Optimistic Crash Recovery without Changing Application Messages

S. Venkatesan, *Member, IEEE*, Tony Tong-Ying Juang, and Sridhar Alagar

Abstract—We present an optimistic crash recovery technique without any communication overhead during normal operations of the distributed system. Our technique does not append any information to the application messages, it does not suffer from the domino effect, and each processor rolls back at most once during recovery. We present three distributed rollback algorithms, their complexities, and correctness proofs. Their performances are measured through extensive simulations.

Index Terms—Crash recovery, distributed algorithms, fail-stop failures, message complexity, optimistic message logging, time complexity.

1 Introduction

ROLLBACK recovery using checkpointed states is a widely used scheme for recovering from transient processor failures. Each processor locally checkpoints its state and its history in a stable log at certain times. When a processor fails, it can restart from the latest saved state. There are two approaches to checkpointing and system recovery—the *synchronous* approach and the *asynchronous* approach.

In the synchronous approach (or global checkpointing), the processors coordinate their checkpointing actions such that the global state obtained by collecting the checkpointed states of all of the processors is *consistent* [4], [22], [12]. Each time a checkpoint is taken, additional messages are generated during normal operations even if there are no processor failures.

In the asynchronous approach, processors checkpoint their states independently. A *consistent global state* is constructed during recovery and it may be necessary for some (or all) of the processors in the system to roll back. To aid in minimizing the amount of rollback necessary at each processor, messages are logged using either the *pessimistic message logging* or the *optimistic message logging* approaches. In pessimistic message logging, each message is logged to stable storage before it is processed [3], [15]. In optimistic message logging, the received messages are logged in volatile storage. Periodically (or when the processor is idle), each processor independently saves the contents of its volatile log in a stable log and clears the volatile log.

We present a crash recovery technique using asynchronous checkpointing and optimistic message logging. The technique does not append any information to the application messages. We present three distributed rollback algorithms to determine the maximum recoverable system state after the simultaneous failure of an arbitrary number of processors. Both volatile and stable logs are used in determining the recoverable state. The first rollback algorithm uses $O(md)$ messages and $O(Dd)$ time where m and D are the number of communication channels and the diameter of the network, respectively, and d is a number less than n, the number of processors. The time complexity can be reduced to $O(d + D\log d)$ (resulting in the second rollback algorithm) without increasing the message complexity. The third rollback algorithm, obtained by further refining the first rollback algorithm, is time-optimal, and it uses $O(mn)$ messages and $\Theta(n)$ time. Our recovery algorithms avoid the domino effect [16], [17], and a processor rolls back at most once during recovery. Algorithms for related problems are also presented.

2 System Model

A distributed computing system is represented by an undirected graph $G = (V, E)$ where $V = \{P_1, P_2, \ldots, P_n\}$ represents a set of n fail-stop processors [18] connected by a communication network consisting of a set E of m bidirectional communication channels. The channels are FIFO. Communication between the processors is by message-passing only. The processors and the channels incur unpredictable but finite delays in performing their tasks.

The application program runs uninterrupted when there are no failures. The application program at each processor may

1) receive application messages (called input messages) from outside entities;
2) send application messages (called output messages) to outside entities; and
3) exchange application messages with the application programs running at other processors.

Thus, there are three kinds of application messages.

The application program is suspended when there is a processor failure and the crash recovery algorithm is executed. After the crash recovery algorithm terminates, the

- S. Venkatesan and S. Alagar are with the Computer Science Program, University of Texas at Dallas, Richardson, TX 75083-0688. E-mail: {venky, sridhar}@utdallas.edu.
- T.T-Y. Juang is with the Department of Computer Science, Chung-Hua Polytechnic Institute, Hsin Chu, Taiwan 30067. E-mail: juang@chpi.edu.tw

Manuscript received Apr. 23, 1993.
For information on obtaining reprints of this article, please send e-mail to: transpds@computer.org, and reference IEEECS Log Number D95236.

suspended application program resumes. The crash recovery algorithm uses *recovery messages*. Throughout this paper, the term "message" refers to a recovery message.

The performance of the crash recovery algorithms is measured by the *message complexity* and the *time complexity* of the algorithm. The worst-case message complexity measures the maximum number of recovery messages used. The messages are of length $O(\log n)$ bits. The worst-case time complexity measures the elapsed time among all executions assuming that the message transmission time on each channel is one time unit and the processing time is negligible.

The application program is *event-driven* [7], [21] where a processor in state s waits until an application message m is received, begins a new event, changes its state from s to s', and sends a (possibly empty) set of application messages. A processor's execution during an event is deterministic and is solely dependent on the contents of the message received and on the state of the processor. The jth event of P_i is denoted by e^i_j and the state of P_i immediately after e^i_j is denoted by s^i_j.

Let $\text{SENT}_{i \to j}(e)$ represent the total number of application messages sent by P_i to P_j (from the beginning of the application program) up to (and including) event e of P_i, and let $\text{RECD}_{i \leftarrow j}(e)$ be the total number of application messages received by P_i from P_j (from the beginning of the application program) up to (and including) event e of P_i. A *cut* is a set of events, one event per processor. A cut C is *consistent* if for every pair of events $e_i, e_j \in C$, $\text{SENT}_{i \to j}(e_i) \geq \text{RECD}_{j \leftarrow i}(e_j)$ and $\text{SENT}_{j \to i}(e_j) \geq \text{RECD}_{i \leftarrow j}(e_i)$. A global state is a set of processor states, one state per processor [4]. A global state GS can be represented by a cut C such that event $e \in C$ if and only if the processor state immediately after event e is a part of GS. A global state is consistent if and only if the corresponding cut is consistent. Thus, a consistent global state corresponds to a consistent cut and vice versa.

Event e_j of P_j *directly depends* on event e_i of P_i if

1) $P_i = P_j$ and e_j occurs immediately after e_i, or
2) a message m sent by P_i during event e_i starts event.

Event e_j of P_j *depends* on event e_i of P_i if there exists an event e_k such that e_j depends on e_k and e_k depends on e_i. The relation "depends on" is similar to the "happens before" relation [13].

Problem Statement. The state of a processor is lost if it fails before saving its state. If the state of P_i that has sent a message m to P_j is lost, then for consistency, the state change resulting from the receipt of message m in P_j must be undone. Thus, the state of the P_j must be *rolled back*. The system is said to be in a *maximum consistent state* after recovering from processor failures if the global state of the system is consistent and the number of events (states) rolled back at each processor is minimum. Johnson and Zwaenepoel [7] show that the maximum consistent global state is unique. The crash recovery problem is to find the maximum consistent global state after processor failures. The following assumptions are made in developing the crash recovery techniques:

1) When a processor fails and restarts, all of its neighbors are notified. (A simple message-exchange protocol may be used to achieve this.)
2) No further processor failures occur during crash recovery. (The recovery algorithm may be restarted if there are further processor failures.)
3) Communication channels connecting nonfaulty processors are error-free.

3 Recovery Without Changing Application Messages

We now consider the problem of recovering from processor failures when the application messages do not contain any explicit information about dependencies.

3.1 Normal Operation

During normal execution of the application program (when there are no failures), each processor logs the incoming application/input messages in a volatile log when they are processed by the application program. Processor P_i, after its jth event e^i_j, records the pair $\{m_j, N_j\}$ in volatile storage where m_j is the application message whose receipt starts event e^i_j and N_j is the set of neighbors to whom P_i sends application messages during event e^i_j. If m_j is an input message, then it is logged in the stable storage also. At certain times, each processor independently saves the contents of its volatile log (and its processor state, if needed) in stable storage and clears the volatile log. Note that the processor states may be checkpointed less frequently compared to logging the application messages in stable storage. Also, it is possible to checkpoint only the application messages without saving the processor states. In such a case, to recreate s^i_j, we start with the initial state of P_i and replay all of the application messages (available in stable log) received and processed by P_i till event e^i_j.

$\text{SENT}_{i \to j}$ is incremented during event e if P_i sends a messages during event e, and $\text{RECD}_{i \leftarrow j}$ is incremented during e if a message from P_j initiated event e. Note these arrays are part of P_i's memory and they are saved whenever P_i is checkpointed. The value of these arrays after event e can be computed from the latest checkpointed state before e and from the information ($\{m, N\}$) stored in the volatile log (form the checkpointed events till event e). Thus, these arrays need not be logged/saved after every event.

3.2 Preliminaries

Event e^i_j of P_i is said to be *logged* if it is possible to restore P_i to s^i_j using the application/input messages (and processor states, if applicable) logged in the stable log, and event e^i_j of a faulty processor P_i is said to be *unlogged* otherwise. Thus, if message m that starts event e of P_i is saved in the volatile log but not in stable storage, then e is an unlogged event. Event e of P_i is said to be an *orphan event* if e is an unlogged event or e depends on an unlogged event. Application messages sent during orphan events are *orphan messages*. Note that a logged event may be an orphan event (if it depends on an unlogged event).

Consider processor P_i. The event that occurs immediately before its earliest orphan event is the *rollback point* of P_i, denoted by RP_i. If e_j^i is the rollback point of P_i, then s_j^i is the state P_i must restart from after crash recovery.

3.3 Recovery Algorithms

Crash recovery is performed in two steps. In the first step, the processors cooperate and each processor determines its rollback point. This step is called the rollback step. Note that, in step one, no processor rolls back; the processors merely determine the events (or states) they must roll back to. In the second step, each processor rolls back (recreates the state it must roll back to) using the stable log (and the volatile log if applicable). We first consider the first step. The second step is considered in Section 3.5.

3.3.1 Main Idea

Consider a faulty processor P_1 (see Fig. 1). Let e_l be its latest logged event. The event e_{l+1} immediately after e_l in P_1 is an unlogged event. All other processors have to identify those (orphan) events that are dependent on e_{l+1}.

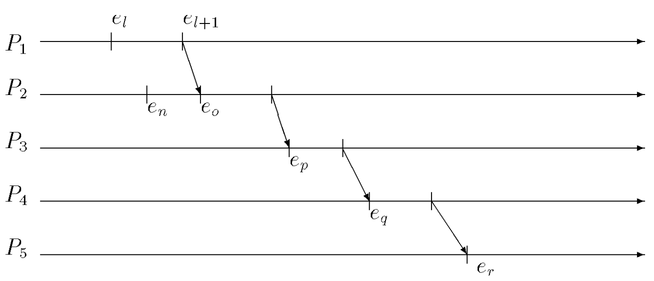


Fig. 1. Transitive dependency through three intermediate processors.

Let P_2 be a neighbor of P_1 and let $x = \text{SENT}_{1\to2}(e_l)$. P_1 sends x to P_2 informing P_2 that P_1 has the knowledge of sending only x messages to P_2. If P_2 has an event e_o such that $\text{RECD}_{2\leftarrow1}(e_o) > x$, then e_o is an orphan event. P_2 identifies all orphan events that depend on e_{l+1}. All the neighbors of P_1 behave similarly and this completes one iteration. Consider the second iteration. Let e_n be the latest nonorphan event of P_2. Processor P_2 sends $\text{SENT}_{2\to3}(e_n)$ to its neighbor P_3. When P_3 receives the count $\text{SENT}_{2\to3}(e_n)$ from P_2, P_3 may identify more orphan events. Similar to P_2, all other neighbors of P_1 also send such counts to their neighbors and they may identify more orphan events. This completes the second iteration. After two iterations, all orphan events that transitively depend on unlogged events of faulty processors through one intermediate processor are identified. Thus, if event e of faulty processor P_i is an unlogged event, event e' of P_j directly depends on e (or e' depends on an event of P_i that occurs after e), and event e'' of P_k directly depends on e' (or e'' depends on an event of P_j that occurs after e'), then,

1) P_i knows about the loss of e when it restarts after its failure;
2) P_j identifies e' to be an orphan event after one iteration; and
3) P_k identifies e'' to be an orphan event after two iterations.

Similarly, after $n - 1$ iterations, all orphan events (that depend on unlogged events through at most $n - 2$ intermediate processors) are identified.

We next present three rollback algorithms. As a preprocessing step, for the first two rollback algorithms, a spanning tree rooted at a designated node is constructed [5]. The third algorithm does not require a spanning tree.

3.3.2 Rollback Algorithm 1

The rollback algorithm (Fig. 2) consists of several iterations. Each processor P_i uses a variable TRP_i to denote its tentative rollback point and a Boolean variable UPDTD_i to indicate whether TRP_i is updated during the current iteration. P_i initializes TRP_i to the latest event logged in the stable storage if it is faulty, and to the latest event of P_i if it is nonfaulty. During the execution of the rollback algorithm, TRP_i is the latest nonorphan event of P_i with the knowledge that P_i has gained so far about orphan messages. The event immediately following TRP_i of P_i, if it exists, is an orphan event. As the rollback algorithm executes, P_i gains more knowledge about dependencies and updates TRP_i. At the end, TRP_i will be equal to RP_i for all $P_i \in V$.

To each neighbor P_j, processor P_i sends an *update* message, waits for an *update* message from each neighbor and processes them. This completes one iteration. During the kth iteration, P_i initializes its local variable UPDTD_i to false, sends an *update*(x) message to its neighbor P_j where $x = \text{SENT}_{i\to j}(\text{TRP}_i)$, and waits for an *update* message from each neighbor. The value x of the *update*(x) message sent by P_i to P_j identifies TRP_i implicitly. When P_j receives an *update*(x) message from P_i, processor P_j infers that P_i's tentative rollback point is an event e such that x is the total number of application messages sent by P_i to P_j till P_i's event e. If the number of messages received by P_j from P_i till P_j's tentative rollback point (TRP_j) is greater than x, then P_j's tentative rollback point depends on an orphan event of P_i, and hence P_j must update TRP_j.

P_i processes the *update* messages sent by its neighbors as follows. Let *update*(c) be a message received by P_i from its neighbor P_j. Processor P_i scans its log and determines $\text{RECD}_{i\leftarrow j}(\text{TRP}_i)$. If $\text{RECD}_{i\leftarrow j}(\text{TRP}_i) > c$, then P_i examines its log, finds the *latest* event e such that $\text{RECD}_{i\leftarrow j}(e) = c$, and sets TRP_i to e and UPDTD_i to true. On the other hand, if

$\text{RECD}_{i \leftarrow j}(\text{TRP}_i) \le c$, P_i need not update TRP_i in response and the value of UPDTD_i remains unchanged. In this manner, all of the *update* messages are processed and TRP_i is updated. After processing the *update* messages received from all of its neighbors, P_i completes the current iteration and proceeds to the next iteration if needed.

Termination. The rollback algorithm terminates if no processor updates its tentative rollback point during the present iteration. Clearly, UPDTD_i is set to true only if P_i updates TRP_i during the current iteration. Boolean OR of UPDTD_1, UPDTD_2, …, UPDTD_n is computed by using a spanning tree T rooted at, say, P_r and the result is broadcast as follows. Each leaf P_i of T sends UPDTD_i to its parent. A nonleaf processor P_j receives Boolean values from all of its children, computes Boolean OR of UPDTD_j and the values received from its children, and sends the computed value to its parent if P_j is not the root; if P_j is the root, then it broadcasts (on the tree T) the value computed by the function *Boolean_OR*. If the broadcast value is TRUE, the next iteration begins; a FALSE value terminates the rollback algorithm.

A formal description of the algorithm consisting of procedure *rollback_1* and function *Boolean_OR* appears in Fig. 2.

```
function Boolean_OR;
{executed by processor Pi}
begin
  if Pi is a leaf then send UPDTDi to the
   parent
  else
    wait for the message containing a Boo-
     lean value (true/false) from each
     child;
    VALUE ← Boolean OR of UPDTDi and values
     received form each child;
    if Pi is the root then broadcast VALUE
     on the tree T;
    else send VALUE to the parent;
    endif;
  endif;
  VALUE ← value broadcast on T by the root;
  return (VALUE)
end;

{rollback algorithm executed by processor Pi}
algorithm rollback_1;
begin
  TRPi ← the latest event of Pi from stable
   and volatile logs;
  NOTDONE ← true;
  while NOTDONE loop
    send an update (SENTi→j (TRPi)) message
     to each neighbor Pj;
    UPDTDi ← false;
    repeat
      wait for an update message from a
       neighbor;
      {process each update message as fol-
       lows:}
      let m ← update(c) be the message re-
       ceived from Pj;
      compute RECDi←j(TRPi);
      if RECDi←j(TRPi) > c then
        find the latest event e of Pi such
         that RECDi←j(e) = c;
        TRPi ← e;
        UPDTDi ← true;
      endif;
    until (an update message from each
     neighbor is received);
    NOTDONE ← (Boolean_OR);
  endloop; {end of an iteration}
  RPi ← TRPi;
end;
```

Fig. 2. Algorithm *rollback_1*.

Correctness. Let I_k denote the kth iteration. Let R_k be the set of processors that update their tentative rollback points during I_k, $F_k \subseteq R_k$ be the set of processors that have made the final update to their tentative rollback points during the kth iteration[1] (and do not update in subsequent iterations of the rollback algorithm), and $N_k = V - R_k - \bigcup_{i=0}^{k-1} F_i$. Thus, each processor in N_k does not update its tentative rollback point in I_k and it has not found it correct rollback point till I_k. We denote the rollback point of P_i (event TRP_i) at the end of I_k by $\text{TRP}_i(k)$.

THEOREM 1. *$R_{k+1} = \phi$ iff the tentative rollback points at the end of I_k are consistent.*

PROOF. $\Rightarrow$. We prove this by taking contrapositive. Assume that the tentative rollback points of P_i and P_j are not consistent at the end of I_k. Thus, events $\text{TRP}_i(k)$ and $\text{TRP}_j(k)$ are not consistent. Without loss of generality, assume that $\text{RECD}_{i \leftarrow j}(\text{TRP}_i(k)) > \text{SENT}_{j \rightarrow i}(\text{TRP}_j(k)) = c$. Now consider I_{k+1}. Processor P_j sends an *update*(c') message to P_i where $c' \le c$. When P_i receives the *update*(c') message,

1) if $\text{RECD}_{i \leftarrow j}(\text{TRP}_i)) > c'$, then P_i updates TRP_i and,

2) if $\text{RECD}_{i \leftarrow j}(\text{TRP}_i)) \le c'$, then TRP_i is a predecessor of event $\text{TRP}_i(k)$.

In either case, P_i updates TRP_i during I_{k+1} and $P_i \in R_{k+1}$. Thus, $R_{k+1} \ne \phi$.

$\Leftarrow$. For all P_i and P_j in V, if the tentative rollback points $\text{TRP}_i(k)$ and $\text{TRP}_j(k)$ at the end of I_k are pairwise consistent, then $\text{RECD}_{i \leftarrow j}(\text{TRP}_i(k)) \le \text{SENT}_{j \rightarrow i}(\text{TRP}_j(k))$ and $\text{RECD}_{j \leftarrow i}(\text{TRP}_j(k)) \le \text{SENT}_{i \rightarrow j}(\text{TRP}_i(k))$. During the $k + 1$st iteration, P_i sends an *update* $(\text{SENT}_{i \rightarrow j}(\text{TRP}_i(k)))$ message to P_j, and P_j does not update TRP_j on receiving this message. Similarly, P_i does not update TRP_i during I_{k+1} on receiving the *update* message from P_j. This is true for all pair of processors, hence $R_{k+1} = \phi$. □

1. $F_0 \subseteq$ {faulty processors} is the set of processors that do not update their tentative rollback point in any iteration.

We say that P_i sends a **new** *update* message to P_j during an iteration if the value sent by P_i to P_j in the *update message* (value c if *update*(c) is sent) during that iteration is less than the value sent (by P_i to P_j in the *update* message) during the previous iteration. For P_i to send a **new** *update* message during I_k, P_i must have updated TRP_i during I_{k-1}.

From Theorem 1, algorithm *rollback_1* terminates if and only if the tentative rollback points of all of the processors are pairwise consistent. We next bound the number of iterations.

LEMMA 1. *For all* $k \geq 0$, *if* $R_k \neq \phi$ *then* $F_k \neq \phi$.[2]

PROOF. Assume that $R_k \neq \phi$ and assume for contradiction that $F_k = \phi$. Thus, each processor of R_k will receive a new *update* message and will find its final recovery point during a later iteration. These new *update* messages cannot be sent by any of the processors in F_i, $i < k$, since processors in F_i do not update their tentative rollback points after I_i. So, we have to consider only those processors not in F_i, $i < k$. In I_{k+1}, processors in N_k do not send new *update* messages since they do not update their tentative rollback points during I_k. All of the processors in R_k find their final rollback points at iterations later than k on receiving new *update* messages and these *update* messages are "triggered" by new *update* messages sent during I_{k+1} by the processors of R_k. Consider an arbitrary processor $P_i \in R_k$. Since P_i updates TRP_i (on receipt of a new *update* message) at an iteration later than k and since only processors of R_k send new *update* messages during I_{k+1}, event $\text{TRP}_i(k)$ depends on event $\text{TRP}_j(k)$ for some $P_j \in R_k$. This dependency can be represented by a dependency chain. Now consider only those dependency chains that begin with event $\text{TRP}_a(k)$ and end with event $\text{TRP}_b(k)$ for all $P_a, P_b \in R_k$. Among these dependency chains, let DC = $\{e_1, e_2, \ldots, e_l\}$ be the longest dependency chain (with the maximum number of events in it). Let $e_1 = \text{TRP}_c(k)$ for some $P_c \in R_k$ and let $e_l = \text{TRP}_d(k)$ for some $P_d \in R_k$. Since DC is the longest dependency chain, e_1 does not depend on $\text{TRP}_{c'}(k)$ for any $P_{c'} \in R_k$. Clearly, $P_c \in R_k$ will not receive a new *update* message after I_k and P_c will not update TRP_c after I_k. Thus, $P_c \in F_k$ contradicting the assumption that $F_k = \phi$. □

THEOREM 2. *Algorithm* rollback_1 *terminates after at most n iterations and finds the maximum consistent recovery point of each processor*.

PROOF. During I_k, if $R_k \neq \phi$, $F_k \neq \phi$. Thus, during I_k, at least one processor P_i finds its RP_i value (and does not update its TRP value at later iterations). The rollback algorithm terminates after $k + 1$ iterations if $R_k = \phi$. Thus, algorithm *rollback_1* terminates after at most n iterations.

Clearly, the final rollback point of each processor is maximum—P_i sets TRP_i to e only because the event that occurs immediately after event e of P_i is started by an orphan message. From Theorem 1, the rollback points are consistent when the algorithm terminates. □

2. R_0 = set of faulty processors.

Message Complexity. During each iteration, every processor sends an *update* message to each neighbor and hence $2m$ *update* messages (m is the number of channels) are sent. Boolean OR is computed once during each iteration using $O(n)$ messages. Thus, $O(m)$ messages are sufficient for each iteration. Let d be the number of iterations needed. After d iterations, no processor updates its tentative recovery point and the rollback algorithm terminates at the end of I_{d+1}. By Theorem 2, $d \leq n - 1$. The message complexity of algorithm *rollback_1* is $O(md)$ with a one-time preprocessing step (constructing a spanning tree) that uses $O(m + n\log n)$ messages.

Time Complexity. During each iteration, a processor sends one *update* message to each neighbor, receives one *update* message from each neighbor, and all of the processors compute the Boolean OR of local values. Since message processing time is negligible and messages take unit time to traverse a channel (assumed for time complexity only), exchanging *update* messages uses $O(1)$ time. Before the end of each iteration, the value $\text{UPDTD}_1 \vee \ldots \vee \text{UPDTD}_n$ is computed distributively using the tree T. If T is a breadth-first tree, then Boolean OR can be computed using $O(D)$ time where D is the diameter of the network. Since the total number of iterations is d, the time complexity is $O(Dd)$. The time complexity of the one time preprocessing step $O(n\log n)$.

3.3.3 Rollback Algorithm 2

Algorithm *rollback_1* checks for termination (by computing Boolean OR) at the end of each iteration. If the value of d is known to all of the processors, then the termination condition need not be evaluated and the time complexity can be improved. After d iterations, each processor can terminate the rollback algorithm without invoking function Boolean OR. Since the value of d is not always known a priori, we guess the value in *stages*.

During stage t, each processor executes 2^t iterations. At the end of stage t (but not at the end of each iteration), we check the termination condition by performing a Boolean OR of the local decisions as in Algorithm *rollback_1* (using function *Boolean_OR* of Fig. 2). If a processor updates its tentative recovery point during the last iteration of stage t, then more iterations are needed and we proceed to stage $t + 1$. If no processor updates its tentative recovery point during the last iteration of stage t, the processors terminate.

Complexity. In algorithm *rollback_2*, we evaluate the termination condition (computing Boolean OR) at the end of each stage and stage t consists of 2^t iterations. If *maxstages* is the maximum number of stages needed, then *maxstages* = $O(\log d)$ (recall that d is the number of iterations needed) and the total number of iterations = $2^0 + 2^1 + \ldots + 2^{maxstages} = O(d)$. Thus, the message complexity is $O(md)$. Each iteration can be completed in one time unit and Boolean OR is computed once for each stage (at the end of each stage). Thus, the time complexity is $O(d + D\log d)$ where D, the diameter of the network, is the time needed for one invocation of function *Boolean_OR* assuming that a breadth-first tree is used.

3.3.4 Rollback Algorithm 3

The time complexity of algorithm *rollback_2* is $O(n\log n)$ if d and D are $O(n)$. The worst-case time complexity can be reduced by assuming that $d = n - 1$ and not evaluating the termination condition (Boolean OR). Algorithm *rollback_3* consists of $n - 1$ iterations. The time and message complexities of algorithm *rollback_3* are $O(n)$ and $O(mn)$, respectively.

THEOREM 3. *$\Theta(n)$ time units are necessary and sufficient for rolling back processors in general networks.*

PROOF. Each P_i must be informed about the processor failures before P_i can begin the crash recovery algorithm. The diameter of the underlying network may be $O(n)$ in the worst case, and hence $\Omega(n)$ time units are needed. $O(n)$ is the time complexity of algorithm *rollback_3*. □

The results are summarized in Table 1. Algorithms *rollback_1* and *rollback_2* have the same message complexity and algorithm *rollback_2* has a better time complexity. For complete graphs, $D = 1$ and the first two algorithms are very efficient in both the message and the time complexity. If D and d are large (say $O(n)$), then algorithm *rollback_3* is preferable because of its time complexity (and all the three algorithms have the same message complexity in this case). Also, algorithm *rollback_3* is uniform—all processors execute the same local algorithm.

TABLE 1
MESSAGE AND TIME COMPLEXITIES

Algorithm	Message Complexity	Time Complexity
rollback_1	$O(md)$	$O(Dd)$
rollback_2	$O(md)$	$O(d + D\log d)$
rollback_3	$O(mn)$	$\Theta(n)$

3.4 Experimental Investigation

In this section, we evaluate the performance of the three rollback algorithms by extensive simulation of several distributed systems. The application program simulated is a simple distributed on-line transaction processing system that accesses local and remote objects. Accessing remote objects is done by message-passing. When a processor receives a message, it processes the message and sends messages to some randomly selected neighbors, similar to the simulation in [9]. A global clock is used to schedule the events in each processor. (The global clock is not used by the rollback algorithms; it is used only for the simulation.) When a message is sent to a processor, the random delay in transmitting the message is assumed to be exponentially distributed with a mean of 0.1 millisecond. Processors independently checkpoint their states (including volatile log) at predetermined intervals. The failure time is determined randomly, and multiple processors are assumed to have failed. Figs. 3, 4, 5, and 6 show our simulation results with a 90% confidence interval. The simulation was run 100 times by varying the seed to determine each point in the figures.

Fig. 3 shows the number of messages used by algorithm *rollback_1*. (Recall that the message complexity of algorithm *rollback_1* is $O(md)$.) The number of messages increases as the number of processors increases. However, for a small change in n the value of d may also play a role. In Fig. 3 (for density = 2), this is reflected by a decrease in the number of messages when n changes from 22 to 24, which is due to a decrease in the value of d. (The value of d depends also on the application program.) In general, the number of messages (asymptotically) increase with n, as m and d depends on n. Also, note that the number of messages increases with density[3] due to the increase in m. Fig. 4 compares the three rollback algorithms with respect to the number of messages required for rolling back. Algorithms 1 and 2 differ only by a constant in their message complexity, and this is well reflected in the figure. Algorithm 3 is expensive since d is not always equal to $n - 1$.

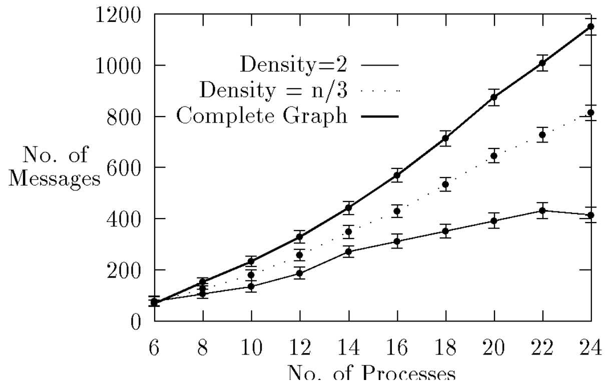


Fig. 3. Number of messages used by Algorithm *rollback_1*. Checkpoint interval = 50.

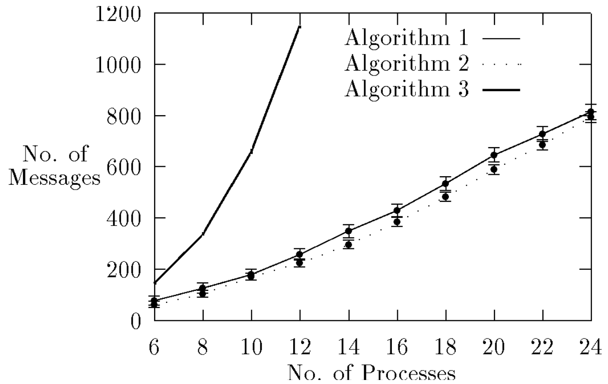


Fig. 4. Comparison of number of messages used by the three algorithms.

Fig. 5 shows the variation of rollback time of algorithm *rollback_1* when the number of processors is increased. The total time for crash recovery includes the time to find the rollback points and the time for restoring each processor to the state immediately after its rollback point. In our simulation, we measured only the time to find the rollback points. (A processor's state is restored from the stable storage at most once during recovery.) As both the diameter D

3. The density of a network is the ratio of the number of channels to the number of nodes.

and the length of dependency chain d may increase with n, the rollback time also increases with n. For complete graphs the diameter is always 1, and hence its rollback time is less than the rollback time in lower density graphs. Fig. 6 compares the performance of our three algorithms with respect to rollback time. Algorithm 3 performs badly when the length of the dependency chain is very less compared to $n - 1$. Algorithm 2 performs better than Algorithm 1 because *Boolean_OR* is not executed after every iteration, and *Boolean_OR* is a time consuming operation if the diameter of the network is high.

From our experimental study, we find that the average value of d is very small compared to the total number of processors. (However, we cannot generalize it as the value of d depends on the application program and checkpointing frequency.) Thus, on the average, the first two rollback algorithms are substantially more efficient than the third rollback algorithm (in terms of the number of messages and elapsed time) although the third rollback algorithm is asymptotically time-optimal in the worst case.

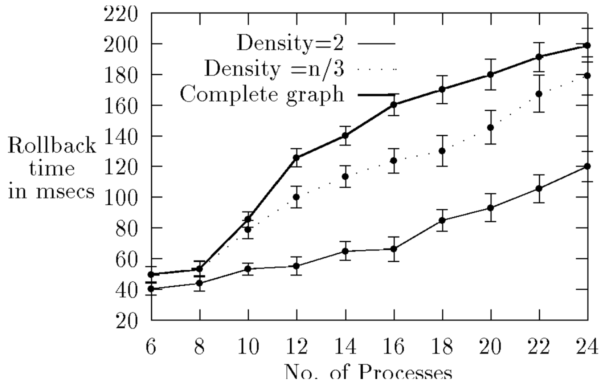


Fig. 5. Rollback time of Algorithm *rollback_1* for various densities.

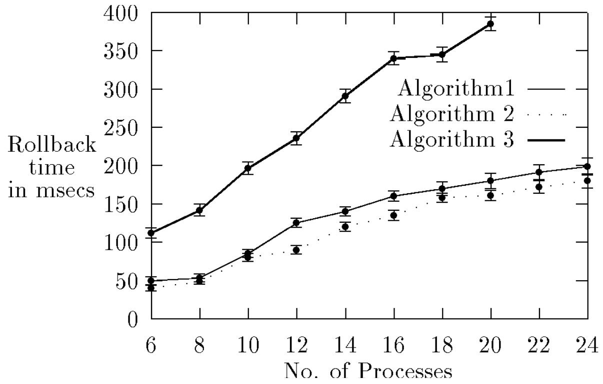


Fig. 6. Comparison of rollback time of three algorithms.

3.5 Restarting the Distributed Program

Using a rollback algorithm, processor P_i can find its recovery point RP_i. Clearly, RP_i is the latest event of P_i that is not an orphan event. Now, the state of P_i at the end of its event RP_i must be restored. If s_k^i is a checkpointed state of P_i and $e_l^i = \mathrm{RP}_i$, then P_i is restored to s_l^i by restarting from s_k^i and replaying all of the messages (available in volatile and stable storage) that started events $e_{k+1}^i, e_{k+2}^i, \ldots, e_l^i$. Messages generated during replay are duplicate messages and must not be sent to others unless they were "lost" by the faulty recipients. The neighbors of the faulty processors resend those "lost messages" to the faulty processors. A faulty processor P_i sends a *resend*$(\mathrm{RECD}_{i \leftarrow j}(\mathrm{RP}_i))$ message to neighbor P_j. On receipt of a *resend*(y) message from P_i, processor P_j checks if $\mathrm{SENT}_{j \rightarrow i}(\mathrm{RP}_j) > y$, and, if so, P_j resends the last $(\mathrm{SENT}_{j \rightarrow i}(\mathrm{RP}_j) - y)$ application messages that P_j sent to P_i during the previous run (before failures) by generating them if necessary. Regeneration of messages may be avoided if the senders log the outgoing messages.

The counters saved in the checkpoints of each processor (the total number of application messages sent to a neighbor and the total number of application messages received from a neighbor) increase monotonically and become unbounded if the application program uses a large number of application messages. These counters may be reset periodically by using the generalized scheme for bounding sequence numbers [14].

The checkpointed information will grow as each process takes a new checkpoint and messages are logged in stable storage. Also, an output message m cannot to committed unless we are sure that the process that generates m will not have to roll back beyond the event during which it generated m. Algorithms for these two problems are very closely related to the rollback algorithms as shown in [24].

4 Conclusions

The crash recovery algorithms presented in this paper use asynchronous checkpointing and place no communication overhead during normal operation of the system.

Strom and Yemini [23] introduce the concept of optimistic crash recovery in distributed systems and present a rollback algorithm. Their algorithm appends a vector of numbers to each application message and may use an exponential number of messages in the worst case [21]. Johnson and Zwaenepoel [7] unify several approaches to fault-tolerance based on message logging and checkpointing. They also show that there exists a unique maximum recoverable system state after failures and present a recovery algorithm. Every state interval of a process is assigned a unique interval index. Application messages sent during a state interval are tagged with the interval index of the state interval. A direct dependency vector is associated with each state interval and the vector is updated whenever a message is received. The recovery algorithm is executed when the state intervals become stable. In our scheme, the recovery algorithms are executed only when there is a failure. Since they find a recoverable state whenever state intervals become stable, recovery at the time of failures is simple. Our recovery algorithms are distributed whereas their algorithm is centralized.

Bhargava and Lian [2] propose an optimistic scheme for checkpointing and recovering for multiple failures. Processes take checkpoints independently. To recover from a failure, a process executes a two phase algorithm. In the first phase, the process collects information about the messages exchanged in the system. In the second phase, it builds a local system graph based on the information gathered and uses it to determine the set of processes that must rollback and the checkpoints to which they should rollback. The advantage of their scheme is checkpointing processes, rollback processes, and operational processes can proceed concurrently. A process does not make two consecutive rollbacks without performing any useful computation, but, in the worst case, a process may need to rollback to the beginning of execution in. Their scheme does not use message logging. We use message logging to reduce the amount rollback done, but normal operations are suspended during recovery.

Sistla and Welch [21] present two rollback algorithms that are more decentralized than the rollback algorithm of [7]. The first rollback algorithm uses $O(n^2)$ messages assuming that $O(n)$ numbers are appended to the application messages where n is the number of processors. The second rollback algorithm of [21] uses $O(n^3)$ messages by appending one number to the application messages. Our algorithms are an improvement over their second algorithm since we do not append any number to the application messages. Venkatesan and Juang [24] present an optimistic crash recovery algorithm that uses $O(n^2)$ messages when one number is appended to the application messages. Our work, in contrast, does not append any information to the application messages. Thus, there is no delay introduced in updating or appending numbers during normal operations. Also, since nothing is appended to the application messages, there is no communication overhead when there are no failures. However, a process might log unnecessary messages since it does not have a global dependency information. Our recovery algorithms, in the worst case, use more messages than the algorithms that append information [8], [21]. Our results, along with the results of Venkatesan and Juang [24], show that there is a trade–off between the amount of information appended to the application messages and the number of messages used during recovery.

Crash recovery can also be achieved using synchronous checkpointing [4], [22], [12]. In [12], processors coordinate their checkpointing actions such that the global state obtained by collecting the checkpointed states of all of the processors is consistent. When a processor fails, each processor rolls back and restarts from the latest checkpointed state. Each time a checkpoint is taken, this approach generates additional messages during normal operations even if there are no processor failures. Recovery techniques have been used for coping with faults that are not necessarily due to processor crashes (for example software faults). The reader is referred to [6], [10], [11], [19] for further reading.

The distributed system under consideration is an asynchronous system with FIFO communication channels. The FIFO property of the communication channels is used in the development of the algorithms. New approaches are needed for crash recovery if the channels are not FIFO. Assuming that the channels are F-channels [1], one may develop crash recovery techniques using message logging. The recovery algorithms presented in this paper can be applied to any application program. It is possible to make a particular protocol cope with and recover from failures by designing a crash recovery technique specific to that protocol and incorporating it into the protocol [20].

ACKNOWLEDGMENTS

We would like to thank Ivor Page and G. R. Dattatreya for helping us in several aspects of simulation. We are indebted to the anonymous referees for several insightful comments that have improved the presentation of the paper.

This research was supported in part by the U.S. National Science Foundation under Grant No. CCR-9110177 and by the Texas Advanced Technology Program under Grant No. 9741-036. A preliminary version of this paper appeared in the Proceedings of 11th International Conference on Distributed Computing Systems, May 1991.

REFERENCES

[1] M. Ahuja, “An Implementation of F Channels, a Preferable Alternative to FIFO Channels,” *Proc. 11th Int’l Conf. Distributed Computing Systems*, pp. 180–187, 1991.

[2] B. Bhargava and S. Lian, “Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems,” *Proc. Seventh Symp. Reliable Distributed Systems*, pp. 3–12, 1988.

[3] A. Borg, J. Baumbach, and S. Glazer, “A Message System Supporting Fault Tolerance,” *Proc. ACM Symp. Operating Systems Principles*, pp. 90–99, 1983.

[4] K. Chandy and L. Lamport, “Distributed Snapshots: Determining Global States of Distributed Systems,” *ACM Trans. Computing*, no. 3, pp. 3–75, 1985.

[5] R. Gallager, P. Humblet, and P. Spira, “A Distributed Algorithm for Minimum Weight Spanning Trees,” *ACM Trans. Programming Languages and Systems*, vol. 5, no. 1, pp. 66–77, 1983.

[6] K. Hwang, and W. Tsai, “Asynchronous Recovery Protocols for Distributed Systems,” *Proc. 12th Ann. Computer Software and Applications Conf.* (COMPSAC), pp. 512–520, 1988.

[7] D. Johnson and W. Zwaenepoel, “Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing,” *J. Algorithms*, vol. 11, no. 3, pp. 462–491, 1990.

[8] T.-Y. Juang and S. Venkatesan, “Efficient Algorithms for Crash Recovery in Distributed Systems,” *Proc. 10th Int’l Conf. Foundations of Software Technology and Theoretical Computer Science*, pp. 349–361, 1990.

[9] J. Kim and T. Park, “An Efficient Protocol for Checkpointing Recovery in Distributed Systems,” *IEEE Trans. Parallel and Distributed Systems*, 1993.

[10] K. Kim, “Programmer-Transparent Coordination of Recovering Concurrent Processes: Philisophy and Rules for Efficient Implementation.” *IEEE Trans. Software Eng.*, vol. 14, no. 6, pp. 810–821, June 1988.

[11] K. Kim and A. Kavianpour, “A Distributed Recovery Block Approach to Fault-Tolerant Execution of Application Tasks in Hypercubes,” *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 1, pp. 104–111, 1993.

[12] R. Koo and S. Toueg, “Checkpointing and Rollback-Recovery for Distributed Systems,” *IEEE Trans. Software Eng.*, vol. 13, no. 1, pp. 23–31, Jan. 1987.

[13] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” *Comm. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[14] W. Lloyd and P. Kearns, “Bounding Sequence Numbers in Distributed Systems: A General Approach,” *Proc. 10th Int’l Conf. Distributed Computing Systems*, pp. 312–319, 1990.

[15] M. Powell and D. Presotto, “Publishing: A Reliable Broadcast Communication Mechanism,” *Proc. Ninth ACM Symp. Operating System Principles*, pp. 100–109, 1983.

[16] B. Randell, “System Structure for Software Fault Tolerance,” *IEEE Trans. Software Eng.*, vol. 1, no. 2, pp. 220–232, 1975.

[17] D. Russell, “State Restoration in Systems of Communicating Processes,” *IEEE Trans. Software Eng.*, vol. 6, no. 2, pp. 183–194, 1980.

[18] F. Schneider, “Byzantine Generals in Action: Implementing Fail-stop Processors,” *ACM Trans. Computing*, vol. 2, no. 2, pp. 145–154, 1984.

[19] S. Shrivastava, *Reliable Computer Systems: Collected Papers of the Newcastle Reliability Project*. Springer Verlag, 1985.

[20] M. Singhal, "A Dynamic Information-Structure Mutual Exclusion Algorithm for Distributed Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 3, no. 1, pp. 121–125, 1992.

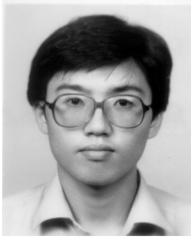
[21] A. Sistla and J. Welch, "Efficient Distributed Recovery Using Message Logging," *Proc. ACM Symp. Principles of Distributed Computing*, pp. 223–238, 1989.

[22] M. Spezialetti and P. Kearns, "Efficient Distributed Snapshots," *Proc. Sixth Int'l Conf. Distributed Computing Systems,* pp. 382–388, 1986.

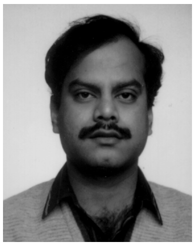
[23] R. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Trans. Computer Systems*, vol. 3, no. 3, pp. 204–226, 1985.

[24] S. Venkatesan and T.-Y. Juang, "Efficient Algorithms for Optimistic Crash Recovery," *Distributed Computing,* vol. 8, pp. 105–114, 1994.

S. Venkatesan received the BTech in civil engineering and the MTech in computer science from the Indian Institute of Technology, Madras, in 1981 and 1983, respectively, and the PhD in computer science from the University of Pittsburgh in 1988. He joined the University of Texas at Dallas in January 1989 where he is currently an associate professor of computer science. His research interests are fault-tolerant distributed systems, survivable telecommunications networks, and mobile computing.



Tony Tong-Ying Juang received his BS in naval architecture from National Taiwan University in 1983 and his MS and PhD degrees in computer science from the University of Texas at Dallas in 1989 and 1992, respectively. He is an associate professor of computer science at the Chung-Hua Polytechnic Institute. His research interests include distributed algorithms, fault-tolerant distributed computing, distributed operating systems, and computer communications.



Sridhar Alagar received the BE in computer science in 1990 from the Indian Institute of Science. From May 1990 to July 1991, he worked as a software engineer at PSI Data Systems Ltd., Bangalore. In August 1991, he joined the Graduate School, University of Texas at Dallas and obtained a PhD in computer science in 1995. Since September 1994, he has been a technical staff member at Alcatel Network System, where he is working in the areas of testing distributed systems, mobile computing, and survivable telecommunication networks.