.

The Message Delivery Protocol:

1) A message M from sender S to receiver R is sent along with S's current *dependency vector* and $SSN_s[r]$. S saves a copy of M until R notifies S that the message has been logged (step 4).

2) At receiver R the message must pass through two filters: an *SSN protocol* filter to detect duplicate messages and an *orphan detection* filter to detect orphan messages. (These two filters are provided in the next subsection). If the message successfully passes through the filters then it can be delivered to the application. R updates its own dependency vector with the dependency vector sent along with the message since R's state now depends on the state interval in which S sent M. This is done as follows:

$DV_r[r] \leftarrow (DV_r[r].Inc, DV_r[r].RSN+1)$, and

$DV_r[s] \leftarrow max((DV_r[s].Inc, DV_r[s].RSN),$

$(MsgDV[s].Inc, MsgDV[s].RSN))$, for $s \neq r$

where *max* is the lexicographically largest (incarnation,RSN) pair.

3) R sends the message to disk along with $SSN_s[r]$ and the message's dependency vector (including the receive order in R's $DV_r[r]$ entry and the sender's incarnation number in $DV_r[s]$). R also sends its $DV_r[r]$ entry back to S, so that the sender will temporarily have the message's ordering information.

4) R ''eventually'' receives notice that the message has been logged and then ''eventually'' notifies S that the message has been logged.

Each process q is periodically checkpointed along with its DV_q, SSN_q, $eSSN_q$, and $eInc_q$. Checkpoint C_q^i can be reclaimed when it meets both of the following conditions:

1) No longer needed for state backout:

   if $(LV_q[n].Inc, LV_q[n].RSN) >= (C_q^{i+1}.DV[n].RSN,\ C_q^{i+1}.DV[n].RSN)$ for all processes n.

2) No longer needed for for message recovery:

   all messages sent by q between C_q^i and C_q^{i+1} have been logged in stable storage by the receivers.

3.5.2. Recovery

The recovery of a failed process q consists of the following steps:

1. The earliest checkpoint of q is restored. This checkpoint includes DV_q, SSN_q, $eSSN_q$, $eInc_q$. The current IST_q is also retrieved from stable storage.
2. All ordered messages $m \in k_l^q$ and $m \in k_r^q$ are replayed in *RSNorder*. Playing back k_r^q in *RSNorder* is trivial since messages are logged in *RSNorder*. Playing back k_r^q in *RSNorder* involves the merging of logs obtained from all of the senders. This *ordered playback* ends when:
   a) an orphan message is discovered, or
   b) all messages in k_l^q (and k_r^q) have been replayed. or,
   c) a *gap* in the k_r^q messages occurs: caused by either 1) the ‘‘simultaneous’’ failure of a sender(s) 2) or by the overlap of the k_r^q and k_s^q groups.
3. At this point, since q’s input stream may diverge from its pre-failure execution q:
   a) begins a new incarnation ($DV_q[q].Inc \leftarrow DV_q[q].Inc + 1$),
   b) logs this fact in q’s IST_q ($IST_q(DV_q[q].Inc, q) = DV_q[q].RSN + 1$) on

stable storage and marks this new entry as ‘‘unconfirmed’’.

c) broadcasts *Recovery* messages, containing this new *IST* entry, along with any of q’s *other IST* entries which are (still) marked ‘‘unconfirmed’’, to all other processes p.

4. All remaining messages in k_f^q (which are all unordered messages) are then replayed and (re)logged.

5. Eventually q will receive acknowledgments from all processes p that they have received and logged q’s new entry. q then marks this entry in its IST_q on stable storage as ‘‘confirmed’’.

As discussed above, the second step includes an optimization to previous published techniques and may result in fewer orphan processes. The fourth step corrects the problem discussed in Section 3.3. The robust maintenance of the *IST* is also essential to restoring a valid system state — if the *IST* is missing any entries then a process may not determine that it is an orphan. This issue has not previously been described (see Section 3.7 for more discussion).

When a *Recovery* message is received by p, p:

1. Updates IST_p with the new entry from q. If any of p’s entries in IST_p are marked ‘‘unconfirmed’’ then these are resent to q.

2. Checks to see if its *current* state interval depends on some orphan message:
$\exists\, n \mid (Inc_n > DV_p[n].Inc) \wedge (IST(Inc_n, n) \leq DV_p[n].RSN)$, where $p \neq n$
This check must be performed for *each* one of n’s entries (Inc_n) in the IST. If the process p is determined to be an orphan process then it must be rolled back to its earliest checkpoint.

3. Sends an acknowledgment to q which indicates that p has received the *Recovery* message and has logged q's *IST* entry to stable storage in IST_p.

Every message received by q (and sent by p) is put through the following two filters as mentioned in the previous subsection.

SSN Protocol filter: checks for duplicate messages (re)generated by rolled back processes

a) $SSN_p[q] = eSSN_q[p]$ and $Inc_p = eInc_q[p]$

The message is the next expected message and is passed to the orphan detection filter.

b) $SSN_p[q] < eSSN_q[p]$

There are two possible cases:

1) $Inc_p \leq eInc_q[p]$

The message is a regenerated one and is deleted. or,

2) $Inc_p > eInc_q[p]$

The message is the first message of the new incarnation to be received by q. If it passes the orphan detection filter then q updates both $eInc_q[p]$ and $eSSN_q[p]$.

c) $SSN_p[q] > eSSN_q[p]$

The receiver p has failed and the message is part of q's volatile log k_s^q. The message is held for subsequent replay when $SSN_p[q] = eSSN_q[p]$ (case a).

Orphan detection filter: The message is an orphan and is deleted if the following is true:

$\exists\, p \mid (MsgDV[p].Inc < Inc_p) \wedge (IST(Inc_p, p) \leq MsgDV[p].RSN)$, where

$p \neq q$

This check must be performed for *each* one of p's entries (Inc_p) in the IST.

3.6. Proofs of Correctness

The detection of an orphan or unordered message during replay (step 2 of the recovery algorithm) indicates that complete recovery can not be achieved and further execution *may* diverge from the pre-failure execution of the failed process. All replay *prior* to the first orphan/unordered message has been identical to the pre-failure execution (since the application is assumed to be deterministic) and, hence, any messages sent so far are identical to and compose a prefix of the sequence of messages in l_f^q (See Figure 3.3). If all regenerated messages are either 1) deleted if previously received or 2) received if previously *not* received (due to the failure(s)) then reexecution *up to* the first orphan/unordered message is *correct*. All messages sent have been received with no duplicates or omissions.

All remaining messages in l_f^q *depend on* the orphan/unordered message and are now orphans. Being orphans, these messages have not been (re)sent in the post-failure execution and hence must not be received. Every process that depends on such a message must be rolled back to a state that does not depend on 1) a message that has been unordered as a result of the failure(s) or 2) an orphan message.

To prove the correctness of the above recovery algorithm the next three subsections show, in detail:

1) that regenerated messages (in l_f^q) are detected and deleted,

2) that orphans (in l_f^q) are detected by the receiving processes and their receipt is

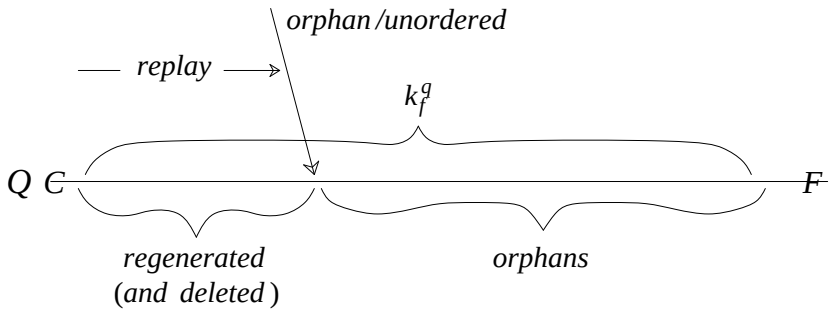

Figure 3.3: Recovery consists of identical replay of the pre-failure execution of failed process Q up to the first detected orphan/unordered message. Since the post-failure execution of Q may now diverge from its pre-failure execution, a new incarnation is begun and all outputs of Q ($m \in l_f^q$) which have not been regenerated during recovery are orphans.

undone — purging the system of all state intervals that depend on any state intervals (of each failed process Q) that could not be successfully reenacted, and

3) that the remaining messages in k_f^q (those that have not yet been replayed) are correctly handled — all non-orphan messages in k_f^q are received and all orphan messages are not sent (or received).

3.6.1. Deleting Regenerated Messages

Theorem 3.1: The SSN protocol filter is sufficient to correctly delete all regenerated messages and accept only non-regenerated messages.

Proof —

a) Assume a regenerated message m sent by p to q is not deleted but is instead accepted.

A regenerated message is, by definition of recovery, sent prior to a *Recovery*

message. In order for the regenerated message to be accepted the (Inc_p,$SSN_p[q]$) pair that arrived with it must have been (lexicographically) ≥ ($eInc_q[p]$,$eSSN_q[p]$). Also, for the message to be *erroneously* accepted q must not have failed (see below) and the message must have been previously received by q. Since p's post-failure behavior prior to sending a *Recovery* message is identical to its pre-failure behavior, then it must have sent the identical Inc_p,$SSN_p[q]$ on the pre-failure version of the message as it has just sent on the regenerated message. In order for q to accept the pre-failure message the following had to be true: $eSSN_q[p] = SSN_p[q]$. Since $eSSN_q[p]$ would have been subsequently incremented, and q has not failed, it is not less than the $SSN_p[q]$ of the regenerated message. (If q *had* failed and rolls back to a state *after* having received the original message, then q's restored $eSSN_q[p]$ must be $> SSN_p[q]$. If q had failed and rolled back to a state *prior* to receiving the message, then accepting the message would not be erroneous).

b) Assume a non-regenerated message m sent by p to q is deleted instead of accepted.

In order for the message to be deleted the following must be true: (Inc_p,$SSN_p[q]$) < ($eInc_{q[p]}$,$eSSN_q[p]$). Since non-regenerated messages are sent after *Recovery* messages (and message delivery is FIFO), all such messages must have the latest, and largest, Inc_p. As a result they are all lexicographically larger than ($eInc_{q[p]}$,$eSSN_q[p]$) since $Inc_p > eInc_q[p]$. The first such message updates the eSSN and eInc values maintained by q. If any subsequent message has a lexicographically smaller send sequence pair then it must be regenerated (case a).

∴ since regenerated messages can not be accepted instead of deleted and non-

regenerated messages can not be deleted, the above algorithm correctly handles all messages. □

In other words, the *SSN* protocol filter preserves the correct ordering of messages across incarnations.

It should be pointed out that when a duplicate message is detected *RSN info* is *not* sent back to the (rolled back) sender (step 3 of the Message Delivery Protocol). This is not important as far as correctness is concerned, since this simply results in an ‘‘extra’’ unordered message in the event of the failure of the receiver prior to the logging of the original (not duplicate) message to stable storage.

3.6.2. Handling Orphan Messages

When a process receives a message it checks whether or not the message is an orphan prior to processing it (e.g. see Lemma 3.1 and Figure 3.4). If the message *is* an orphan it is deleted since, by definition, it depends on some state interval that is no longer part of the system history. The dependency vector transmitted with the message is a recording of the *depends on* relation (Lemma 3.2). If the receiving process knows which state intervals are part of the system history and which are not, for all possible sending processes S, then the receiving process can determine if the incoming message is an orphan.

Lemma 3.1: Any message $m \in k_f^q$ may be an orphan.

Proof —

A way for a message to become an orphan is when, following recovery involving the message sender, the state interval during which the message was sent is no longer part of the live history[Stro85] of the system. This has nothing to do with

whether the message is unlogged (k_s), logged at the sender (k_r), or logged on stable storage (k_l). $\therefore$ any message $m \in k_f^q$ may be an orphan. □

For example, in Figure 3.4 the ordering information for the first message received by P has not been logged at the sender or on stable storage when P fails. Due to concurrency both R and Q may have successfully logged their input messages to stable storage. The messages received by R and Q are orphans because they depend on the first message received by P.

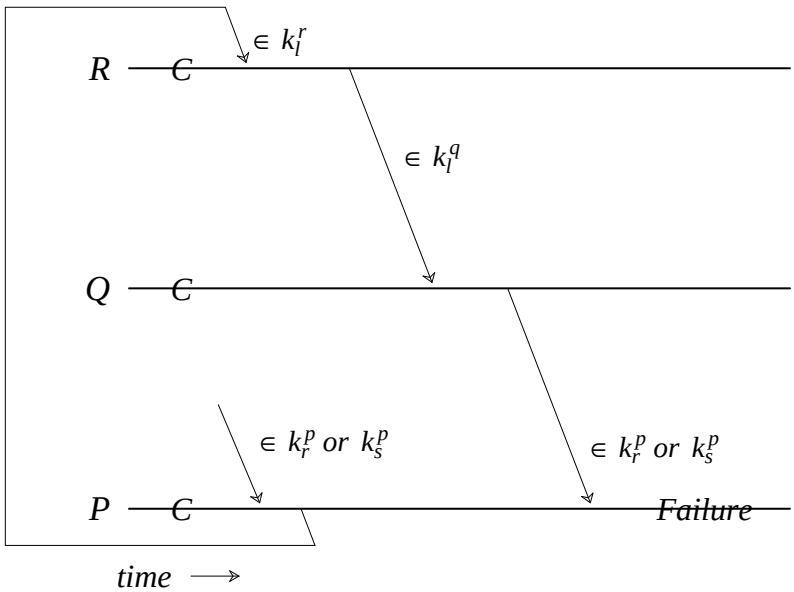

Figure 3.4: The first message received by P can be an unordered message. Hence the messages received by R and Q can be orphan messages.

Lemma 3.2: It is sufficient for a process q to know (record) only the lexicographically latest state interval of each process p upon which it depends.

Proof —

This would be insufficient only if the following were possible: Process q can depend on a state interval of some process p that is in the system history *and* depend on some earlier interval of the same process that is *not* in the system history. This is trivially disproved since a state interval of any process p depends on all previous state intervals of that process (definition of *depends on*).

$\therefore$ the dependency vector correctly and completely records the *depends on* relation. □

Each IST_q records the partial ordering of events for a process q. The system history is defined by the *IST*s of all processes in the system. The following example illustrates how the system history is recorded. Suppose the following messages (inputs) to process q reside on stable storage — listed in (Inc_q,RSN) pairs: (0,0)(0,1)..(0,6),(1,4)..(1,8),(2,9)..(2,13),(3,7)..(3,10). Process q has the following IST entries: $(0,q) = 0$; $(1,q) = 4$; $(2,q) = 9$; $(3,q) = 7$. This means that q received messages one through six in incarnation zero (any subsequent messages in incarnation zero were never logged). Upon rollback message (0,4) was determined to be an orphan and a new incarnation was begun during which messages (1,4)...(1,8) were logged before another rollback was required, and so on. (3,10) was the last message to be logged. The events q contributes to the system history are thus (0,0)..(0,3)(1,4)..(1,6)(3,7)..(3,10). All other events (messages) are ‘‘lost’’ (orphaned) and any state interval that *depends on* one or more of them is an orphan.

This dependency exists due to orphan messages which can manifest themselves in two ways:

a) An *arriving* message is an orphan (i.e. the IST has already been updated), or

b) A message that has already arrived is determined to be an orphan when the

Recovery message arrives (i.e. the IST had *not* been updated when the message arrived).

An *arriving* orphan message is detected by process q if the following is true:

$\exists\, p \mid (MsgDV[p].Inc < Inc_p) \wedge (IST(Inc_p, p) \leq MsgDV[p].RSN)$ [Stro85], where $p \neq q$

This check must be performed for *each* one of p's entries (Inc_p) in the IST. Since, as shown in Lemma 3.1, a message $m \in k_l^q$ can be an orphan message, dependency vectors *must* be logged with the message and its receive order information. This has been overlooked in previous papers and will be addressed again in Section 3.7 .

In addition to checking each message if it is an orphan, whenever a process q receives a *Recovery* message it checks to see if its *current* state interval depends on some unordered/orphan message:

$\exists\, p \mid (Inc_p > DV_q[p].Inc) \wedge (IST(Inc_p, p) \leq DV_q[p].RSN)$ [Stro85], where $p \neq q$

This check must be performed for *each* one of p's entries (Inc_p) in the IST. If the process q is determined to be an orphan process then it must be rolled back to its earliest unreclaimed checkpoint.

Theorem 3.2: Every process that is orphaned by the loss of pre-failure state intervals due to unordered message(s), and no processes that are not orphans, will be rolled back.

Proof —

The post-failure execution is identical to the prefailure execution until the first unordered/orphan message is encountered during the replay of a rolled-back

process. Then a new IST entry is logged to stable storage before any further (re)execution is performed. By definition of the IST, a new entry $IST(Inc_q, q) = RSN$ causes all state intervals greater than $IST(Inc_q - 1, q) = RSN - 1$ to become orphan intervals. The new IST entry is guaranteed to be received by every process because:

1) (*Recovery*) messages are reliably delivered without errors (assumption) and,

2) *Recovery* messages will be resent in the event of the failure of some process(es) p prior to the logging of q's new entry onto the stable storage copy of IST_p. (This is the purpose of ''confirmed'' flag associated with new IST entries on stable storage).

Since a process' dependency vector correctly records which state intervals it depends on (Lemma 3.2) and the ISTs record the system history, the presented recovery algorithm for detecting orphans detects all orphan processes and only orphan processes. □

3.6.3. Messages in k_f^q Following Ordered Playback

The detection of an orphan or unordered message during replay (step 2 of the recovery algorithm) indicates that complete recovery can not be achieved and further execution *may* diverge from the pre-failure execution of the failed process. All replay *prior* to the first orphan/unordered message has been identical to the pre-failure execution. This phase is called *ordered playback*. The remaining messages in k_f^q can be found in stable logs and in sender volatile logs. The issue that needs to be addressed is how to handle these messages. The obvious choices are to discard all of them, to replay all of them, or some combination of the two.

This part of the recovery algorithm has not been discussed in previous message logging papers.

Discarding a message in the log of a recovering process is equivalent to a lost message during normal operation. Hence, in order to ensure consistency, either the sender is rolled back to an interval prior to the sending of the discarded message, or a copy of the message has to be stored (logged) somewhere when it is originally sent so that it can be retransmitted (replayed) as part of recovery.

Consider the first alternative of discarding some of the messages remaining in k_f^q after the ordered playback and rolling back the sender(s) of those messages. There are two problems with this solution: A) how to identify the senders that need to be rolled back, and B) how to ensure that senders do not reclaim the checkpoints that they will need for this purpose. Identifying senders that need to be rolled back requires a way to map messages (those that were found in the logs and discarded) back to their senders. This information can be either maintained at the sender (a list of messages along with the receiver IDs, receiver incarnations and RSNs) or maintained with the message logs (e.g. a sender ID for each message). The second problem with rolling back the senders is that they may need to be rolled back to a checkpoint that has already been reclaimed, by following the checkpoint reclaiming rules described in[Stro85].

The checkpoint reclaiming rules in[Stro85] allow a checkpoint C^i to be reclaimed when 1) it is not needed for state backout (it does not depend on any state interval that has not been logged) and 2) all the messages sent prior to a subsequent checkpoint, C^{i+1}, have been acknowledged as logged to stable storage. However, as discussed in Section 3.3 these checkpoint reclaiming rules may

reclaim a checkpoint that is needed for rolling back a sender of a message discarded from the stable log. Hence, an additional requirement must be met before a checkpoint can be reclaimed. Specifically, C^i can be reclaimed only if all messages sent prior to C^{i+1} are somehow guaranteed not to become orphans. This requires that every message, upon which any one of these messages depends must be logged to stable storage.

The use of sender rollback to solve the lost message inconsistency problem results in additional process rollbacks and extra overhead. Hence, a solution based on replaying all the messages remaining after ordered rollback is preferable. Specifically, *all* non-orphan messages on the stable log are replayed, in RSN order, and then (re)logged to stable storage. In detail, every message $m \in k_f^q$ is either:

- *ordered* — all messages $m \in k_l^q$ and any messages $m \in k_r^q$ retrieved from non-failed senders. or,
- *unordered* — all messages $m \in k_s^q$ retrieved from non-failed senders, any messages $m \in k_l^q$ that follow (RSN order) an orphan message, and any newly sent messages that, therefore, were not in k_f^q. This category includes messages sent by senders that *did* fail.

Assumption: for ease of explanation we assume that either all messages in k_r^q and k_s^q for a particular sender are retrieved (sender hasn't failed) or none of them are (sender has failed). As long as the sender sends these messages in the original SSN order this restriction is easily lifted.

All non-orphan messages in k_f^q, ordered followed by unordered, will be received and only the senders of orphan messages will have to roll back. Thus case 2 requires no extra information to be logged and no additional processes to be

rolled back.

Theorem 3.3: All senders of orphan messages have rolled back or will roll back.

Proof —

By definition of orphan messages —

An orphan message transitively depends on some unordered message. Therefore, the sender of an orphan message must have either:

1) receiv*ed* an orphan message (and is therefore an orphan process which has or will roll back, Thm. 3.2), or

2) detected an unordered message *during recovery* (in which case the process has already rolled back).

Hence, all senders of orphan messages have rolled back or will roll back. □

Theorem 3.4: All non-orphan messages in k_f^q will be received.

Proof —

It follows from Thm. 3.2 that all *orphan* messages will not be received and from Thm. 3.3 that they will not be sent. Therefore we now look only at non-orphan messages.

- All messages in k_f^q are retrieved from the senders unless the sender(s) have failed (assumption), or
- For each sender process p that *has* failed the messages it contributed to k_f^q will either 1) be regenerated or 2) become orphans in the post-failure execution of p.

    1) If the message(s) is regenerated it will be received by q. The regenerated message might, however, be processed by q in a different

lexicographical position in the post-failure (Incarnation,RSN) order compared to its original position in k_f^q. However, this position can only differ with respect to concurrent events (messages from other senders) and not with respect to those messages sent from p (follows from Thm. 3.1 — messages can only be received in monotonically increasing send sequence order).

2) If the message(s) is an orphan then it is no longer in the system history and must not be sent or received. The *sender* has already rolled back — thereby undoing the send. Since the sender has failed:

if the message *is not logged* it can not be retrieved from the failed sender since messages that are not yet logged are stored in volatile memory and are lost during failure/rollback.

if the message *is logged*, the logged orphan can pass through the orphan detection filter only if the IST table entry indicating that it is an orphan has *not* been made (and, therefore, a *Recovery* message has not been received). If this occurs the receiving process will have to roll back (again) to undo the receipt of the logged orphan when the *Recovery* message is eventually received (which is guaranteed by Thm. 3.2). This time the orphan will not be received (the IST *must* have been updated).

$\therefore$ all messages in k_f^q are either retrieved from non-failed senders, are regenerated by failed/rolled back senders, or they become the orphans of the failed sender. Retrieved messages and regenerated messages are received and orphans are not (or, if they are received their receipt is later undone). Hence all non-orphan messages in k_f^q will be received by the post-failure/rollback execution of process q. □

It would appear that the problem of case 1a (reclaiming checkpoints) exists for case 2 as well since messages that are logged may become orphans. This is not the case.

Theorem 3.5: A checkpoint C_q^i can be reclaimed when it is:

1) No longer needed for state backout:

   $(LV_q[p].Inc, LV_q[p].RSN)$ >= $(C_q^{i+1}.DV[p].Inc, C_q^{i+1}.DV[p].RSN)$ for all processes p.

   That is — every state interval upon which the checkpointed state interval C_q^{i+1} depends has been logged. *and,*

2) No longer needed for for message recovery:

   When all messages sent by q between C_q^i and C_q^{i+1} are logged by the receivers to stable storage.

Proof —

1) We have already shown that a dependency vector correctly records a state interval's dependencies (Lemma 3.2). Therefore, if messages are eventually logged and log vectors are periodically updated then C_q^i will eventually not be needed for state backout.

2) Once a message is logged it is always used during replay — unless it is determined to be an orphan (by the orphan detection filter) or a duplicate (by the SSN protocol filter). If the message sent by q is a duplicate then it is clearly not needed for recovery. If the message sent by q can become an orphan then C_q^i is needed for state backout and will not be reclaimed (part 1).

$\therefore$ checkpoint C_q^i can be reclaimed when all messages sent prior to the

checkpointed state have been logged to stable storage and when every state interval upon which C_q^{i+1} depends has been logged. □

3.6.4. Summary

In Section 2.1 we stated that recovery must restore a consistent global state — a system state composed entirely of consistent process (and channel) states such that every pair of processes and the connecting channels ‘‘agree’’ on the number and content of messages which have been exchanged. In the message logging algorithm, for a single process q that fails, the pre-failure messages sent to q (k_f^q) are broken down into three categories: logged (k_l^q), received but not logged (k_r^q), sent but not received (k_s^q). All of the messages in k_f^q are either retrieved from the stable message log (k_l^q), from non-failed senders (some/none/all of k_r^q and k_s^q), or are regenerated by failed senders (unless they are orphans). All *orphan* messages in k_f^q are neither sent (Thm. 3.3) nor received (Thm. 3.2) while all *non-orphan* messages in k_f^q are sent (retrieved from stable storage, from the volatile memory of non-failed senders or from the reexecution of failed senders) and received (Thm. 3.4). Each pre-failure message sent by q (l_f^q) is either regenerated and deleted (Thm. 3.1) or is orphaned and the receiver is rolled back (Thm. 3.2). Thus, all messages sent are received and all messages received are sent such that no messages are lost or duplicated.

3.7. Other Issues

In the presented recovery algorithm, the IST of a process is maintained on stable storage using an optimized distributed commit protocol. A confirmed/unconfirmed flag is added to *q*'s own IST entries so that a failed process *q* can continue recovery (and normal operation) *without* waiting for the replies to the broadcasted *Recovery* message. Thus, the critical path during recovery is 1) retrieving the earliest checkpoint, data structures, and the current IST, 2) replaying ordered messages until an unordered or orphan message is encountered, 3) logging the new IST entry to stable storage and broadcasting the *Recovery* messages, and 4) finishing replay with unordered messages (from stable storage and the senders volatile logs). With this method every process is still guaranteed to eventually receive and log every (new) IST entry. Further optimizations are possible. In any case, robustly updating the IST is critical to any message logging algorithm (in which orphans can occur) since processes must be able to correctly determine whether or not their current state depends on some unrecoverable state interval.

In step 4 of the Recovery algorithm any remaining messages on the stable log are treated as unordered messages (which are not part of the system history) and may later be logged as ordered messages (which are part of the system history). Any logged unordered messages which are determined to be orphans will not be (re)logged. We have not provided any additional mechanism for reclaiming these unordered messages. These messages are reclaimed when, as stated in Section 3.3, the previous checkpoint has been reclaimed. The disadvantage of not reclaiming these unordered and orphan messages earlier is that they will be replayed during

every recovery until they have been reclaimed. These messages could be reclaimed sooner if, during replay:

1) The SSN, sender ID, and sender incarnation number of each message being (re)logged by q is compared to any unordered messages in q's stable log. A match indicates that the unordered message can be reclaimed. and,

2) When an unordered message is determined to be an orphan, a special system message can be sent to reclaim the orphan from the stable log.

We anticipate that very few such messages will exist and that the overhead on stable storage and during recovery will be minimal and therefore not warrant early reclamation.

The issue of committing outputs to the outside world has not been discussed in detail because it is identical to the problem of reclaiming checkpoints. Output commitment can be implemented in a variety of ways. For example, each node in the system can be held responsible for collecting information about the logging progress of every process in log vectors (Section 3.2). When *all* messages (state intervals) upon which the output message depends have been logged, the message may be released [Stro85]. This requires periodic broadcasting of logging progress by all processes. Alternatively ‘‘intelligent’’ disk nodes could keep track of any messages tagged ‘‘output’’ and then release them in a similar fashion. Other methods that attempt to commit the outputs more quickly, with some performance overhead, have been presented [Stro88, Elno92a] (see Chapter 6, Section 6.2.1.4 for a brief discussion).

3.8. Summary

A new asynchronous message logging algorithm for multicomputers has been presented. A correctness model based on[Lamp78, Chan85] was used as the foundation for a complete set of proofs. The presented algorithm is distributed, application-transparent, and n-fault-tolerant. Process checkpoints and interprocess messages are logged to stable storage asynchronous of process execution. Messages and receive sequencing information are also temporarily logged in the local volatile memory of the sender. This allows the algorithm to use volatile message logs in addition to stable message logs to recover as much of the failed process(es)' pre-failure execution as possible. The use of these volatile logs reduces the number of non-failed processes (orphans) which must roll back in order to achieve a consistent system state. In fact, an orphan process Q will occur only when 1) a message m received by P was not successfully logged to disk at the time of P's failure, 2) P sent a message to Q after receiving m and prior to failing, *and* 3) the sender of m does not have a copy of that message (with *RSN*) — i.e. the sender also failed.

One of the major contributions of this chapter is the presentation of a complete proof that this optimized algorithm is *correct* — in the presence of failures the algorithm can be used to recover one or more processes in a distributed application without violating the application's partial ordering of events and without duplicating or deleting any events. Previous papers have given partial proofs or have made strong assumptions that negate the advantages of using message logging as an error recovery technique.

The presented algorithm maintains and improves upon the principal

advantages of the message logging technique by checkpointing and rolling back individual processes whenever possible. Further, it employs no synchronization (pessimism) and is therefore more likely to be able to support applications with different computation-communication ratios. Since the algorithm is n-fault-tolerant and does not assume the existence of any local stable storage it is suitable for use on multicomputers which have multi-tasking nodes and remote storage devices — as long as sufficient extra network bandwidth, disk bandwidth, and disk storage is provided for the message logs and checkpoint states. Chapter 6 contains a performance analysis of this and other distributed application-transparent recovery algorithms appropriate for multicomputers.