Checkpointing and Rollback-Recovery for Distributed Systems*

Richard Koo**
Sam Toueg†

Department of Computer Science
Cornell University
Ithaca, New York 14853

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.

1. Introduction

Checkpointing and rollback-recovery are well-known techniques that allow processes to make progress in spite of failures[12]. 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[9] 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.

The paper is organized as follows: We discuss the notion of consistency in a distributed system in section 2, and describe our system model in section 3. In section 4 we identify the problems to be solved. Sections 5 and 6 contain the checkpoint and rollback-recovery algorithms respectively. The algorithms are extended for concurrent executions in section 7. In section 8 we consider optimizations. Sections 9 and 10 contain a discussion and our conclusion.

2. Consistent Global States in Distributed Systems

The notion of a consistent global state is central to reasoning about distributed systems. It was considered by Randell[11], Russell[13], and Presotto[10], and formalized by Chandy and Lamport[2]. In this section, we summarise 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*[8] event b 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 views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defence position, policy, or decision.

**This author was supported by the Defence Advanced Research Projects Agency (DoD) under ARPA order 53[illegible]8 Contract MDA903-85-C-0124, and by the National Science Foundation under grants DCR-8412582 and MCS 83-03135.

†This author was supported by the National Science Foundation under grant MCS 83-03135.

CH2345-7/86/0000/1150$01.00 © 1986 IEEE

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 abbreviate *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. Figure 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 before it is sent in any temporal frame of reference. For example, the cuts c and c' in Figure 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 the work of Chandy and Lamport[2] for a formal discussion of consistent global states.

3. System Model

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

(1) Processes do not share memory or clocks. They communicate via messages sent through reliable first-in-first-out (FIFO) channels with variable nonzero transmission time.

(2) Processes 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.

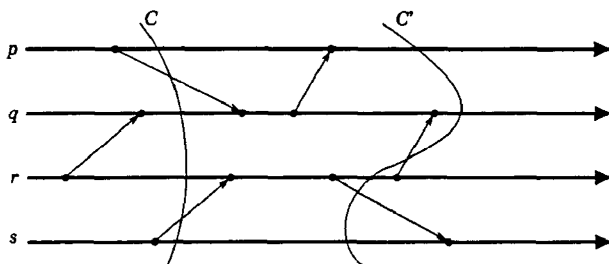

FIG. 1. Consistent and inconsistent cuts in a distributed system

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[6,16].

4. 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. For example, consider the system history in Figure 2. Process p takes a checkpoint at time X and then sends a message to q. 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.

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

4.1. 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[1,5,14,19].

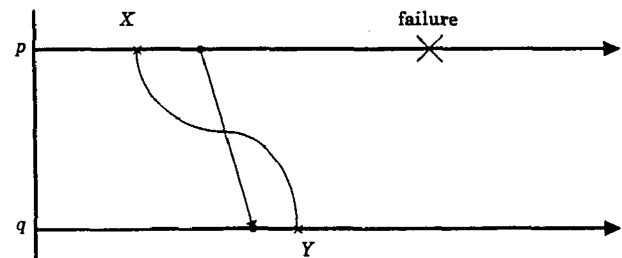

FIG 2. Inconsistent checkpoints.

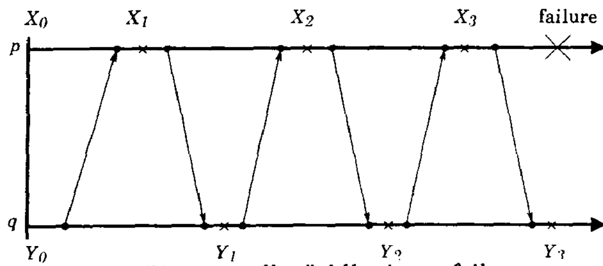

FIG. 3. "Domino effect" following a failure.

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[17].

The main disadvantage of the first approach is the "domino effect" as illustrated in Figure 3[10,11]. 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 Tamir[17], our method ensures that when a process takes a checkpoint, a minimal number of additional processes are forced to take checkpoints.

4.2. 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 as shown below.

Figure 4 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, 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, while q has a record of its receipt. Therefore the global state is

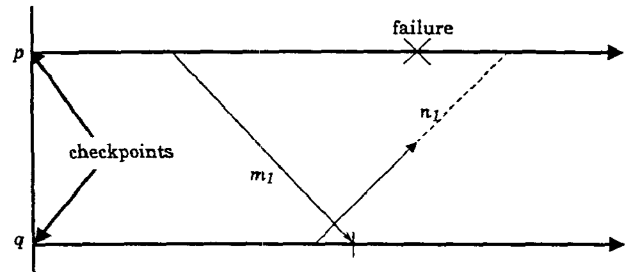

FIG. 4. Histories of p and q up to p's failure.

inconsistent. To restore consistency, q must also roll back (to "forget" the receipt of m_1) and notify p. However, after q rolls back, q has no record of sending n_1 while p has a record of its receipt. Hence, the global state is inconsistent again, and upon notification of q's rollback, p must roll back a second time. After q recovers, q sends n_2 and receives m_2. Suppose p rolls back before receiving n_2 as shown in Figure 5. With the second rollback of p, the sending of m_2 is "forgotten". To restore consistency, q must roll back a second time. After p recovers it receives n_2, and upon notification of q's rollback, it must roll back a third time. It is now clear that p and q can be forced to roll back forever, even though no additional failures occur.

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. However, in Tamir[17], a single failure forces the system to roll back as a whole. Furthermore, the system crashes (and does not recover) if a failure occurs while it is rolling back.

5. Checkpoint Creation

5.1. Naive Algorithms

From Figure 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

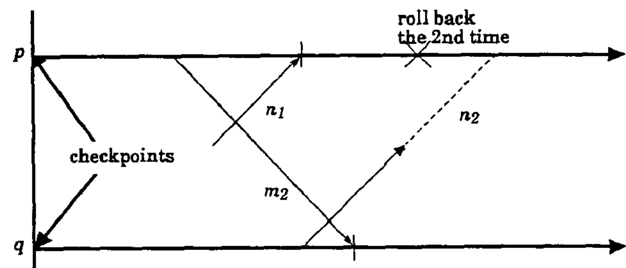

FIG. 5. History of p and q up to p's 2nd rollback.

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 Figure 3, while the latter is meaningless for a system that lacks perfectly synchronized clocks.

5.2. 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 "checkpoints".

Consider a system with a consistent set of permanent checkpoints. A checkpoint algorithm is *resilient* to failures 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 (in last section) 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. (The proofs of all lemmas and theorems in this paper can be found in Koo and Toueg[7].)

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

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

5.3. Our Checkpoint Algorithm

We assume the algorithm is invoked by a single process that wants to take a permanent checkpoint. We also assume that no failures occur in the system. In section 5.3.4 we extend the algorithm to handle failures, and in section 7 we describe concurrent invocations of this algorithm.

5.3.1. 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.

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. 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 label $m.l$. Each process appends outgoing system messages with monotonically increasing labels. 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 $last_rmsg_q(p) \geq first_smsg_p(q) > \perp$.

5.3.2. 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 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 "take 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 checkpoints after it takes a new permanent checkpoint.

The algorithm is presented in Figure 6. 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.

5.3.3. Proofs of Correctness

We consider a single invocation of the algorithm, and we assume no process fails in the system.

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

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

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. □

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 in C1, where p_0 is the initiator of C1. 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 $1 \le i \le k$, $c'(p_i)$ is either $c(p_i)$ or the checkpoint p_i had before executing C1.

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

<u>Daemon process:</u>

send(*initiator*, "take a tentative checkpoint and ⊤");
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.

<u>All processes p:</u>

INITIAL STATE:
  $first_smsg_p(daemon) = \top$;
  $willing_to_ckpt_p = \begin{cases} \text{"yes" if } p \text{ is willing to checkpoint} \\ \text{"no" otherwise} \end{cases}$;

UPON RECEIPT OF "take a tentative checkpoint and $last_rmsg_q(p)$" from q DO
  if $willing_to_ckpt_p$ and $last_rmsg_q(p) \ge first_smsg_p(q) > \bot$
    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 = "undo tentative checkpoint" or m = "make tentative checkpoint permanent" 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. 6. Algorithm C1: the Checkpoint Algorithm

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

*****await** does not prevent a process from receiving messages

5.3.4. 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 3, 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[15]. We will discuss ways to prevent blocking in section 8.

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 6.

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.

6. Rollback-Recovery

We assume that the algorithm is invoked by a single process that wants to roll back and recover (henceforth denoted *restart*). We also assume that the checkpoint algorithm and the rollback-recovery algorithm are not invoked concurrently. Concurrent invocations of these algorithms are described in section 7.

6.1. Motivation

The rollback-recovery algorithm is 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 4.2. 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 $last_rmsg_p(q) > last_smsg_q(p)$.

6.2. 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 roll back. 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 8.

The rollback-recovery algorithm is presented in Figure 7. 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 $\perp$");
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;
  $last_rmsg_p(daemon) = \top$;

$$willing_to_roll_p = \begin{cases} \text{"yes"} & \text{if } p \text{ is willing to roll back} \\ \text{"no"} & \text{otherwise} \end{cases} ;$$

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**
   $ready_to_roll_p \leftarrow$ false;
   for all $r \in roll-cohort_p$
    send(r, "prepare to roll back and $last_smsg_p(r)$");
   for all $r \in roll-cohort_p$ **await**(r, $willing_to_roll_r$);
   if $\exists\, r \in roll_cohort_p$, $willing_to_roll_r$ = "no"
    or r has failed **then** $willing_to_roll_p \leftarrow$ "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 $r \in roll-cohort_p$, **send**(r, m);
OD.

FIG. 7. Algorithm R: the Rollback Algorithm

6.3. 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. □

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

Lemma 5: If the system is consistent before the execution of Algorithm R, the system is consistent after the termination of Algorithm R. □

Lemma 5 ensures that a single execution of Algorithm R brings the system to a consistent state after a failure; since processes roll back at most once in any execution of R, the rollback algorithm prevents livelocks. Thus, 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 instance, all processes are required to roll back every time any process wants to roll back[17]. 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 non-empty 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. □

7. 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.

8. 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[15]. 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.

9. Message Loss

Rollback-recovery can cause message loss as illustrated in Figure 8. When p is rolled back to X following a failure at F, the global state is consistent, but the message m from q is lost. It is lost because the state of the channels corresponding to the global state $\{X, Y\}$ contains m.

One method to circumvent message loss is to have that processes use transmission protocols that transform lossy channels to virtual error-free channels, e.g., sliding window protocols[18]. Another method is to ensure that the state of the channels corresponding to the most recent set

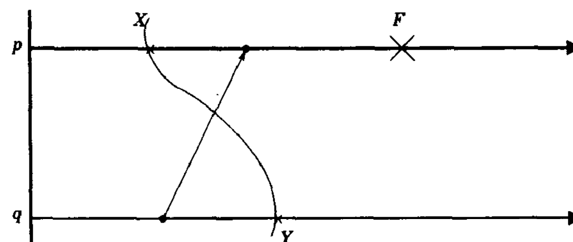

FIG. 8. Message loss following p's rollback to X.

of checkpoints contains no messages. We can modify the checkpoint and rollback-recovery algorithms to implement this latter method, but such modification increases the number of processes that are forced to take checkpoints and roll back.

10. 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 forced to take checkpoints. Similarly, a minimal number of additional processes are forced to restart when a process restarts after a failure. We also show that the stable storage requirement of our algorithms is minimal.

Acknowledgements

We would like to thank Amr El Abbadi, Ken Birman, Rance Cleaveland, and Jennifer Widom for commenting on earlier drafts of this paper.

Bibliography

[1] T. Anderson, P. A. Lee and S. K. Shrivastava, System fault tolerance, in *Computing System Reliability*, T. Anderson, B. Randell (eds.) Cambridge University Press, Cambridge, 1979, pp. 153-210.

[2] K. M. Chandy and L. Lamport, Distributed snapshots: Determining global states of distributed systems, *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63-75, February 1985.

[3] C. Dwork and D. Skeen, The inherent cost of nonblocking commitment, *Proc. ACM Symposium on Principles of Database Systems*, March 1983.

[4] M. Fischer, N. Griffeth, and N. Lynch, Global states of a distributed system, *IEEE Transaction on Software Engineering*, May 1982, pp. 198-202

[5] V. Hadzilacos, An algorithm for minimizing rollback cost, *Proc. ACM Symposium on Principles of Database Systems*, March 1982.

[6] T. Joseph and K. Birman, Low cost management of replicated data in fault-tolerant distributed systems, *ACM Transactions on Computer Systems*, February 1986, pp. 54-70.

[7] R. Koo and S. Toueg, Checkpointing and Rollback-Recovery for Distributed Systems, To appear in a special issue of *IEEE-TSE*.

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

[9] B. Lampson and H. Sturgis, Crash recovery in a distributed storage system, *Xerox PARC Tech. Rep.*, Xerox Palo Alto Research Center, April 1979.

[10] D. L. Presotto, Publishing: A reliable broadcast communication mechanism, *Tech. Rep. UCB/CSD 83-165, Computer Science Division, University of California, Berkeley*, December 1983.

[11] B. Randell, System structure for software fault tolerance, *IEEE Transactions On Software Engineering*, vol. SE-1, no.2, June 1975, pp. 226-232.

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

[13] D. L. Russell, Process backup in producer-consumer systems, *Proc. ACM Symposium on Operating Systems Principles*, November, 1977.

[14] D. L. Russell, State restoration in systems of communicating processes, *IEEE Transactions on Software Engineering*, vol. SE-6, no. 2, March 1980, pp. 183-194.

[15] D. M. Skeen, Crash recovery in a distributed database system, *Ph.D. dissertation, Computer Science Division, University of California, Berkeley*, 1982.

[16] R. Strom and S. Yemini, Optimistic recovery in distributed systems, *Transactions on Computer Systems*, August 1985, pp. 204-226.

[17] Y. Tamir and C. H. Sequin, Error recovery in multicomputers using global checkpoints, *Proc. of 13th International Conference on Parallel Processing*, August 1984.

[18] A. S. Tanenbaum, *Computer Networks*, Prentice Hall, New Jersey, 1981, pp. 148-164.

[19] W. G. Wood, A decentralized recovery control protocol, *Proc. of the 11th Annual International Symposium on Fault-Tolerant Computing*, June 1981.