Efficient Distributed Recovery Using Message Logging

A. Prasad Sistla and Jennifer L. Welch

GTE Laboratories Incorporated

Abstract: Various distributed algorithms are presented, that allow nodes in a distributed system to recover from crash failures efficiently. The algorithms are independent of the application programs running on the nodes. The algorithms log messages and checkpoint states of the processes to stable storage at each node. Both logging of messages and checkpointing of process states can be done asynchronously with the execution of the application. Upon restarting after a failure, a node initiates a procedure in which the nodes use the logs and checkpoints on stable storage to roll back to earlier local states, such that the resulting global state is *maximal* and *consistent*. The first algorithm requires adding extra information of size $O(n)$ to each application message (where n is the number of nodes); for each failure, $O(n^2)$ messages are exchanged, but no node rolls back more than once. The second algorithm only requires extra information of size O(1) on each application message, but requires $O(n^3)$ messages per failure. Both the above algorithms require that each process should be able to send messages to each of the other processes. We also present algorithms for recovery on networks, in which each process only communicates with its neighbors. Finally, we show how to decompose large networks into smaller networks so that each of the smaller network can use a different recovery procedure.

1. Introduction

Distributed computer systems offer the potential advantages of increased availability and reliability over centralized systems. In order to realize these advantages, we must develop recovery procedures to cope with node failures. For this, the recovery procedures must ensure that the external behavior of the system is unaffected by the failures, that is, that the external behavior of the failure prone system is same as that of a failure free system. Achieving this goal is complicated by the fact that a node failure causes a process to lose the contents of its volatile store and hence its state.

In this paper, we give a precise definition of the recovery problem using the I/O automaton model and present several algorithms to solve this problem. The outline of a formal proof of one of the algorithms is included.

Like many of the standard recovery procedures in the literature, we use the following two techniques: whenever a node restarts after a failure, each of the processes at the different nodes is rolled back to an earlier state using *stable storage*, so that the resulting global state is *consistent*; the external outputs generated by the processes are delayed until it is made sure that the states of processes that generated the outputs will never be rolled back. Roughly speaking, in a consistent global state, if the state of one process reflects the receipt of a message from another process, then the state of the sender process reflects the sending of the message. In order to minimize the roll back for efficiency consideration, the restored global state should be as recent as possible.

There are two approaches for achieving a consistent global state after a failure. One approach is to ensure that

at all times, nodes keep checkpoints (i.e., previous states) in stable storage that are consistent with each other. To obtain the checkpoints, nodes must periodically cooperate in computing a consistent global checkpoint [CL, KT]. Some methods using this approach require suspending the application computation while the checkpoint computation is performed, which is not always feasible in all applications. Also, the more infrequently the checkpoint computation is done, the more out-of-date the checkpoints will be, and thus more work will be lost following a failure.

In the second approach, nodes log incoming messages to stable storage, and after a failure, use these message logs to compute a consistent global state. Algorithms that take this approach can be further classified into those that use pessimistic and those that use optimistic message logging.

In pessimistic (or synchronous) message logging, every message received is logged to stable storage before it is processed [BBG, PP]. Thus the stable information across nodes is always consistent. However, this method slows down every step of the application computation, because of the synchronization needed between logging and processing of incoming messages.

In optimistic (or asynchronous) message logging, messages received by a node are logged in stable storage asynchronously from processing [SY,JZ]. In this case, logging can lag behind processing. Failure-free computation is not disturbed, but some extra work must be done upon recovery to make sure that the restored states are consistent. In [JZ], the authors prove that in such schemes, there is a unique maximal consistent global state that can be recovered from stable storage. Obviously, one would like to recover to this state, in order to undo the minimal amount of the computation performed before the crash.

We present several distributed algorithms, based on asynchronous message logging, that allow nodes to recover to the maximal consistent global state after a failure. This causes a minimal amount of the previous computation to be undone. Our algorithms are correct as long as no further failures occur during the recovery procedure.

The first algorithm requires adding extra information of size $O(n)$ to each application message (where n is the number of nodes); for each failure, $O(n^2)$ messages are exchanged, but no node rolls back more than once. The second algorithm only requires extra information of size $O(1)$ on each application message, but requires $O(n^3)$ messages per failure. The first two algorithms assume the communication network is fully connected. Our third algorithm works in any communication network and only requires processes to communicate with their neighbors. Finally we discuss how to decompose large networks into smaller networks that can use independent recovery procedures.

Other recovery methods based on asynchronous message logging are presented in [SY] and [JZ]. Although our first algorithm is similar to that in [SY], the one presented in [SY] can, in the worst case, cause a process to roll back $O(2^n)$ times, and thus generate an exponential number of messages, in response to a single failure. The algorithm in [JZ] is a centralized one; we believe distributed algorithms, such as ours, are more suited to the nature of this problem.

In Section 2, we give a precise description of the problem. Section 3 contains some definitions about consistent state intervals and message logging. In Section 4, we present the first algorithm together with the proof of correctness. In Section 5, we present our second algorithm. Section 6, discusses extensions to our work for arbitrary networks. The Appendix is a summary of the I/O automaton model [LT], which we use for our formal treatment.

2. Problem Statement

We consider a system of n *nodes* that communicate with each other and with the outside world through messages. Between each ordered pair of distinct nodes there is a message *channel*. The channel delivers messages from one node to the other in the order in which the messages were sent; it does not lose, duplicate or insert messages; each of the messages is delivered after an arbitrary finite delay. We model an arbitrary distributed application program as a set of *application processes* running on the various nodes. The application processes communicate by sending messages. Upon receiving a message, an application process can change its local state in an arbitrary way, as long as it is deterministic, and send messages to the outside world and to other application processes.

In order to define the recovery problem, we consider two systems: an ideal system in which failures do not occur,

which we call the *reliable system* or *RSys*; and the actual system in which failures can occur, which we call the *failure-prone system* or *FSys*.

In a reliable system, each node runs an application process together with a *buffer* process. The buffer process buffers all the incoming messages and delivers them to the application process; it also buffers all the messages generated by the application process and sends them to their destinations, which can be other nodes or the external world.

In a failure-prone system, each node runs an application process and a recovery process. Each of the nodes can crash and then restart. The problem is to design algorithms for the recovery processes so that failure-prone system behaves like the reliable system, as far as the external world is concerned -- that is, for the set of interactions between the failure-prone system and the outside world to be a subset of the set of interactions between the reliable system and the outside world.

The recovery processes can use *stable storage*, storage that is unaffected by failures. In more detail, each node's local state is partitioned into volatile and stable state. After a node crashes, the node's volatile state is initialized, but the stable state is unchanged. We assume for simplicity of presentation that the application process only accesses volatile state. The recovery process acts as a layer around the application process, and filters all messages going into and coming out of the application process. Messages originating in the application process are called *application messages*, and messages originating in the recovery process are called *recovery messages*. Both kinds of messages use the same channels.

The rest of this section formalizes these notions using I/O automata. A brief introduction to I/O automata is given in the appendix. We present each of the components of the system as an I/O automaton. Through out the paper, we use the following definition of *fairness* of an execution. An execution is fair, if whenever an action is enabled in a state of the execution then eventually the action either occurs or gets disabled in the execution.

2.1. Reliable System

We assume that the system consists of a set P of n nodes. For every ordered pair (p,q) of distinct nodes, there is a channel from p to q (See Figure 2-1). The channel from p to q provides FIFO delivery of every message sent from p to q without losing or duplicating or inserting messages. There is no fixed upper bound on the delay between the sending and receipt of a message. Nodes also communicate with the outside world, or environment, directly through messages. Let $P'=P\cup\{\text{env}\}$.

For any two nodes p and q, the automaton Channel(p,q) is defined as follows. Let M be the set of all messages. Intuitively, the state of the automaton is given by a queue which the sequence of messages sent by node p to node q that are not yet received by q. The set of input actions to channel(p,q) consist of actions of the form Send(p,m,q), for all m in M. The effect of the action Send(p,m,q) is to add the message to the queue of messages to be sent. The set of output actions consist of actions of the form Recv(q,m,p), for all m in M. The action Recv(q,m,p) is only enabled in those states in which the message m is at the head of the queue; the effect of this action is to remove m from the head of the queue.

A reliable node p is modeled by a pair of automata, one buffering the incoming and outgoing messages, and the other representing an arbitrary application process (See Figure 2-2). Messages from the outside world and messages from other nodes arrive asynchronously. The buffer automaton stores them in queues and feeds them one-at-a-time to the application process upon request, implementing a nondeterministic merge of all the incoming queues. The buffer automaton and the automaton modeling the application process at node p are denoted by *Buff*(p) and *App*(p) respectively. *Buff*(p) delivers an input to *App*(p) using an action of the form Deliver(p,m,q). The automaton *App*(p) after processing an input communicates with *Buff*(p) using an output action of the form *SendOut*(p,M) where M is an $n+1$ array and $M[q]$ contains the value of the message to be sent to q. The outputs sent by *App*(p) using an action of the form SendOut(p,M), are buffered by *Buff*(p) before they are sent to their destinations.

Let M_E be the set of all messages used for communication between the environment and the nodes, and let M_A be

the set of all messages used for communication between the application processes.

The state of *Buff*(p) is composed of the following variables. For all q (but p) in P' there is a queue *inq*[q] containing all undelivered messages received from q, and there is a queue *outq*[q] that contains all unsent messages to q. All these queues are initially empty. The state also contains a boolean variable *ready* which is initially false. This variable is used to make the Deliver and SendOut actions alternate.

The input actions of *Buff*(p) are the following: In(p,m), for all m in M_E; Recv(p,m,q), for all m in M_A and all q (but p) in P; and SendOut(p,M), for all arrays M such that for all $i \leq n$, $M[i]$ is in M_A and $M[n+1]$, which we denote by M[*env*], is in M_E. The effect of In(p,m) is to add m to *inq*[*env*]. The effect of Recv(p,m,q) is to add m to *inq*[q]. The effect of SendOut(p,M) is to set *ready* to true and add $M[q]$ to *outq*[q] for all q (but p) in P'.

The output actions of *Buff*(p) are the following: Deliver(p,m,q), for all m in $M_A \cup M_E$ and all q (but p) in P'; Send(p,m,q), for all m in M_A and all q (but p) in P; Out(p,m), for all m in M_E. The action Deliver(p,m,q) is enabled only when m is at the head *inq*[q] and *ready*=true; its effect is to remove m from *inq*[q] and set *ready* to false. The action Send(p,m,q) is enabled when m is at the head of *outq*[q], and its effect is to remove m from *outq*[q]. The action Out(p,m) is enabled when m is at the head of *outq*[*env*], and its effect is to remove m from this queue.

The automaton App(p) represents an arbitrary application process which is deterministic and which satisfies the following conditions.

(1) The input actions of App(p) are Deliver(p,m,q), for all m in $M_A \cup M_E$ and all q (but p) in P'; the output actions of App(p) are SendOut(p,M) for all message arrays M from p.

(2) App(p) must preserve well-formedness for p -- a sequence of actions σ of App(p) is defined to be *well-formed for* p if σ is a prefix of the infinite sequence (SendOut(p,.) Deliver(p,.,.))$^{\omega}$. (See Appendix for definition of "preserve".)

Let RSys be the automaton modeling the reliable system, obtained by composing Buff(p), App(p), and Channel(p,q) for all p and q in P, and then hiding all actions except In(p,m) and Out(p,m) for all p in P and all m in M_E. (See Appendix for definition of "hide".)

It is easy to see that Buff(p) preserves well-formedness for p, for all p in P; and every schedule of RSys is well-formed for p, for all p in P.

2.2. Failure-Prone System

Now we consider failures. We assume that nodes can crash but that channels are reliable.

To model a failure-prone node, we replace the automaton Buff(p) with another automaton Recov(p), representing the recovery process. Recov(p) acts as a filter or layer around App(p)(See Figure 2-3).

We must make the following changes to App(p), resulting in an automaton named App'(p). We add more input actions: Crash/Restart(p), which initializes its state, and Restore(p,s) for all states s of App(p), which restores its state to that specified by s. The state sets of App(p) and App'(p) are the same.

Recov(p) must satisfy the following conditions. Let M_R be the set of all messages used for communication between the recovery processes.

(1) The input actions of Recov(p) are In(p,m) for all m in M_E, Recv(p,m,q) for all m in M_R and all q (but p) in P, SendOut(p,M) for all messages arrays M from p, and Crash/Restart(p). The output actions of Recov(p) are Send(p,m,q) for all m in M_R and all q (but p) in P, Out(p,m) for all m in M_E, and Deliver(p,m,q) for all m in $M_A \cup M_E$ and all q (but p) in P'.

(2) The state of Recov(p) is partitioned into volatile and stable. The effect of the Crash/Restart(p) action is to set the volatile part to its initial value and to leave the stable part unchanged.

(3) Let FSys be the automaton modeling the failure-prone system, obtained by composing Recov(p), App'(p), and Channel(p,q) for all p and q in P, and then hiding all actions except In(p,m) and Out(p,m) for all p in P and all m in M_E. We require that for any fair execution e of FSys in which at most one Crash/Restart action occurs, there is a fair execution f of RSys such that $e|ext$(RSys) $= f|ext$(RSys). Thus, the two executions have the same external behavior.

3. State Intervals and Consistency

Given any execution e of RSys, we make the following definitions relative to e.

For each p in P, divide e into *state intervals*: a new state interval begins with each Deliver(p,.,.) action. State intervals are numbered sequentially starting at 0; the number, or *index*, of a state interval s is denoted *index*(s). Suppose state interval s contains Deliver(p,m,q) and SendOut(p,M). Then m is said to *start* s and all the messages in M are said to be *generated* in s.

We define a binary relation *directly depends on* among state intervals of e. Let s and t be state intervals of p and q in e.

- If $p=q$ and *index*(s)$\geq$ *index*(t), then s directly depends on t.
- If $p \neq q$ and s is started by a message generated in t, then s directly depends on t.

We define a binary relation *transitively depends on* among state intervals of e to be the transitive closure of "directly depends on". This is the same as the partial order "happens before" of Lamport [La].

A *global state* of execution e is an n-vector $(i_1, \ldots, i_n)$ such that for all p, i_p is the index of a state interval of p in e. (This is a slight abuse of notation, because the elements of a global state are not local states but are indices; also note that there is no requirement that the collection of state intervals corresponding to the indices be a collection that could all occur at the same time in the execution.)

We define a global state $(i_1, \ldots, i_n)$ to be *consistent* in execution e if for all p, each message delivered to App(p) by the start of p's i_p-th state interval is generated by some q during or before q's i_q-th state interval. It follows easily that global state $(i_1, \ldots, i_n)$ is consistent in e if for all p and q in P, p's i_p-th state interval does not transitively depend on any state interval of q with index $> i_q$.

We define a partial order $\leq$ between global states of execution e as follows. Let $S=(i_1, \ldots, i_n)$ and $T=(j_1, \ldots, j_n)$ be global states of e. We define $S \leq T$ if and only if $i_p \leq j_p$ for all p. In this case, S is said to be *below* T.

The following lemma is from [JZ].

Lemma 1: Fix an execution e of RSys.

(1) All the global states of e form a lattice with respect to the partial order $\leq$.

(2) For a fixed global state R of e, all the consistent global states of e below R form a lattice with respect to $\leq$.

(3) There is a maximum (with respect to $\leq$) consistent global state of e below any fixed global state R of e.

The next definition and lemma are the key to the correctness of our methods of finding the maximum consistent global state. Let $S=(i_1, \ldots, i_n)$ be a global state of execution e. Define max-below(S)$=(j_1, \ldots, j_n)$ as follows. For all p, let j_p be the maximum integer $\leq i_p$ such that, for all q, p's j_p-th state interval does not transitively depend on the (i_q+1)-st state interval of q.

Lemma 2: For any global state S of execution e, max-below(S) is the maximum consistent global state below S.

4. Algorithm With Transitive Dependencies

In this section we describe our first algorithm. Recovery processes maintain transitive dependencies between state intervals of their corresponding application processes, which enables them, after a failure, to find the maximum consistent global state (below the most recent logged state intervals at the time the processes begin the recovery). A tag of size O(n) is added to each application message. After a failure, only O(n^2) recovery messages need to be exchanged, and each application process only needs to roll back once, in order to return the system to the maximum consistent global state.

In Subsection 4.1, we describe the algorithm informally. This version actually does not include checkpointing. Subsection 4.2 contains the formal description of the algorithm and Subsection 4.3 the proof of correctness. We describe our checkpointing mechanism in Subsection 4.4 and discuss an optimization using volatile storage in Subsection 4.5.

4.1. Normal Operation

Recov(p) keeps in volatile storage n queues of incoming messages waiting to be delivered to the application process, one queue for the environment and one for every node other than p. When Recov(p) receives an input from the environment (cf. the In action) or a message from another node (cf. the Recv action for an application message), it adds the message to the end of the appropriate queue.

Recovery processes maintain the transitive dependencies between state intervals of their corresponding application

processes in the following way. Each recovery process p keeps an n-vector TD_p; intuitively, $TD[p]$ is the index of p's current application state interval, and $TD[q]$, $q \neq p$, is the highest index of any state interval of q's application process on which p's current application state interval transitively depends. Initially $TD[p]$ is 0 and the other elements are −1. All application messages generated by p are tagged with the current value of TD. Upon receiving an application message with tag V, p increments $TD[p]$ by 1, and sets $TD[q]$, $q \neq p$, to the maximum of $TD[q]$ and $V[q]$. (The same technique for maintaining transitive dependencies is used in [SY].)

We now describe the interaction between the recovery process and the application process. Once the application process has indicated that it is ready to accept another message (cf. the SendOut action), Recov(p) can deliver to the application process the first message, minus its tag, from one of the queues of incoming messages (cf. the Deliver action when *status* is normal). Then Recov(p) updates its transitive dependency vector and the volatile log recording the order in which messages are delivered. The application process then computes, based on the message just delivered to it, and eventually performs a SendOut action. When a SendOut occurs, Recov(p) tags each message with the current value of the transitive dependency vector and puts it in a queue of outgoing messages for that recipient. The message at the head of an outgoing queue, for any recipient except the environment, is always enabled for sending (cf. the Send action).

No output directed to the environment should occur until it is guaranteed that the state interval that generated this output (and thus the output itself) will never be rolled back. Extra mechanism is needed to ensure this condition. Recov(p) keeps an array N (N for "notified"); $N[q]$ is the maximum state interval of q that p has heard is logged in q's stable storage. Nodes periodically communicate their maximum logged state interval in Notify messages (cf. the Notify action and the Recv action for a Notify message). An Out action can occur once the message is at the head of the output queue and the generating state interval only transitively depends on other state intervals known to be logged.

In order to recover from crashes, which initialize volatile storage, recovery processes make use of stable storage. Periodically, Recov(p) writes the volatile log of delivered messages to a log on stable storage (cf. the Log action). The logging is not synchronized with the receipt or sending of application messages or with the delivery of messages to the application process. In order to avoid losing inputs from the environment, Recov(p) immediately writes each input to another log on stable storage when an In action occurs; a counter is used to keep track of how many inputs have occurred in order to identify the entries in this log. Similarly, in order to avoid duplicating outputs to the environment, Recov(p) immediately writes an indication that an output has occurred to stable storage (in the form of *S_last_out*). (Compacting of these stable logs is discussed in Section 4.4.)

4.2. Handling a Failure

We model a crash followed by a restart as a single action, Crash/Restart. When a node crashes and restarts, its volatile state is initialized. Then its status is set to "recovering" and it sends an Init message to all other nodes with the value of the index of the maximum state interval obtainable from the stable log. Upon receiving an Init message (cf. the Recv action for an Init message), a process broadcasts the index of its latest logged message in a Relay message, changes its status to recovering, and empties all input and output queues as well as the volatile log. Recov(p) collects the values sent in Init and Relay messages into an array L. Once p has received recovery messages from all the processes (so that L is completely filled in), the Restore action is enabled.

After Restore occurs, the application process' state is set to the checkpointed state from stable storage and Recov(p)'s status changes to "replaying" (or back to normal if the stable log is empty). Then Recov(p) feeds successive messages to the application process as before, but the messages are drawn from the stable log instead of the volatile incoming queues (cf. the Deliver action when *status* is replaying). This replaying continues until the end of the stable log or just before reaching a message that is an "orphan" with respect to L. A message with transitive dependency vector V is an *orphan* with respect to L if $V[q] > L[q]$ for some q, i.e., the message was generated in a

state interval that transitively depends on an unlogged state interval. Then the rest of the stable log is discarded, the status is returned to normal, and all the inputs that either were lost from the environment's volatile incoming queue before being delivered or may have accumulated during the recovery/replay procedure are added to the end of the environment's incoming queue.

The In action always adds the input to the stable input log, but only adds the input to the environment's (volatile) queue of incoming messages if the node's status is normal. If the node's status is not normal, then the inputs are collected in the stable log and are added to the end of the volatile queue when replay is complete (as discussed above).

During replay, the application process will generate duplicates of messages and outputs that it generated before the recovery. Duplicate outputs, i.e., outputs that have already occurred, can be detected by comparing the index of the generating state interval with the variable *S_last_out*; duplicates are simply discarded while non-duplicates are added to the outgoing queue. Duplicate messages to other nodes are simply sent on by the recovery process. The recipient's recovery process filters out the duplicates at the point when it is choosing the next message to deliver as follows (cf. the Deliver action when status is normal). Each recovery process keeps a vector of direct dependencies, *DD*, which is updated whenever a message is delivered to the application process. Any message from q to p whose generating state interval index is not greater than $DD[q]$ at p is a duplicate and is discarded by p.

Any application message that is received during the recovery/replay procedure is added to the end of the appropriate incoming queue for later processing, unless the recipient is waiting to receive a Relay message from the sender (in which case the message is discarded).

4.3. Formal Description

We now describe the automaton Recov(p).

STATE:

Volatile:
- $DD[q]$ for all q (but p in P: maximum state interval of q on which p's current state interval directly depends, used to filter duplicate messages; initially −1
- *inq*[q] for all q (but p) in P': FIFO queue of messages received from q and not yet delivered; initially empty
- $L[q]$ for all q in P: maximum state interval of q that is logged, used in recovery; initially 0
- *log*: FIFO queue of messages delivered to application process; initially empty
- $N[q]$ for all q in P: maximum state interval of q that is known to p to be logged, used to commit outputs; initially 0
- *num_in_delivered*: number of inputs (from environment) delivered to application process; initially 0
- *outq*[q] for all q (but p) in P': FIFO queue of messages from p to q waiting to be sent; initially empty
- *ready*: boolean controlling when to deliver next message; initially false
- *restore*: boolean controlling when to restore application state; initially false
- $TD[q]$ for all q in P: maximum state interval of q on which p's current state interval transitively depends; initially $TD[p]$=0 and rest are −1
- *status*: normal, recovering, replaying; initially normal

Stable:
- *S_chkpt*: checkpointed state of application process; initially the start state of App(p)
- *S_DD*: value of *DD* associated with state in *S_chkpt*; initially −1
- *S_inputs*: FIFO queue of inputs from environment that have occurred so far; initially empty
- *S_last_out*: index of state interval of last output that occurred; initially nil
- *S_log*: FIFO queue of messages delivered to application process; initially empty
- *S_num_in*: number of inputs that have occurred so far; initially 0
- *S_num_in_delivered*: number of inputs processed by checkpointed state interval; initially 0
- *S_TD*: value of *TD* associated with state in *S_chkpt*; initially $S\_TD[p]$=0 and rest are −1

Define the derived variable *last_logged_index* to be $S\_TD[p]$ plus the number of entries in *S_log*.

INPUT ACTIONS:

SendOut(p,M) for all message arrays M for p
eff: if *status* = normal **then**
  ready := true
  add ($M[q]$,*TD*) to end of *outq*[q] for all q (but p) in P' with $M[q]$ not empty
endif
if (*status* = replaying) **then**
  ready := true
  add ($M[q]$,*TD*) to end of *outq*[q] for all q (but p) in P with $M[q]$ not empty
  if $M[env]$ not empty **and** $TD[p] > S\_last\_out$ **then**
    add ($M[env]$,*TD*) to end of *outq*[*env*]
  endif

if (no more messages in *S_log*) **or**
(next message in *S_log* is an orphan
with respect to *L*) **then**
discard rest of *S_log*
add *I* to end of *inq*[*env*], where *I* is the suffix
of *S_inputs* since *num_in_delivered*
N[*p*] := *last_logged_index*
status := normal
endif
endif

In(*p*,*m*) for all *m* in M_E
eff: *S_num_in* := *S_num_in*+1
add (*m*,*S_num_in*) to end of *S_inputs*
if *status* = normal **then**
add (*m*,*S_num_in*) to end of *inq*[*env*]
endif

Recv(*p*,*m*,*q*) for all *m* in M_R and all *q* (but *p*) in *P*
eff:
case *m*=(*m'*,*V*), *m'* in M_A:
if *status* = normal **or** *L*[*q*] ≠ nil **then**
add *m* to end of *inq*[*q*]
endif
case *m*= Init(*l*):
L[*q*] := *l*
L[*p*] := *last_logged_index*
status := recovering
N[*r*] := 0 for all *r* in *P*
ready := false
log := empty
inq[*r*] := empty for all *r* (but *p*) in *P'*
outq[*r*] := empty for all *r* (but *p*) in *P'*
if *L*[*r*] ≠ nil for all *r* in *P* **then** *restore* := true **endif**
add Relay(*L*[*p*]) to end of *outq*[*r*] for all *r* (but *p*) in *P*
case *m* = Relay(*l*):
L[*q*] := *l*
if *status* = recovering **and**
L[*r*] ≠ nil for all *r* in *P* **then**
restore := true
endif
case *m* = Notify(*l*):
if *status* = normal **or** *L*[*q*] ≠ nil **then** *N*[*q*] := *l* **endif**

Crash/Restart(*p*)
eff: initialize volatile variables
status := recovering
L[*p*] := *last_logged_index*
add Init(*L*[*p*]) to end of *outq*[*q*] for all *q* (but *p*) in *P*

OUTPUT ACTIONS:

Out(*p*,*m*) for all *m* in M_E
pre: (*m*,*V*) is at head of *outq*[*env*]
V[*q*] ≤ *N*[*q*] for all *q* in *P*
eff: remove (*m*,*V*) from head of *outq*[*env*]
S_last_out := *V*[*p*]

Send(*p*,*m*,*q*) for all *m* in M_R and all *q* (but *p*) in *P*
pre: *m* is at head of *outq*[*q*]
eff: remove *m* from head of *outq*[*q*]

Deliver(*p*,*m*,*q*) for all *m* in $M_A \cup M_E$ and all *q* (but *p*) in *P'*
pre: *ready* = true
status = normal
if *q*=*env* **then**
(*m*,*V*) is at head of *inq*[*env*]
else
(*m*,*V*) is first entry in *inq*[*q*] with *V*[*q*] > *DD*[*q*]
endif
eff: *ready* := false
remove (*m*,*V*) and any skipped entries
from head of *inq*[*q*]
add (*m*,*V*,*q*) to end of *log*
TD[*p*] := *TD*[*p*]+1
if *q*=*env* **then**
num_in_delivered := *V*
else
TD[*r*] := max(*TD*[*r*],*V*[*r*]) for all *r* (but *p*) in *P*
DD[*q*] := *V*[*q*]
endif

Deliver(*p*,*m*,*q*) for all *m* in $M_A \cup M_E$ and all *q* (but *p*) in *P'*
pre: *ready* = true
status = replaying
(*m*,*V*,*q*) is next message in *S_log*
eff: *ready* := false
TD[*p*] := *TD*[*p*]+1
if *q*=*env* **then**
num_in_delivered := *V*
else
TD[*r*] := max(*TD*[*r*],*V*[*r*]) for all *r* (but *p*) in *P*
DD[*q*] := *V*[*q*]
endif

Restore(*p*,*s*) for all states *s* of App'(*p*)
pre: *restore* = true
s = *S_chkpt*
eff: *restore* := false
num_in_delivered := *S_num_in_delivered*
L[*q*] := nil for all *q* in *P*
TD := *S_TD*
DD := *S_DD*
if (*S_log* is empty) **or**
(first message in *S_log* is an orphan
with respect to *L*) **then**
status := normal
discard rest of *S_log*
add *I* to end of *inq*[*env*], where *I* is the suffix of
S_inputs since *num_in_delivered*
N[*p*] := *last_logged_index*
else
status := replaying
endif

INTERNAL ACTIONS:

Log(p)
pre: *status* = normal
log ≠ empty
eff: *log* := empty
add *log* to end of *S_log*
N[*p*] := *last_logged_index*

Notify(p)
pre: *status* = normal
eff: add Notify(*N*[*p*]) to end of *outq*[*q*] for all q (but p) in P

4.4. Correctness

The algorithm just presented is correct provided that no further failures occur while the system is recovering from a previous failure. For ease of notation, the proof in this section assumes that at most one crash occurs in the entire execution, but it can be extended in an obvious way to handle any finite number of failures (as long as no failures occur during the recovery procedure). In the full version of the paper, we will prove that the algorithm can handle an infinite number of failures; this will require a different notion of fairness.

We must show that for every fair execution e of FSys (with at most one Crash/Restart), there is a fair execution e' of RSys such that e and e' have the same sequence of external actions. It is important for e' to be fair in order to rule out useless "solutions" in which no outputs are ever performed in the execution e.

First we show that every execution e (fair or not) of FSys (with at most one Crash/Restart) can be mapped to an execution of RSys with the same external behavior. Then we show that if e is fair, its image under this mapping is also fair.

We use the following method for proving that every execution e of FSys (with at most one Crash/Restart) can be mapped to an execution e' of RSys with the same external behavior. Essentially e' is the result of performing cut-and-paste operations on the original execution, deleting the part of each process' computation that is rolled back after a failure. In order to define e' precisely, we define two mappings, S and A. Roughly speaking, S maps the state of FSys at a given position in e to a state of RSys and A maps the action of FSys at a given position in e to an action of RSys or to the empty string. We obtain a sequence of alternating states and actions of RSys by replacing each state and action in e with its image under S or A and "patching" the holes left when A is empty. To show that this sequence is the desired execution e', we must show that the mappings S and A satisfy certain conditions.

For the rest of this subsection, fix an execution $e=s_0\pi_1 s_1 \ldots$ of FSys in which at most one Crash/Restart occurs. Let *length*(e) be the number of actions in e (it could be infinite).

It is easy to see that each node p executes as follows. First, p computes with normal status, exchanging application messages and Notify messages with other nodes, delivering messages to the application process, receiving inputs and generating outputs. At some point, p either experiences the Crash/Restart or receives an Init message from the node that crashed. Then p changes its status to recovering and sends Relay messages to the other nodes. Once p has received Relay or Init from all other processes, p restores the state of the application process to the checkpoint and replays the messages in the stable log until reaching the end or an orphan. During the replay, the status is replaying. Then the status is set back to normal and p continues in the manner before the crash.

For each p, let l_p be the value of $last\_logged\_index_p$ just before p crashes or receives an Init message, and let v_p be the value of $TD_p[p]$ when *status*(p) switches from replaying to normal, i.e., the index of the state interval to which p recovers.

Lemma 3 states that the *TD* variables correctly track the transitive dependencies between application state intervals. This lemma, together with Lemma 2, is used to prove Theorem 4, which states that the algorithm finds the maximum consistent global state below the last logged indexes (at the time the processes begin the recovery).

Lemma 3: In every state of e, for all p and q, $TD_p[q]$ is the index of the highest state interval of q on which p's current state interval transitively depends.

Theorem 4: The maximum consistent global state below $(l_1, \ldots, l_n)$ is $(v_1, \ldots, v_n)$.

Now we define the mappings. For each p, divide e into five parts:

- Part 1 for p goes from the first state in e to the state immediately before the Crash/Restart(p) action, Recv(p,Init(l),q) action, or the (v_p+1)-st occurrence of a Deliver(p,.,.) action, whichever occurs first.
- Part 2 for p goes from immediately after the end of Part 1

to the state immediately before the Crash/Restart(p) or Recv(p,Init(l),q) action. (It can be empty.)
- Part 3 for p goes from immediately after the end of Part 2 to the state immediately before the Restore(p) action (i.e., it is the time when *status*(p) has value recovering).
- Part 4 for p goes from immediately after the end of Part 3 to the last state in which *status*(p) has value replaying. If the next action in e is SendOut(p,M) that also occurred earlier in e, then this action is included in part 4 for p. (If no states have $status_p$ equal to replaying, then part 4 for p consists solely of the Restore(p) action.)
- Part 5 for p goes from immediately after the end of Part 4 to the end of the execution.

The extra condition in the definition of part 4 is needed to handle the case when part 3 for p begins before p's v_p-th SendOut action occurs. Thus in the replay, the last SendOut is not a duplicate of a previous action from part 1.

The action mapping essentially deletes all actions for p between the time p reaches its v_p-th state interval at the end of part 1 and the time p finishes replaying at the end of part 4, except for In and Out actions, and Send and Recv actions for messages that were either generated in the sender's part 1 or were generated in the sender's part 4 and are not duplicates (i.e., the first copy of this message was lost from the sender's outq at the start of part 3). The state mapping reflects the values of *ready* and App'(p) during parts 1 and 5 for p and freezes them during parts 2, 3, and 4 at their values at the end of part 1. The most complicated part of the state mapping is defining the contents of the message queues to reflect the action mapping correctly.

Let N_A be the set of all integers between 1 and *length*(e) inclusive. Define a mapping A from N_A to *acts*(RSys) $\cup$ {ε}, where ε is the empty string, as follows.
- If π_i is In(p,m) or Out(p,m), then $A(i)=\pi_i$.
- If π_i is Deliver(p,m,q) or SendOut(p,M), then $A(i)=\pi_i$ if π_i is in part 1 or 5 for p and $A(i)=\varepsilon$ otherwise.
- Suppose π_i is Recv(p,(m,V),q). $A(i)$= Recv(p,m,q) if m is generated in q's part 1 or 5 or if m is generated in q's part 4 and there is no preceding π_i action in e. Otherwise $A(i)=\varepsilon$.
- Suppose π_i is Send(p,(m,V),q). $A(i)$= Send(p,m,q) if m is generated in p's part 1 or 5 or if m is generated in p's part 4 and there is no preceding π_i action in e. Otherwise $A(i)=\varepsilon$.
- For any other value of π_i, $A(i)=\varepsilon$.

We add subscripts to the state variables in order to distinguish the same variable at different nodes. Each variable of Buff(p) or Recov(p) is subscripted with p. The channel variable of Channel(p,q) is subscripted with pq.

Let N_S be the set of all integers between 0 and *length*(e) inclusive. Define a mapping S from N_S to *states*(RSys) as follows. For $S(i)$, we must define the state of App(p), as well as values for $channel_{pq}$, $inq_p[q]$, $outq_p[q]$, and $ready_p$, for all p and q.
- App(p): same as App'(p) in s if s_i is in part 1 or 5, otherwise the same as App'(p) in e at the end of part 1.
- $ready_p$: same as $ready_p$ in s if s_i is in part 1 or 5 for p; same as $ready_p$ at end of part 1 for p if s_i is in part 2, 3 or 4 for p.
- $inq_p[q]$, $q \neq env$: Take the sequence of application messages in Recv(p,.,q) actions in e up to s_i, discard any messages generated in q's part 2, discard any messages generated in q's part 4 that are duplicates of any generated in q's part 1, and discard any messages already delivered to App'(p).
- $inq_p[env]$: Take the sequence of messages in In(p,.) actions in e up to s_i and discard any messages delivered in p's part 1.
- $outq_p[q]$, $q \neq env$: Take the sequence of messages for q in SendOut(p,.) actions in e up to s_i, discard any messages generated in p's part 2, 3 or 4, and discard any messages already sent (in a Send(p,(m,V),q) action).
- $outq_p[env]$: Take the sequence of messages for *env* in SendOut(p,.) actions in e up to s_i, discard any messages generated in p's part 2, 3 or 4, and discard any messages already sent (in an Out(p,m) action).
- $channel_{pq}$: Take the sequence of application messages in Send(p,.,q) actions in e up to s_i, discard any messages generated in p's part 2, discard any messages generated in p's part 4 that are duplicates of any generated in p's part 1, and discard any messages already received.

Define a mapping μ from N_S to alternating sequences of states of RSys and actions of RSys inductively as follows. $\mu(0)=S(0)$. If $A(i)$ is empty, then $\mu(i)=\mu(i-1)$, otherwise $\mu(i)=\mu(i-1)A(i)S(i)$. Define e' to be the limit of $\mu(i)$, $i \geq 1$.

Lemma 5 states that the mappings satisfy certain conditions. Lemma 6 shows that consequently e' is the desired execution of RSys.

Lemma 5: (a) $A(i)|ext(\text{RSys}) = \pi_i|ext(\text{FSys})$ for all i in N_A.

(b) $S(0)$ is the start state of RSys.

(c) Choose any i in N_A. If $A(i)$ is empty, then $S(i-1)=S(i)$. If $A(i)$ is not empty, then $(S(i-1),A(i),S(i))$ is a transition of RSys.

Proof: Parts (a) and (b) are true by the definitions of the mappings. We sketch the proof of (c) for two interesting cases.

Case 1: π_i= Out(p,m). So (m,V) is at the head of

$outq_p[env]$ and $V \leq N_p$ in s_{i-1}. We must show that m is at the head of $outq_p[env]$ in $S(i-1)$.

- Suppose π_i is in part 2 for p. We must show that m is generated in part 1 for p. This is equivalent to showing that $V[p] \leq v_p$. Recall that $V \leq N_p$. We can show that in parts 1 and 2 for p, $N_p \leq (l_1, \ldots, l_n)$ (where l_p is the last logged index at the end of part 1 for p). By Theorem 2, $(v_1, \ldots, v_n)$ is the maximum consistent global state below $(l_1, \ldots, l_n)$. By Lemma 1, V is a consistent state and by the above argument $V \leq (l_1, \ldots, l_n)$. Thus $V \leq (v_1, \ldots, v_n)$.
- Suppose π_i is in part 5 for p. The heart of the argument is that p's part 4 mimics part 1 and *S_last_out* correctly filters out duplicate outputs.

Case 2: π_i= Deliver(p,m,q). If $q=env$, then (m,V) is at the head of $inq_p[env]$ in s_{i-1}, and if $q \neq env$, then (m,V) is the first entry in $inq_p[q]$ with $V[q] > DD_p[q]$.

- Suppose i is in part 1 for p. It is enough to show that m is at the head of $inq_p[q]$ in $S(i-1)$. If $q=env$, this is obvious. Suppose $q \neq env$. By Theorem 2, $(v_1, \ldots, v_n)$ is a consistent global state. Thus all messages delivered in part 1 of p are generated in part 1 of q. Thus $V[q] \leq v_q$. The definition of S implies that m is in $inq_p[q]$ in $S(i-1)$. It can be shown that m is at the head of $inq_p[q]$ in s_{i-1}, and thus m is at the head of $inq_p[q]$ in $S(i-1)$.
- Suppose i is in part 5 for p. If $q=env$, then it is enough to show that *S_inputs* and *num_in_delivered* correctly cause $inq_p[env]$ to be set after the replaying. If $q \neq env$, then it is enough to show that the replaying of q mimics the computation of q during part 1 for q.

Lemma 6: e' is an execution of RSys and $e'|ext(\text{RSys}) = e|ext(\text{FSys})$.

Proof: By part (a) of Lemma 5, the external behaviors are the same. Use parts (b) and (c) of Lemma 5 to show inductively that for all i, $\mu(i)$ is an execution of RSys and $S(i)$ is the final state of $\mu(i)$.

We now show that if e is fair, then e' is also fair.

Lemma 7: If e is fair, then e' is fair.

Proof: We must show that once a locally-controlled (i.e., output or internal) action of RSys becomes enabled in e', it eventually occurs or becomes disabled in e'. The argument for Out(p,m) is based on the fact that messages continue to be logged and Notify messages continue to be sent, so that eventually every output at the front of the output queue can be committed.

4.5. Checkpointing

In the algorithm just presented, the stable logs (*S_log* and *S_inputs*) grow without bound, forever increasing. Obviously this causes space problems. In addition, the longer the log is, the more time it takes processes to complete the recovery procedure. These problems can be avoided by periodically summarizing some prefix of the logs in a *checkpointed* state of the application process.

Define an application state interval to be *guaranteed* relative to a state of Recov(p) if the state interval is started by message (m,V) and $V[q] \leq N[q]$ for all q in P. A guaranteed state interval will never be rolled back. Periodically each process p sends a "guarantee" message to each other process q containing the value of $DD_p[q]$ (direct dependency) associated with its maximum guaranteed state interval. Upon receiving a guarantee message with value l from q, process p updates $C_p[q]$ to be equal to l. Process p need never resend to q any message generated in p's state interval l or earlier, since q has all these messages in stable storage and will never roll back past them.

Process p may perform the Chkpt(p) action to compact the logs up to state interval l, where l is computed as follows. Let l be the maximum state interval index such that for all $q \neq p$, either $l \leq C[q]$ or no application message is generated for q in any state interval between $C[q]$ and l. The logs can be compacted up to state interval l because p will never roll back past l, and for every state interval of p up to l, no application message that p sends in that state interval ever needs to be resent. The state of App'(p) corresponding to the l-th state interval, as well as the associated values of the direct and indirect dependency vectors and the number of inputs delivered, are stored in *S_chkpt*, *S_DD*, *S_TD*, and *S_num_in_delivered*.

4.6. Optimization

So far, we have been assuming that during recovery, each recovery process only uses application messages that are logged on stable storage. For each of the non-failed nodes, the incoming messages that have been delivered but not yet logged are available in volatile storage, waiting to be logged. The recovery processes on these nodes can use the messages in volatile storage in addition to those logged on stable storage to recover.

It may not be necessary for the application process on a non-failed node to roll back. In order to decide whether an application process should roll back, the algorithm in Section 4.2 can be modified as follows.

In response to an Init message, each recovery process sends its current state interval index in the Relay message and does *not* discard its volatile log of delivered messages. Once L is filled in, p determines if its current application state interval depends on a state interval $L[q]+1$ of any q, i.e., if $TD[q] > L[q]$. If so, then p restores the application state to the checkpoint and replays the delivered messages logged in stable storage *and* in volatile storage until an orphan is reached.

5. Algorithm Without Transitive Dependencies

In this section we present a recovery procedure that uses direct dependencies instead of transitive dependencies. The key part of the recovery procedure is a distributed algorithm for finding the maximum consistent global state, using the logs and checkpoints at the nodes. When a node restarts after a crash, it invokes the above algorithm so that the application processes can be restored to the appropriate states. This algorithm is also invoked periodically during normal computation, for the purpose of committing output messages and compacting logs.

Every application message sent is tagged with the index of the current state interval of the sender. Whenever a recovery process logs a message, it also logs the tag and the id of the sender with the message.

5.1. Finding the Maximum Consistent Global State

In order to find a maximum consistent global state, the recovery processes use the following algorithm. The algorithm is initiated by one recovery process broadcasting an Initiate message containing the highest index of all messages logged by that recovery process. After receiving the first Initiate message, each recovery process sends the highest index of all its logged messages to all the other processes. After this, recovery process p executes the following. It maintains two n-vectors, $old\text{-}index_p$ and $new\text{-}index_p$, which contain indices of state intervals.

Initially, for each $q \neq p$, $old\text{-}index_{[p}[q]$ contains nil.

1. For each $q \neq p$: wait and receive a value from q into the variable $new\text{-}index_{[p}[q]$.

2. If for each $q \neq p$, $old\text{-}index_p[q] = new\text{-}index_p[q]$, then exit. Starting from the beginning of the log, find the earliest message such that the next message was generated by some process q in a state interval with index $>$ $new\text{-}index_p[q]$. Let L_p be the index of this message. (If there is no such message, let L_p be the index of the last message in the log.) Send L_p to every other recovery process q. Update $old\text{-}index_p$ to be $new\text{-}index_p$.

For all p, let l_p be the index of the last logged state interval at process p when it begins the algorithm (i.e., when it receives the Initiate message for nodes other than the initiator, and when it sends it for the initiator).

Theorem 8: If there are no failures after the initiation of algorithm A0, each recovery process exits the algorithm after at most n iterations, and the values contained in $L_1, \ldots, L_n$ when a recovery process exits the algorithm form the maximum consistent global state below $(l_1, \ldots, l_n)$.

Proof: (Sketch) Consider an execution. We say that the state interval s of process p has *level m dependency*, $m \geq 1$, on state interval t of process q if and only if there exist distinct processes $r_1, \ldots, r_{m+1}$ with respective state intervals $u_1, \ldots, u_{m+1}$, such that $r_1 = q$, $r_{m+1} = p$, $t \leq u_1$, $u_{m+1} = s$, and for all x such that $1 \leq x < m+1$, a message generated in state interval u_x of process r_x is processed by some state interval before u_{x+1} of process r_{x+1}. Level 1 dependency is the same as direct dependency. Note that in an execution of a system with n processes, if a state interval s of some process transitively depends on state interval t of another process, then s has level m dependency on t, for some $m < n$.

It is easy to show that in the above algorithm, after a recovery process executes i iterations, the variable *old-index* gives the maximum state interval index in the log such that all the state intervals below this do not have a level i dependency on the state intervals $l_1, \ldots, l_n$ of processes $1, \ldots, n$ respectively. This clearly shows that the execution of the above procedure terminates within n iterations. From our previous arguments, it follows that L_p after termination is the maximum state interval index of process p within the log such that, for all q, none of the state intervals of process p with index $\leq L_p$ depends on a state interval of process q with index above l_j. Now using Lemma 2, it follows that $(L_1, \ldots, L_n)$ is the maximum consistent global state below $(l_1, \ldots, l_n)$.

5.2. Recovery Procedure

Whenever a node fails and restarts, its recovery process initiates the algorithm of Section 5.1 to compute the maximum consistent global state, and each recovery process restores its application process to this state. After a recovery process receives a first recovery message, it ignores and deletes any more application messages that it receives between successive recovery messages. This serves the purpose of flushing out any dangling application messages that were sent before the recovery started. After a process exits the above procedure, it will replay the log starting from the local checkpoint until the state interval index computed by the previous algorithm is reached. Any messages generated during this procedure will be sent to the respective destination processes. It will also purge the old log beyond the state interval index as computed above. Duplicate messages are handled as in the case of the algorithm in Section 4.

In order to update checkpoints, commit output messages (that is, enabling the output actions associated with the outputs), and compact the log, we use the following procedure. As in the case of the algorithm in section 4, output messages are held in a queue until they are sent out. The processes periodically execute the algorithm to compute the maximum consistent global state. Let $m_1, \ldots, m_n$ be the state interval indices computed by the algorithm of Section 5.1. Now each process p can commit all outputs that were generated before or during the state interval m_p, because process p knows that it cannot roll back to a position before m_p. It will be tempting to conclude that the local checkpoints of the process p can be updated to m_p. However, this can create a problem for the following reason: Suppose process p generated a message for process q before p's m_p-th state interval, and this message was not delivered to App'(q) before q's m_q-th state interval. Although process p will never roll back to a point below m_p, it still needs to keep some of the messages below m_p, in order to regenerate messages of the above kind during possible future recovery.

Let k_p be the maximum state interval index of process p below m_p, such that, for all $q \neq p$, all messages generated by p for process q before k_p are delivered before m_q. Now process p can compact all logged messages below k_p and install a new checkpoint at k_p. In order to compute the values of k_p, the processes exchange another round of messages after computing the maximum consistent global state. In this round, each process q, for all $p \neq q$, finds the most recent message from p that is in the log of q below m_q, and sends to p the state interval index of p on which this message directly depends. Using these state interval indices, process p can determine the value of k_p, and advance the checkpoint to k_p, and get rid of logged messages below k_p.

The above recovery procedure will work properly as long as there are no further failures during the recovery procedure.

6. Extensions

6.1. Networks

In the algorithms presented in sections 4 and 5, each of the processes sends messages to each of the other processes. This may not be a desirable if the number of processes is large and the communication network is not fully connected. We show, below, how the second algorithm can be modified so that each of the recovery processes only communicates with its neighbors on the network.

The key part of the second algorithm is the procedure to compute the maximum consistent global state. We present a different implementation of the algorithm given in Section 5.1. Each process maintains a local variable L. The algorithm is initiated by some process by sending an *initiate* message to all of its neighbors. After receiving the first initiate message, a process sends initiate messages to all its neighbors; any subsequent initiate messages will be ignored by the process. After sending initiate messages to all its neighbors, a process sets L to the maximum state interval index up to which it has logged messages, and then it sends a recovery message with the value of L to all its neighbors. After this, each process repeats the following procedure.

1. **wait and receive** a recovery message from any of the neighbors;

2. **if** the message received in step 1 is from process q and contains value m
   and a message from q, generated in a state interval with

```
index greater than m has been logged then
  set r to the index of the earliest such message,
  if r<L then
    set L to r
    send a recovery message with value L
          to all the neighbors
  endif
endif
```

A global state in which no recovery messages are being processed at any node and no recovery messages are waiting to be processed either in the channels or at the nodes, is called a *quiescent* state. It is easy to show that, during the execution of the above algorithm, whenever the system of processes reach a quiescent state then it remains in this state forever. It is also not difficult to show that, after the initiation of the above algorithm, the processes eventually reach a quiescent state, and in this state, the values of the variables L at the different nodes give the maximum consistent global state. We say that the above algorithm has *terminated* at an instance of its execution if the global state at this instance is a quiescent state. Termination of the above algorithm can be detected using the algorithms of [CM]. After the termination is detected, each of the processes will be notified and they will continue with the remainder of the computation.

As pointed out earlier, the above algorithm for finding the maximum consistent global state can be used to find the global state to which processes roll back after a recovery; it can also be used periodically for committing outputs to the external world.

6.2. Committing Outputs

The model of the reliable system assumes that outputs are buffered before they are sent out. However, another possibility is to assume that outputs are immediately sent to the environment without buffering in the reliable system. In this case, the methods for committing outputs presented in the previous algorithms is inadequate for the following reason. The order in which outputs occur at different nodes in the new model is consistent with the transitive dependency relation. This is not true in the old model, because the outputs are buffered and released independently. To commit outputs correctly in the new model, a node must only commit an output after it has been informed that all outputs generated in the state intervals on which the current state interval transitively depends have occurred.

The following simpler algorithm for committing outputs works whether outputs are buffered or not. Whenever a state interval is committed then the outputs generated in the state interval are committed. In this protocol, the processes use a special type of messages called *committed* messages. Process p commits the state interval s if the following conditions are satisfied: p has logged the message m that initiated the state interval s; if m is generated by process q in its state interval r then p has already received a committed message from q with argument value r; and p has already committed the state interval s–1. Immediately after committing the state interval s, p sends out all the outputs to the external world that were generated during the state interval s, and simultaneously writes the index of s into a location called *max-committed-state* in stable storage so that it will not resend the outputs of s again after a failure. Then it sends committed messages with argument value s to all its neighbors.

It can easily be shown that whenever the above procedure commits the outputs generated during a state interval, that state interval will never be rolled back. The advantage of this approach over the one used in Section 5 is that the processes do not need to compute the maximum consistent global state in order to commit outputs. The disadvantage of this method is that it may take longer to commit outputs than in the case of the previous algorithms; also, the number of messages used are proportional to the number of messages sent during the actual execution of the application processes.

It is also not difficult to see that the maximum committed state intervals at any instance form a consistent global state. We can use these committed state intervals for the purpose of recovery also. Whenever a process restarts after a failure, it recovers to the state given by the variable max-committed-state and sends initiate messages to all the other processes; after receiving the initiate message, each of the other processes recovers to the state given by the max-committed-state. The disadvantage of this recovery method is that the processes may not recover to the maximum consistent global state.

6.3. Decomposing Large Networks

One of the disadvantages of using all the previous approaches to large networks is the following. All these methods may cause a node far removed from the failed node to be rolled back. This may be undesirable and can be avoided using the following approach.

Consider a network N and let P,Q be a partition of the nodes such that the graph of N restricted to each of P,Q is connected. We want to prevent a roll back of an execution of a process in Q due to a failure of a node in P, and vice versa. To achieve this, we use independent recovery algorithms for nodes in P and Q. For each link (p,q), where p is in P and q is in Q, the recovery algorithm for the nodes in P treats all messages sent on this link as outputs to the environment, that is, it sends these messages only when it has made sure that the state interval generating the messages will never be rolled back. The recovery procedure for Q treats messages on the link (p,q) as inputs from the environment, that is, the recovery process at node q logs an input message from p immediately after its arrival and then releases it to the local application process. If the node q fails and restarts, then after the restart, it queries node p for any messages that were sent during the failure period; such messages can be regenerated by p using its log. Also, in order to compact the log at node p, process q may periodically need to send to p the sequence number up to which q has logged the incoming messages from p.

It is easy to see that if the above approach is used, then any failure among the nodes in P will not cause a roll back of any computation at a node in Q, and vice versa. The recovery procedures used for P and Q can be entirely different. Also, if we use the recovery methods given in this paper, then the whole network will be able to function properly even if there are simultaneous failures of a node in P and of a node in Q.

The above approach can be used with respect to any partition $\{P_1,P_2,\ldots,P_m\}$ of the network as long as the graph of the network restricted to each P_i is connected. The partition can be conveniently chosen. For example, if the network is composed of various local area networks connected by gateways then the nodes can be partitioned so that all nodes belonging to the same local area network form one group of the partition.

The disadvantage of this approach is that it may slow down the application computation because messages on links connecting nodes in different groups of the partition must be logged before they are delivered to the application processes. This is the price we pay for shielding the processes in one group of the partition from failures in a different group.

Acknowledgment

We thank Hagit Attiya for stimulating discussions about the contents of this paper.

References

[BBG] Anita Borg, Jim Baumbach, and Sam Glazer, "A Message System Supporting Fault Tolerance," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, October 1983, pp. 90 - 99.

[CL] K. Mani Chandy and Leslie Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," ACM Transactions on Computer Systems, vol. 3, no. 1, February 1985, pp. 63 - 75.

[CM] K. Mani Chandy and Jayadev Misra, "A Paradigm for Detecting Quiescent Properties in Distributed Computations," TR-85-02, Department of Computer Sciences, The University of Texas at Austin, January 1985.

[JZ] David B. Johnson and Willy Zwaenepoel, "Recovery in Distributed Systems Using Asynchronous Message Logging and Checkpointing," *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, August 1988, pp. 171 - 181.

[KT] Richard Koo and Sam Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, January 1987, pp. 23 -31.

[L] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, July 1978, pp. 558 - 564.

[LT] Nancy A. Lynch and Mark R. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms," *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, August 1987, pp. 137 - 151. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-387, April 1987.

[PP] Michael L. Powell and David L. Presotto, "Publishing: A Reliable Broadcast Communication Mechanism,"

Proceedings of the Ninth ACM Symposium on Operating System Principles, October 1983, pp. 100 - 109.

[SY] Robert E. Strom and Shaula Yemini, "Optimistic Recovery in Distributed Systems," *ACM Transactions on Computer Systems*, vol. 3, no. 3, August 1985, pp. 204 - 226.

Appendix

In this Appendix, we review the aspects of the model from [LT] that are relevant to this paper.

An *input-output automaton A* is defined by the following components. (1) There is a (possibly infinite) set of *states* with a subset of *start states*. (2) There is a set of *actions*, partitioned into *input*, *output*, and *internal* actions. The input and output actions are the *external* actions of A, denoted $ext(A)$. (3) The *transition relation* is a set of (state, action, state) triples, such that for any state s' and input action π, there is a transition (s',π,s) for some state s. (Input actions are presumed to originate in the automaton's environment; consequently the automaton must be able to react to them no matter what state it is in.) Action π is *enabled* in state s' if there is a transition (s',π,s) for some state s.

An *execution* e of A is a finite or infinite sequence $s_0\pi_1s_1\ldots$ of alternating states and actions such that s_0 is a start state, (s_{i-1},π_i,s_i) is a transition of A for all i, and if e is finite then e ends with a state. The *schedule* of an execution e is the subsequence of actions appearing in e.

Let A be an automaton and P be a predicate on sequences of actions of A. A *preserves* P if for every schedule βa of A such that P is true of β and a is a locally-controlled action of A, then P is also true of βa.

Automata can be composed to form another automaton. Automata to be composed must have no output actions in common, and the internal actions of each must be disjoint from all the actions of the others. A state of the composite automaton is a tuple of states, one for each component. A start state of the composition has a start state in each component of the state. Any output action of a component becomes an output action of the composition, and similarly for an internal action. An input action of the composition is an action that is input for every component for which it is an action. In a transition of the composition on action π, each component of the state changes as it would in the component automaton if π occurred; if π is not an action of some component automaton, then the corresponding state component does not change.

Given an automaton A and a subset Π of its actions, we can *hide* the actions in Π to form an automaton differing from A only in that each action in Π becomes an internal action. This operation is useful for hiding actions that model interprocess communication in a composite automaton, so that they are no longer visible to the environment of the composition.

Execution e of automaton A is *fair* if for each locally-controlled action π, the following two conditions hold. (1) If e is finite, then π is not enabled in the final state of e. (2) If e is infinite, then either π appears infinitely often in e, or states in which π is not enabled appear infinitely often in e.

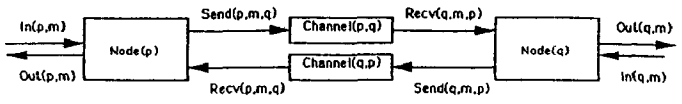

Figure 2-1: Nodes and Channels

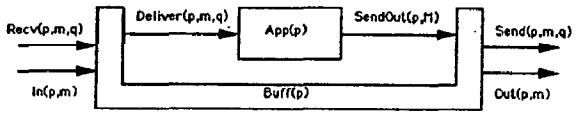

Figure 2-2: Reliable Node

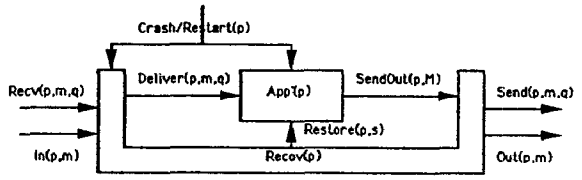

Figure 2-3: Failure-Prone Node with Recovery Process