Checkpointing and Rollback-Recovery for Distributed Systems

RICHARD KOO AND SAM TOUEG

Abstract—**We consider the problem of bringing a distributed system to a consistent state after transient failures. We address the two components of this problem by describing a distributed algorithm to create consistent checkpoints, as well as a rollback-recovery algorithm to recover the system to a consistent state. In contrast to previous algorithms, they tolerate failures that occur during their executions. Furthermore, when a process takes a checkpoint, a minimal number of additional processes are forced to take checkpoints. Similarly, when a process rolls back and restarts after a failure, a minimal number of additional processes are forced to roll back with it. Our algorithms require each process to store at most two checkpoints in stable storage. This storage requirement is shown to be minimal under general assumptions.**

Index Terms—**Checkpoint, consistent state, distributed systems, fault-tolerance, rollback-recovery.**

I. Introduction

CHECKPOINTING and rollback-recovery are well-known techniques that allow processes to make progress in spite of failures [11]. The failures under consideration are transient problems such as hardware errors and transaction aborts, i.e., those that are unlikely to recur when a process restarts. With this scheme, a process takes a checkpoint from time to time by saving its state on stable storage [8]. When a failure occurs, the process rolls back to its most recent checkpoint, assumes the state saved in that checkpoint, and resumes execution.

We first identify consistency problems that arise in applying this technique to a distributed system. We then propose a checkpoint algorithm and a rollback-recovery algorithm to restart the system from a consistent state when failures occur. Our algorithms prevent the well-known "domino effect" as well as livelock problems associated with rollback-recovery. In contrast to previous algorithms, they are fault-tolerant and involve a minimal number of processes. With our approach, each process stores at most two checkpoints in stable storage. This storage requirement is shown to be minimal under general assumptions.

Manuscript received January 31, 1986; revised June 16, 1986. R. Koo was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA Order 5378, Contract MDA903-85-C-0124, and by the National Science Foundation under Grants DCR-8412582 and MCS 83-03135. S. Toueg was supported by the National Science Foundation under Grants MCS 83-03135 and DCR-8601864.

The authors are with the Department of Computer Science, Cornell University, Ithaca, NY 14853.

IEEE Log Number 8611367.

The paper is organized as follows. We discuss the notion of consistency in a distributed system in Section II, and describe our system model in Section III. In Section IV we identify the problems to be solved. Sections V and VI contain the checkpoint and rollback-recovery algorithms, respectively. The algorithms are extended for concurrent executions in Section VII. In Section VIII we consider optimizations. Section IX contains our conclusion.

II. Consistent Global States in Distributed Systems

The notion of a consistent global state is central to reasoning about distributed systems. It was considered in [9], [10], [12], and formalized by Chandy and Lamport [2]. In this section, we summarize their ideas.

In a distributed computation, an *event* can be a spontaneous state transition by a process, or the sending or receipt of a message. Event *a directly happens before* event *b* [7] if and only if

1) *a* and *b* are events in the same process, and *a* occurs before *b*; or

2) *a* is the sending of a message *m* by a process and *b* is the receiving of *m* by another process.

The transitive closure of the *directly happens before* relation is the *happens before* relation. If event *a* happens before event *b*, *b happens after a*. (We abbreaviate *happens before*, "before" and *happens after*, "after.")

A *local state* of a process *p* is defined by *p*'s initial state and the sequence of events that occurred at *p*. A *global state* of a system is a set of local states, one from each process. The *state of the channels* corresponding to a global state *s* is the set of messages sent but not yet received in *s*. We can depict the occurrences of events over time with a time diagram, in which horizontal lines are time axes of processes, points are events, and arrows represent messages from the sending process to the receiving process. In this representation, a global state is a cut dividing the time diagram into two halves. The state of the channels comprises those arrows (messages) that cross the cut. Fig. 1 is a time diagram for a system of four processes.

Informally, a cut (global state) in the time diagram is *consistent* if no arrow starts on the right-hand side and ends on the left-hand side of it. This notion of consistency fits the observation that a message cannot be received be-

0098-5589/87/0100-0023$01.00 © 1987 IEEE

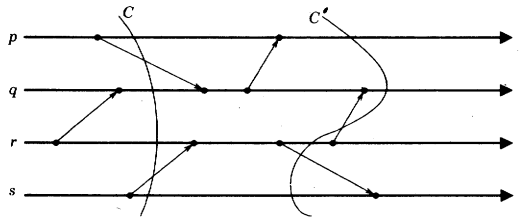


Fig. 1. Consistent and inconsistent cuts.

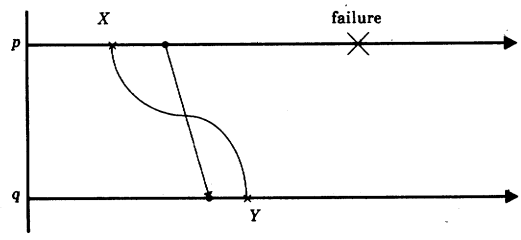


Fig. 2. Inconsistent checkpoints.

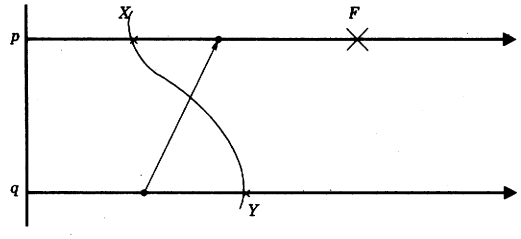


Fig. 3. Message loss due to rollback-recovery.

fore it is sent in any temporal frame of reference. For example, the cuts c and c' in Fig. 1 are consistent and inconsistent cuts, respectively. The state of the channels corresponding to cut c consists of one message from p to q, and another message from s to r. Readers are referred to [2] for a formal discussion of consistent global states.

III. System Model

The distributed system considered in this paper has the following characteristics:

1) Processes do not share memory and communicate via messages sent through channels.

2) Channels can lose messages. However, they are made (virtually) lossless and first-in-first-out by some end-to-end transmission protocol (such as a sliding window protocol [17]).

3) Processes can fail by stopping, and whenever a process fails, all other processes are informed of the failure in finite time. We assume that processes' failures never partition the communication network.

We want to develop our algorithms under a weak set of assumptions. In particular, we do not assume that the underlying system is a database transaction system [4], [6]. This special case admits simpler solutions: the mechanisms that ensure atomicity of transactions can hide inconsistencies introduced by the failure of a transaction. Furthermore, we do not assume that processes are deterministic: this simplifying assumption is made in previous results (e.g., [15] and [6]).

IV. Identification of Problems

A *checkpoint* is a saved local state of a process. A set of checkpoints, one per process in the system, is *consistent* if the saved states form a consistent global state. Restarting a system from a set of inconsistent checkpoints may cause problems as illustrated below.

Process p takes a checkpoint at time X and then sends a message to q (Fig. 2). After receiving this message, q takes a checkpoint at time Y. Subsequently, p fails and restarts from the checkpoint taken at X. The global state at p's restart is inconsistent because p's local state shows that no message has been sent to q, while q's local state shows that a message from p has been received. If p and q are processes supervising a customer's account at different banks, and the message transfers funds from p to q, the customer will have the funds at *both* banks when p restarts. This inconsistency persists even if q is forced to roll back and restart from its checkpoint taken at Y. Consistent checkpoints prevent such problems. Hence, our goal is to derive an algorithm for creating consistent set of checkpoints, and a rollback-recovery algorithm to restart the system from these consistent checkpoints.

Rollback-recovery from consistent checkpoints may cause message losses, as shown in Fig. 3. Process q sends a message m to process p, p receives m and fails at F, and then p and q roll back and recover from X and Y, respectively. At this point, q is in a state in which it has already sent m, and p is in a state in which m has not been received. Furthermore, the channel from q to p is empty. Hence, the system recovers from the consistent state $\{X, Y\}$ in which the message m is lost. This state can also be reached in an execution that had no rollback-recovery: q sends m and reaches Y, p reaches X, and the channel loses m. These two executions are indistinguishable to p and q. In both cases, the loss of m is masked by the end-to-end transmission protocol that we have assumed for the channels. Hence, standard end-to-end protocols can handle message losses that are due to channels, as well as losses that are due to site failures and rollback-recovery.

The problem of ensuring that the system recovers to a consistent state after transient failures has two components: checkpoint creation and rollback-recovery; we examine each one in turn.

A. Checkpoint Creation

There are two approaches to creating checkpoints. With the first approach, processes take checkpoints independently and save all checkpoints on stable storage. Upon a failure, processes must find a consistent set of checkpoints among the saved ones. The system is then rolled back to and restarted from this set of checkpoints [1], [5], [13], [18].

With the second approach, processes coordinate their checkpointing actions such that each process saves only its most recent checkpoint, and the set of checkpoints in the system is guaranteed to be consistent. When a failure occurs, the system restarts from these checkpoints [16].

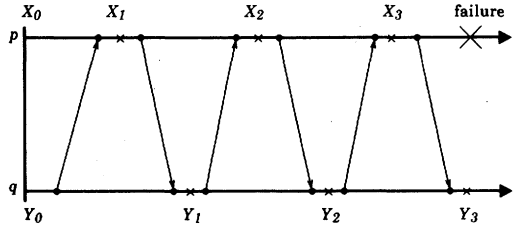


Fig. 4. "Domino effect" following a failure.

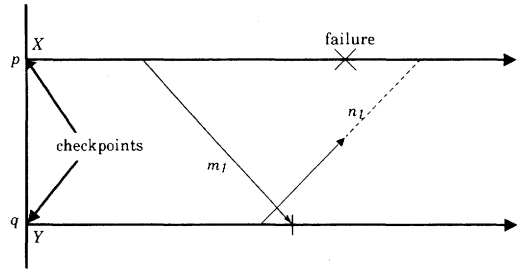


Fig. 5. Histories of p and q up to p's failure.

The main disadvantage of the first approach is the "domino effect" as illustrated in Fig. 4 [9], [10]. In this example, processes p and q have independently taken a sequence of checkpoints. The interleaving of messages and checkpoints leaves no consistent set of checkpoints for p and q, except the initial one at $\{X_0, Y_0\}$. Consequently, after p fails, both p and q must roll back to the beginning of the computation. For time-critical applications that require a guaranteed rate of progress, such as real time process control, this behavior results in unacceptable delays. An additional disadvantage of independent checkpoints is the large amount of stable storage required to save all checkpoints.

To avoid these disadvantages, we pursue the second approach. In contrast to [16], our method ensures that when a process takes a checkpoint, a minimal number of additional processes are forced to take checkpoints.

B. Rollback-Recovery

Rollback-recovery from a consistent set of checkpoints appears deceptively simple. The following scheme seems to work. Whenever a process rolls back to its checkpoint, it notifies all other processes to also roll back to their respective checkpoints. It then installs its checkpointed state and resumes execution. Unfortunately, this simple recovery method has a major flaw. In the absence of synchronization, processes cannot all recover (from their respective checkpoints) simultaneously. Recovering processes asynchronously can introduce *livelocks*; i.e., situations in which a single failure can cause an infinite number of rollbacks, preventing the system from making progress. Such a situation is illustrated below.

Fig. 5 illustrates the histories of two processes, p and q, up to p's failure. Process p fails before receiving the message n_1, rolls back to its checkpoint x, and notifies q. Then p recovers, sends m_2, and receives n_1. After p's recovery, p has no record of sending m_1, whereas q has a record of its receipt. Therefore, the global state is inconsistent. To restore consistency, q must also roll back to its checkpoint y (to "forget" the receipt of m_1). After q rolls back, it has no record of sending n_1 whereas p has a record of receiving n_1. Hence, p must roll back a second time to restore consistency (Fig. 6). Furthermore, q sends n_2 and receives m_2, after it recovers. Messages n_2 is received by p after it rolls back. However, as a result of this second rollback, p "forgets" the sending of m_2. Therefore, q must roll back a second time to restore consistency. And this second rollback of q will cause the third rollback of p because p receives the message n_2. It is now clear that p and q can be forced to roll back forever, even though no additional failures occur.

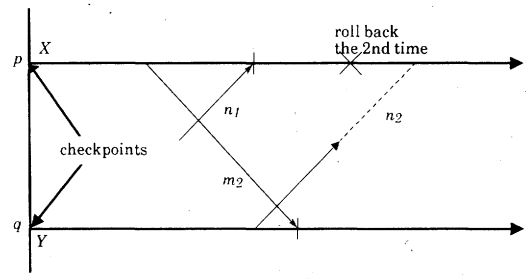


Fig. 6. Histories of p and q up to p's second rollback.

Our rollback-recovery algorithm solves this livelock problem. It tolerates failures that occur during its execution, and forces a minimal number of processes to roll back after a failure, whereas in [16], a single failure forces the system to roll back as a whole and the system crashes (and does not recover) if a failure occurs while it is rolling back.

V. Checkpoint Creation

A. Naive Algorithms

From Fig. 2 it is obvious that if every process takes a checkpoint after every sending of a message, and these two actions are done atomically, the set of the most recent checkpoints is always consistent. But creating a checkpoint after every send is expensive. We may naively reduce the cost of the above method with a strategy such as "every process takes a checkpoint after every k sends, $k > 1$" or "every process takes a checkpoint on the hour." However, the former can be shown to suffer domino effects by a construction similar to the one in Fig. 4, whereas the latter is meaningless for a system that lacks perfectly synchronized clocks.

B. Classes of Checkpoints

Our algorithm saves two kinds of checkpoints on stable storage: permanent and tentative. A permanent checkpoint cannot be undone. It guarantees that the computation needed to reach the checkpointed state will not be repeated. A tentative checkpoint, however, can be undone or changed to be a permanent checkpoint. When the context is clear, we call permanent checkpoints simply "checkpoints."

Consider a system with a consistent set of permanent checkpoints. A checkpoint algorithm is *resilient* to fail-

ures if the set of permanent checkpoints is still consistent after the algorithm terminates, even if some processes fail during its execution. To exclude the impractical "naive" algorithm from our consideration, henceforth we consider only those systems where processes either cannot afford to take a checkpoint after every send, or cannot combine the sending of a message and the taking of a checkpoint into one atomic operation. The following theorem shows that checkpoint algorithms for these systems must store at least two checkpoints in stable storage to be resilient to failures.

Theorem 1: No resilient checkpoint algorithms that take only permanent checkpoints exist.

Proof: By contradiction. Suppose that such an algorithm A exists. Consider the following scenario: p and q are processes. Suppose that by time t, $t > 0$, p has received a message m_q from q, and q a message m_p from p. At t, process p invokes A to take a checkpoint. Suppose that A terminates by time t', and that p takes a permanent checkpoint C_{p,t_p} at time t_p, $t < t_p \leq t'$. Since A is resilient, the set of checkpoints at the termination of A must be consistent. Therefore, process q must also have taken a permanent checkpoint C_{q,t_q} at time t_q, $t < t_q \leq t'$. Let d be the minimum time required for the failure of a process to be detected. Depending on whether $t_p \leq t_q$ or $t_p > t_q$, we now construct another execution of A that shows A is not resilient to failure.

Case 1: $t_p \leq t_q$. Let q fail in the time interval (max (t, $t_q - d$), t_q). Process p discovers the failure after t_q, hence after t_p. (See Fig. 7.) Consequently, C_{p,t_p} is taken although C_{q,t_q} is not. Since C_{p,t_p} is a permanent checkpoint that cannot be undone, and q fails before making a permanent checkpoint, the sending of m_q is "forgotten" forever whereas the receipt of m_q is "remembered" always, no matter what A does after p detects the failure. Hence, contrary to our assumption, Algorithm A is not resilient.

Case 2: $t_p > t_q$. Let p fail in the time interval (max (t, $t_p - d$), t_p). The rest of the proof is analogous to Case 1. □

Theorem 1 shows that in those systems we consider, any resilient checkpoint algorithm must store at least two checkpoints on stable storage.

C. Our Checkpoint Algorithm

We first assume that a *single* process invokes the algorithm to take a permanent checkpoint. In Section VII, we extend the algorithm for concurrent invocations. We also assume that no site fails during the execution of the algorithm. In Section V-C-4, we extend the algorithm to handle such failures. The algorithm sends its messages over (virtually) lossless and FIFO channels.

1) Motivation: The algorithm is patterned on two-phase-commit protocols. In the first phase, the initiator q takes a tentative checkpoint and requests all processes to take tentative checkpoints. If q learns that all processes have taken tentative checkpoints, q decides all tentative checkpoints should be made permanent; otherwise, q decides tentative checkpoints should be discarded. In the second phase, q's decision is propagated and carried out by all processes. Since all or none of the processes take permanent checkpoints, the most recent set of checkpoints is always consistent.

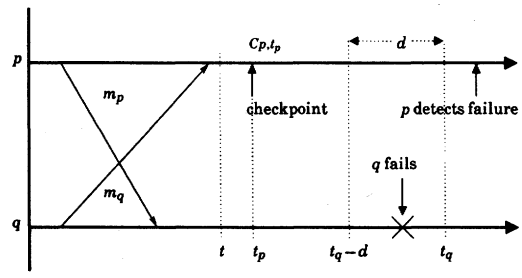


Fig. 7. The scenario when $t_p \leq t_q$ and q fails.

However, our goal is to force a minimal number of processes to take checkpoints. The above algorithm is modified as follows: a process p takes a tentative checkpoint after it receives a request from q *only if* q's tentative checkpoint records the receipt of a message from p, and p's latest permanent checkpoint does not record the sending of that message. (Note that the definition of consistency requires only that every message recorded as "received" in a checkpoint should also be recorded as "sent" in another checkpoint; and not vice versa.) Process p determines whether this condition is true using the label appended to q's request. This labeling scheme is described below.

Messages that are not sent by the checkpoint or rollback-recovery algorithms are *system* messages. Every system message m contains a field, which is a label denoted by $m.l$. Each process uses monotonically increasing labels in its outgoing systems messages. We define $\perp$ and $\top$ to be the smallest and largest labels, respectively. For any processes q and p, let m be the last message that q received from p after q took its last permanent or tentative checkpoint. Define:

$$last\_rmsg_q(p) = \begin{cases} m.l & \text{if } m \text{ exists} \\ \perp & \text{otherwise.} \end{cases}$$

Also, let m be the first message that q sent to process p after q took its last permanent or tentative checkpoint. Define:

$$first\_smsg_q(p) = \begin{cases} m.l & \text{if } m \text{ exists} \\ \perp & \text{otherwise.} \end{cases}$$

When q requests p to take a tentative checkpoint, it appends $last\_rmsg_q(p)$ to its request; p takes the checkpoint only if $last\_rmsg_q(p) \geq first\_smsg_p(q) > \perp$.

2) Informal Description: Process p is a *ckpt_cohort* of q if q has taken a tentative checkpoint, and $last\_rmsg_q(p) > \perp$ before the tentative checkpoint is taken. The set of ckpt_cohorts of q is denoted $ckpt\_cohort_q$. Every process p keeps a variable $willing\_to\_ckpt_p$ to denote its willingness to take checkpoints. Whenever p cannot take a checkpoint (for any reason), $willing\_to\_ckpt_p$ is "no." The initiator q starts the

Daemon process:

send(*initiator*, "take a tentative checkpoint and T");
await(*initiator*, $willing\_to\_ckpt_{initiator}$);
if $willing\_to\_ckpt_{initiator}$ = "yes" **then**
  send(*initiator*, "make tentative checkpoint permanent")
else
  send(*initiator*, "undo tentative checkpoint")
fi.

All processes p:

INITIAL STATE:
$first\_smsg_p(daemon) = \mathsf{T}$;
$willing\_to\_ckpt_p = \begin{cases} \text{"yes"} & \text{if } p \text{ is willing to take a checkpoint} \\ \text{"no"} & \text{otherwise} \end{cases}$

UPON RECEIPT OF "take a tentative checkpoint and $last\_rmsg_q(p)$" from q DO
  if $willing\_to\_ckpt_p$ and $last\_rmsg_q(p) \geq first\_smsg_p(q) > \perp$ **then**
    take a tentative checkpoint;
    for all $r \in ckpt\_cohort_p$ **send**(r, "take a tentative checkpoint and $last\_rmsg_p(r)$");
    for all $r \in ckpt\_cohort_p$ **await**(r, $willing\_to\_ckpt_r$);
    if $\exists\ r \in ckpt\_cohort_p$, $willing\_to\_ckpt_r$ = "no" **then** $willing\_to\_ckpt_p \leftarrow$ "no" **fi**;
  fi;
  send(q, $willing\_to\_ckpt_p$);
OD.

UPON FIRST RECEIPT OF m="make tentative checkpoint permanent" or
  m="undo tentative checkpoint" DO
  if m="make tentative checkpoint permanent" **then**
    make tentative checkpoint permanent;
  else
    undo tentative checkpoint;
  fi;
  for all $r \in ckpt\_cohort_p$, **send**(r, m);
OD.

Fig. 8. Algorithm C1: the Checkpoint Algorithm.

checkpoint algorithm by making a tentative checkpoint and sending a request "take a tentative checkpoint and $last\_rmsg_q(p)$" to all $p \in ckpt\_cohort_q$. A process p *inherits* this request if $willing\_to\_ckpt_p$ is "yes" and $last\_rmsg_q(p) \geq first\_smsg_p(q) > \perp$. If p inherits a request, it takes a tentative checkpoint and sends "takes a tentative checkpoint and $last\_rmsg_p(r)$" requests to all $r \in ckpt\_cohort_p$. If p receives but does not inherit a request from q, p replies $willing\_to\_ckpt_p$ to q.

After p sends out its requests, it waits for replies that can be either "yes" or "no," indicating a ckpt_cohort's acceptance or rejection of p's request. If any reply is "no," $willing\_to\_ckpt_p$ becomes "no"; otherwise $willing\_to\_ckpt_p$ is unchanged. Process p then sends $willing\_to\_ckpt_p$ to the process whose request p has inherited. From the time p takes a tentative checkpoint to the time it receives the decision from the initiator, p does not send any system messages.

If all the replies from its ckpt_cohorts arrive and are all "yes," the initiator decides to make all tentative checkpoints permanent. Otherwise the decision is to undo all tentative checkpoints. This decision is propagated in the same fashion as the request "take a tentative checkpoint" is delivered. A process discards its previous checkpoint after it takes a new permanent checkpoint.

The algorithm (C1) is presented in Fig. 8 (**await** does not prevent a process from receiving messages). For simplicity, we create a fictitious process called *daemon* to assume the initiation and decision tasks of the initiator. In practice, daemon is a part of the initiator process.

3) Proofs of Correctness: We consider a single invocation of the algorithm, and we assume no process fails.

Lemma 1: Every process inherits at most one request to take a tentative checkpoint.

Proof: Immediately after a process p inherits a request it takes a tentative checkpoint. From the time p takes this checkpoint to the time it receives the initiator's decision, p does not send any system messages. Therefore, during this interval of time $first\_smsg_p(q) = \perp$ for all q, and p cannot inherit additional requests. □

Lemma 2: Every process terminates its execution of Algorithm C1.

Proof: Any process that executes C1 without taking a tentative checkpoint clearly terminates. Let p be a process that takes a tentative checkpoint. By Lemma 1, p takes a tentative checkpoint exactly once. Consequently, to prove that C1 terminates at p, it suffices to prove that after p takes a tentative checkpoint, it does not wait forever for either the "yes" or "no" from its ckpt_cohorts, or the initiator's decision.

Let q be a ckpt_cohort of p. If q inherits p's request to take a tentative checkpoint, it sends $willing\_to\_ckpt_q$ to p before it waits for the initiator's decision. On the other hand, if q does not inherit p's request, it sends $willing\_to\_ckpt_q$ to p immediately after receiving p's request. Therefore, there can be no deadlock involving p waiting for q's reply and q waiting for the initiator's decision.

Process p cannot be in a deadlock waiting for replies from its ckpt_cohorts either. To show this, note that if q inherits a checkpoint request from p, p inherits a request before q does. The inherit relation cannot be circular, and hence no deadlock can arise. Therefore, p will receive replies from all its ckpt_cohorts.

After the initiator receives replies from all its ckpt_cohorts, it decides whether to make tentative checkpoints permanent or not. This decision is guaranteed to reach all processes that have taken tentative checkpoints since all processes forward the decision, and channels are reliable. Thus process p does not wait forever for replies from its ckpt_cohorts, or for the initiator's decision. □

The next lemma shows that C1 takes a consistent set of checkpoints.

Lemma 3: If the set of checkpoints in the system is consistent before the execution of Algorithm C1, the set of checkpoints in the system is consistent after the termination of C1.

Proof: Without loss of generality, assume new checkpoints are taken in C1. The proof is by contradiction. Suppose the set of checkpoints after C1 terminates is not consistent. Then there are two processes p and q, such that p sent q a message m after making its permanent checkpoint, and q received m before making its permanent checkpoint. Since all checkpoints are consistent before the execution of C1, q must have taken its permanent checkpoint during this execution. Before q took a tentative checkpoint in C1, $last\_rmsg_q(p) \geq m.l$; hence, p

was in $ckpt\_cohort_q$ and received a request to take a tentative checkpoint from q. When p received the request, $willing\_to\_ckpt_p$ had to be "yes" because q could not have made its tentative checkpoint permanent otherwise. Furthermore, either p had already taken a tentative checkpoint after sending m, or $last\_rmsg_q(p) \geq m.l \geq first\_smsg_p(q) > \perp$. In both cases, p took a tentative checkpoint after sending m. However, p makes its tentative checkpoint permanent if q makes its permanent. Consequently, p took a permanent checkpoint after sending m, a contradiction. □

We now show the number of processes that take new permanent checkpoints during the execution of Algorithm C1 is minimal. Let $P = \{p_0, p_1, \cdots, p_k\}$ be the set of processes that take new permanent checkpoints during the execution of C1, where p_0 is the initiator. Let $C(P) = \{c(p_0), c(p_1), \cdots, c(p_k)\}$ be the new permanent checkpoints taken by processes in P. Define an alternate set of checkpoints: $C'(P) = \{c'(p_0), c'(p_1), \cdots, c'(p_k)\}$ where $c'(p_0) = c(p_0)$ and for $1 \leq i \leq k$, $c'(p_i)$ is either $c(p_i)$ or the checkpoint p_i had before executing C1.

Theorem 2: $C'(P)$ is consistent if and only if $C'(P) = C(P)$.

Proof: The *if* part is by Lemma 3. We now prove the *only if* part. The execution of C1 imposes a "p inherits a request from q" relation on the set of processes. Since this relation is noncircular and there is only one initiator, it can be represented as a tree T: the root of T is the initiator, and p is a child of q if and only if p inherits a request from q. If $p \in T$, it must make a new permanent checkpoint during the execution of C1; hence $p \in P$. If $p \in P$, either p is the initiator or it inherits a request; hence $p \in T$. Therefore, $p \in T$ if and only if $p \in P$.

Our proof is by contradiction. Suppose that $C'(P) \neq C(P)$ and $C'(P)$ is consistent. Let $r \in P$ such that $c'(r) \neq c(r)$. Note that $r \neq p_0$, and that there exists a path from r to p_0 in T. Since $c'(p_0) = c(p_0)$, there is an edge (p, q) on this path such that $c'(p) \neq c(p)$, and $c'(q) = c(q)$. When p inherits q's request, $last\_rmsg_q(p) \geq first\_smsg_p(q) > \perp$. Let m be the message that q receives from p such that $last\_rmsg_q(p) = m.l$. Since $m.l \geq first\_smsg_p(q)$, the sending of m is not recorded in $c'(p)$. But the receipt of m is recorded in $c'(q)$. Thus, $C'(P)$ is not a consistent set of checkpoints, a contradiction. □

Theorem 2 shows that if p_0 takes a checkpoint, then all processes in P must take a checkpoint to ensure consistency.

4) Coping with Failures: We now extend Algorithm C1 to handle processes' failures. We first consider the effects of failures on nonfaulty processes. When failures occur, a nonfaulty process may not receive some of the following messages:

1) "yes" or "no" from ckpt_cohorts.

2) "make tentative checkpoint permanent" or "undo tentative checkpoint" from the initiator.

Suppose that process p fails before replying "yes" or "no" to process q's request. By the assumption of Section III, q will know of p's failure. After q knows that p has failed, it sets $willing\_to\_ckpt_q$ to "no" and stops waiting for p's reply. Therefore, to take care of a missing "yes" or "no," it suffices to change the statement in C1 from

if $\exists\ r \in ckpt\_cohort_p$, $willing\_to\_ckpt_r$ = "no" **then** $willing\_to\_ckpt_p \leftarrow$ "no" **fi**

to

if $\exists\ r \in ckpt\_cohort_p$, $willing\_to\_ckpt_r$ = "no" or r has failed **then**

$willing\_to\_ckpt_p \leftarrow$ "no" **fi**.

Suppose that process p does not receive the decision regarding its tentative checkpoint. If p undoes its tentative checkpoint or makes it permanent, it risks contradicting the initiator. The two-phase structure of C1 requires p to block until it discovers the initiator's decision [14]. We will discuss ways to prevent blocking in Section VIII.

We now consider the recovery of faulty processes. When a process restarts after a failure, its latest checkpoint on stable storage may be tentative or permanent. If this checkpoint is tentative, the restarting process must decide whether to discard it or to make it permanent. The decision is made as follows.

Suppose that the restarting process is the initiator. The initiator knows that every process that has taken a tentative checkpoint is still blocked waiting for its decision. Hence, it is safe for the initiator to decide to undo all tentative checkpoints and send this decision to its ckpt_cohorts. If the restarting process is not the initiator, it must discover the initiator's decision regarding tentative checkpoints. It may contact either the initiator or those processes of which it is a ckpt_cohort; it follows the decision accordingly to terminate C1.

The restarting process is now left with one permanent checkpoint on stable storage. It can recover from this checkpoint by invoking the rollback-recovery algorithm of Section VI.

Let C2 be the Algorithm C1 as modified above. C2 terminates if all processes that fail during the execution of C2 recover. At termination, the set of checkpoints in the system is consistent, and the number of processes that took new permanent checkpoints is minimal. The proofs for these properties are similar to those of C1 and they are omitted.

VI. Rollback-Recovery

We first assume that a *single* process invokes the algorithm to roll back and recover (henceforth denoted *restart*). We also assume that the checkpoint algorithm and the rollback-recovery algorithm are not invoked concur-

rently. In Section VII, we describe concurrent invocations of these algorithms. The algorithm sends its messages over (virtually) lossless and FIFO channels.

A. Motivation

The rollback-recovery algorithm is also patterned on two-phase-commit protocols. In the first phase, the initiator q requests all processes to restart from their checkpoints. Process q decides to restart all the processes if and only if they are all willing to restart. In the second phase, q's decision is propagated and carried out by all processes. We will prove that the two-phase structure of this algorithm prevents livelock as discussed in Section IV-B. Since all processes follow the initiator's decision, the global state is consistent when the rollback-recovery algorithm terminates.

However, our goal is to force a minimal number of processes to roll back. If a process p rolls back to a state saved before an event e occurred, we say that e is *undone* by p. The above algorithm is modified as follows: the rollback of a process q forces another process p to roll back *only if* q's rollback undoes the sending of a message to p. Process p determines if it must restart using the label appended to q's "prepare to roll back" request. This label is described below.

For any processes q and p, let m be the last message that q sent to p before q took its latest permanent checkpoint. Define

$$last\_smsg_q(p) = \begin{cases} m.l & \text{if } m \text{ exists} \\ \top & \text{otherwise.} \end{cases}$$

When q requests p to restart, it appends $last\_smsg_q(p)$ to its request. Process p restarts from its permanent checkpoint only if $last\_rmsg_p(q) > last\_smsg_q(p)$.

B. Informal Description

Process p is a *roll_cohort* of q if q can send messages to it. The set of roll_cohorts of q is $roll\_cohort_q$. Every process p keeps a variable $willing\_to\_roll_p$ to denote its willingness to roll back. Whenever p cannot roll back (for any reason), $willing\_to\_roll_p$ is "no." The initiator q starts the rollback-recovery algorithm by sending a request "prepare to roll back and $last\_smsg_q(p)$" to all $p \in roll\_cohort_q$. A process p *inherits* this request if $willing\_to\_roll_p$ is "yes," $last\_rmsg_p(q) > last\_smsg_q(p)$, and p has not already inherited another request to roll back. After p inherits the request, it sends "prepare to roll back and $last\_smsg_p(r)$" to all $r \in roll\_cohort_p$; otherwise, it replies $willing\_to\_roll_p$ to q.

After p sends out its requests, it waits for replies from each process in $roll\_cohort_p$. The reply can be an explicit "yes" or "no" message, or an implicit "no" when p discovers that r has failed. If any reply is "no," $willing\_to\_roll_p$ becomes "no"; otherwise $willing\_to\_roll_p$ is unchanged. Process p then sends $willing\_to\_roll_p$ to the process whose request p inherits. From the time p inherits the rollback request to the time it receives the decision from the initiator, p does not send any system messages.

If all the replies from its roll-cohorts arrive and are all "yes," the initiator decides the rollbacks will proceed; otherwise it decides no process will rollback. This decision is propagated to all processes in the same fashion as the request "prepare to roll back" is delivered. If failures prevent the decision from reaching a process p, p must block until it discovers the initiator's decision. We discuss nonblocking algorithms in Section VIII.

The rollback-recovery algorithm is presented in Fig. 9. Like the presentation of Algorithm C1, we introduce a fictitious process called *daemon* to perform functions that are unique to the initiator of the algorithm.

```
Daemon process:

send(initiator, "prepare to roll back and ⊥");
await(initiator, willing_to_roll_initiator);
if willing_to_roll_initiator = "yes" then
  send(initiator, "roll back")
else
  send(initiator, "do not roll back")
fi.

All processes p:

INITIAL STATE:
  ready_to_roll_p = true;
  last_rmsg_p(daemon) = ⊤;
  willing_to_roll_p = { "yes" if p is willing to roll back
                      { "no"  otherwise

UPON RECEIPT OF "prepare to roll back and last_smsg_q(p)" from q DO
  if willing_to_roll_p and last_rmsg_p(q) > last_smsg_q(p) and ready_to_roll_p then
    ready_to_roll_p ← false;
    for all r ∈ roll−cohort_p send(r, "prepare to roll back and last_smsg_p(r)");
    for all r ∈ roll−cohort_p await(r, willing_to_roll_r);
    if ∃ r ∈ roll_cohort_p, willing_to_roll_r = "no" or r has failed
        then willing_to_roll_p ← "no" fi;
  fi;
  send(q, willing_to_roll_p);
OD.

UPON RECEIPT OF m = "roll back" or
  m = "do not roll back" and ready_to_roll_p = false DO
  if m = "roll back" then
    restart from p's permanent checkpoint;
  else
    resume execution;
  fi;
  for all r ∈ roll−cohort_p, send(r, m);
OD.
```

Fig. 9. Algorithm R: the Rollback Algorithm.

C. Proofs of Correctness

We consider a single invocation of the rollback-recovery algorithm. The variable $ready\_to\_roll_p$ ensures that a process p inherits at most one request to roll back. As a result, the variable also ensures that a process rolls back at most once. To prove the termination of Algorithm R, it suffices to show that Algorithm R is free of deadlocks.

Lemma 4: Algorithm R is deadlock free.

Proof: Similar to the proof of Lemma 2. □

We now show that the global state of the system is consistent after the termination of R.

Lemma 5: If the system is consistent before the exe-

cution of Algorithm R, the system is consistent after the termination of Algorithm R.

Proof: The proof is by contradiction. Suppose that after Algorithm R terminates at every process, the global state of the system is inconsistent. There must be a message m sent by a process q to p such that during the execution of R, q undid the sending of m while p did not undo the receipt of m. We first show that p inherited a request to roll back. After q inherited a request to roll back, it stopped sending system messages. Hence, it must have sent a request to roll back to p *after* sending m. Moreover, since q undid the sending of m, $m.l > last\_smsg_q(p)$. On the other hand, process p could not have taken a permanent checkpoint after receiving m and before receiving q's request; the creation of this checkpoint and the fact that q did not take a permanent checkpoint would contradict Lemma 3. Consequently, $last\_rmsg_p(q) \geq m.l > last\_smsg_q(p)$. In addition, the variable $willing\_to\_roll_p$ must have been "yes," for the initiator cannot have decided to roll back. Therefore, when q's request reached p, either p had already inherited a rollback request or it inherited q's request.

Since p and q received the same decision, p rolled back. Next we show that p's rollback undid the receipt of m. There are two cases to consider:

Case 1: m reached p after p inherited a rollback request. Since message channels are FIFO, m reached p before q's request did. The initiator made its decision after p replied to q's request. Therefore, p rolled back after receiving m.

Case 2: m reached p before p inherited a rollback request. We have shown that p did not take a permanent checkpoint after receiving m. Hence, the rollback of p undid the receipt of m. □

Lemma 5 ensures that a single execution of Algorithm R brings the system to a consistent state after a failure; since processors roll back at most once in any execution of R, Algorithm R prevents livelocks.

Many existing rollback algorithms exhibit the following undesirable property. If the initiator rolls back, it forces an additional set of processes P to roll back with it, even though the system will be consistent if some of the processes in P omit to roll back. For example, the algorithm in [16] requires all processes to roll back every time any process wants to roll back. However, in some cases, the initiator could roll back alone and the system would still be consistent. With our algorithm, the number of processes that are forced to roll back with the initiator is minimal.

Theorem 3: Let E be an execution of R in which the initiator p_0 and an additional set of processes P roll back. Consider an execution E', identical to E except that a nonempty subset of processes in P omit to roll back upon receipt of the "roll back" decision. The execution E' leaves the system in an inconsistent state.

Proof: The execution of R imposes a "p inherits a 'prepare to roll back' request from q" relation on the set of processes. Since this relation is noncircular and there is only one initiator, it can be represented as a tree T: the root of T is the initiator, p_0, and p is a child of q if and only if p inherits a request from q. If $p \in T$, it rolls back during the execution of R; hence $p \in P \cup \{p_0\}$. If $p \in P \cup \{p_0\}$, either p is the initiator or it inherits a request; hence $p \in T$. Therefore, $p \in T$ if and only if $p \in P \cup \{p_0\}$.

Our proof is by contradiction. Suppose $P' \subseteq P$ is the set of processes that omit to roll back in the execution E', and the system is consistent at the end of E'. Let $r \in P'$. There exists a path from r to p_0 in T. Since r omits to roll back and p_0 rolls back, there is an edge (p, q) on this path, such that p omits to roll back and q rolls back. When p inherits the "prepare to roll back" request from q, $last\_rmsg_p(q) > last\_smsg_q(p)$. Let m be the message that q sent to p such that $m.l = last\_rmsg_p(q)$. When q rolls back it undoes the sending of m. But since p omits to roll back, it does not undo the receipt of m. Thus, at the end of E', the global state of the system is inconsistent, a contradiction. □

VII. Interference

In this section, we consider concurrent invocations of the checkpoint and rollback-recovery algorithms. An execution of these algorithms by process p is *interfered* with if any of the following events occur:

1) Process p receives a rollback request from another process q while executing the checkpoint algorithm.

2) Process p receives a checkpoint request from q while executing the rollback-recovery algorithm.

3) Process p, while executing the checkpoint algorithm for initiator i, receives a checkpoint request from q, but q's request originates from a different initiator than i.

4) Process p, while executing the rollback-recovery algorithm for initiator i, receives a rollback request from q, but q's request originates from a different initiator than i.

One single rule handles the four cases of interference: once p starts the execution of a checkpoint (rollback) algorithm, p is unwilling to take a tentative checkpoint (roll back) for another initiator, or to roll back (take a tentative checkpoint). As a result, in all four cases, p replies "no" to q. We can show this rule is sufficient to guarantee that all previous lemmas and theorems hold despite concurrent invocations of the algorithms. This rule can, however, be modified to permit more concurrency in the system. The modification is that in case 1), instead of sending "no" to q, p can begin executing the rollback-recovery algorithm after it finishes the checkpoint algorithm. We cannot allow a similar modification in case 2) lest deadlocks may occur.

VIII. Optimization

When the initiator of the checkpoint or of the rollback-recovery algorithm fails before propagating its decision to its cohorts, it is desirable for processes not to block for its recovery. To prevent processes from blocking, we can modify our algorithms by replacing the underlying two-phase commit protocol with a nonblocking three-phase

commit protocol [14]. However, nonblocking protocols are inherently more expensive than blocking ones [3].

We next address the following problem: after a ckpt__cohort q of a process p fails, p cannot take a permanent checkpoint until q restarts (p cannot know if the latest checkpoint of q records the sendings of all messages it received from q). To avoid waiting for q's restart, p can remove q from $ckpt\_\_cohort_p$ by restarting from its checkpoint (using the rollback-recovery algorithm). After its restart, process p can take new checkpoints.

XI. Conclusion

We have presented a checkpoint algorithm and a rollback-recovery algorithm to solve the problem of bringing a distributed system to a consistent state after transient failures. In contrast to previous algorithms, they tolerate failures that occur during their executions. Furthermore, when a process takes a checkpoint, a minimal number of additional processes are forces to take checkpoints. Similarly, a minimal number of additional processes are forces to restart when a process restarts after a failure. We also show that the stable storage requirement of our algorithms is minimal.

Acknowledgment

We would like to thank A. El Abbadi, K. Birman, R. Cleaveland, and J. Widom for commenting on earlier drafts of this paper.

References

[1] T. Anderson, P. A. Lee, and S. K. Shrivastava, "System fault tolerance," in *Computing System Reliability*, T. Anderson, and B. Randell, Eds. Cambridge, MA: Cambridge University Press, 1979, pp. 153–210.

[2] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, Feb. 1985.

[3] C. Dwork and D. Skeen, "The inherent cost of nonblocking commitment," in *Proc. ACM Symp. Principles of Database Syst.*, Mar. 1983.

[4] M. Fischer, N. Griffeth, and N. Lynch, "Global states of a distributed system," *IEEE Trans. Software Eng.*, vol. SE-85, pp. 198–202, May 1982.

[5] V. Hadzilacos, "An algorithm for minimizing rollback cost," in *Proc. ACM Symp. Principles of Database Syst.*, Mar. 1982.

[6] T. Joseph and K. Birman, "Low cost management of replicated data in fault-tolerant distributed systems," *ACM Trans. Comput. Syst.*, vol. 4, no. 1, pp. 54–70, Feb. 1986.

[7] L. Lamport, "Time, clocks and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[8] B. Lampson and H. Sturgis, "Crash recovery in a distributed storage system," Xerox Palo Alto Research Center, Tech. Rep., Apr. 1979.

[9] D. L. Presotto, "Publishing: A reliable broadcast communication mechanism," Comput. Sci. Division, Univ. California, Berkeley, Tech. Rep. UCB/CSD 83-165, Dec. 1983.

[10] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 226–232, June 1975.

[11] B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability issues in computing system design," *ACM Comput. Surveys*, vol. 10, no. 2, pp. 123–166, June 1978.

[12] D. L. Russel, "Process backup in producer-consumer systems," in *Proc. ACM Symp. Operat. Syst. Principles*, Nov. 1977.

[13] ——, "State restoration in systems of communicating processes," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 183–194, Mar. 1980.

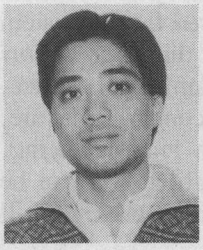
[14] D. M. Skeen, "Crash recovery in a distributed database system," Ph.D. dissertation, Comput. Sci. Division, University California, Berkeley, 1982.

[15] R. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Trans. Comput. Syst.*, pp. 204–226, Aug. 1985.

[16] Y. Tamir and C. H. Sequin, "Error recovery in multicomputers using global checkpoints," in *Proc. 13th Int. Conf. Parallel Processing*, Aug. 1984.

[17] A. S. Tanenbaum, *Computer Networks*. Englewood Cliffs, NJ: Prentice-Hall, 1981, pp. 148–164.

[18] W. G. Wood, "A decentralized recovery control protocol," in *Proc. 11th Ann. Int. Symp. Fault-Tolerant Comput.*, June 1981.



Richard Koo received the B.A. degree in mathematics from Colorado College, Colorado Springs, in 1982.

He is a Ph.D. student in the Department of Computer Science at Cornell University, Ithaca, NY. His current research interests include fault-tolerance, distributed algorithms, and distributed operating systems.

Sam Toueg received the B.Sc. degree in computer science from the Technion—Israel Institute of Technology, Haifa, in 1976, and the M.S.E., M.A., and Ph.D. degrees in computer science from Princeton University, Princeton, NJ, in 1977, 1978, and 1979, respectively.

He spent a postdoctoral year at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, in the Systems Analysis and Algorithms Division. In 1981 he joined the Department of Computer Science at Cornell University, Ithaca, NY, where he is currently an Assistant Professor. His current research interests include fault-tolerance, distributed computing, computer networks, and distributed database systems.

Dr. Toueg is a member of the Association for Computing Machinery, SIGACT, and SIGCOMM.