On Coordinated Checkpointing in Distributed Systems

Guohong Cao, *Student Member*, *IEEE*, and Mukesh Singhal, *Member*, *IEEE*

Abstract—Coordinated checkpointing simplifies failure recovery and eliminates domino effects in case of failures by preserving a consistent global checkpoint on stable storage. However, the approach suffers from high overhead associated with the checkpointing process. Two approaches are used to reduce the overhead: First is to minimize the number of synchronization messages and the number of checkpoints, the other is to make the checkpointing process nonblocking. These two approaches were orthogonal in previous years until the Prakash-Singhal algorithm [18] combined them. In other words, the Prakash-Singhal algorithm forces only a minimum number of processes to take checkpoints and it does not block the underlying computation. However, we found two problems in this algorithm. In this paper, we identify these problems and prove a more general result: There does not exist a nonblocking algorithm that forces only a minimum number of processes to take their checkpoints. Based on this general result, we propose an efficient algorithm that neither forces all processes to take checkpoints nor blocks the underlying computation during checkpointing. Also, we point out future research directions in designing coordinated checkpointing algorithms for distributed computing systems.

Index Terms—Distributed system, coordinated checkpointing, causal dependence, nonblocking, consistent checkpoints.

1 Introduction

Coordinated checkpointing is an attractive approach for transparently adding fault tolerance to distributed applications without requiring additional programmer efforts. In this approach, the state of each process in the system is periodically saved on stable storage, which is called a checkpoint of the process. To recover from a failure, the system restarts its execution from a previous error-free, consistent global state recorded by the checkpoints of all processes. More specifically, the failed processes are restarted on any available machine and their address spaces are restored from their latest checkpoints on stable storage. Other processes may have to rollback to their checkpoints on stable storage in order to restore the entire system to a consistent state.

A system state is said to be consistent if it contains no *orphan message*, i.e., a message whose receive event is recorded in the state of the destination process, but its send event is lost [10], [22]. In order to record a consistent global checkpoint on stable storage, processes must synchronize their checkpointing activities. When a process takes a checkpoint, it asks (by sending checkpoint requests to) all relevant processes to take checkpoints. Therefore, coordinated checkpointing suffers from high overhead associated with the checkpointing process.

Much of the previous work [6], [9], [10], [13] in coordinated checkpointing has focused on minimizing the number of synchronization messages and the number of checkpoints during checkpointing. However, these algorithms (called *blocking algorithms*) force all relevant processes in the system to block their computations during checkpointing. Checkpointing includes the time to trace the dependence tree and to save the states of processes on stable storage, which may be long. Therefore, blocking algorithms may dramatically degrade system performance [3], [7].

Recently, nonblocking algorithms [7], [20] have received considerable attention. In these algorithms, processes need not block during checkpointing by using a checkpointing sequence number to identify orphan messages. However, these algorithms [7], [20] assume that a distinguished initiator decides when to take a checkpoint. Therefore, they suffer from the disadvantages of centralized algorithms, such as poor reliability, bottleneck, etc. Moreover, these algorithms [7], [20] require all processes in the system to take checkpoints during checkpointing, even though many of them may not be necessary. If they are modified to permit more processes to initiate checkpointing processes, which makes them distributed, these algorithms will suffer from another problem: In order to keep the checkpoint sequence number updated, any time a process takes a checkpoint, it has to notify all processes in the system. If each process can initiate a checkpointing process, the network would be flooded with control messages and processes might waste their time taking unnecessary checkpoints.

Prakash-Singhal algorithm [18] was the first algorithm to combine these two approaches. More specifically, it forces only a minimum number of processes to take checkpoints and does not block the underlying computation during checkpointing. However, we found two problems in this algorithm. In this paper, we identify these two problems and prove a more general result: There does not exist a nonblocking algorithm that forces only a minimum number of processes to take their checkpoints. Based on this general

• *The authors are with the Department of Computer and Information Science, Ohio State University, Columbus, OH 43210.*
E-mail: {gcao, singhal}@cis.ohio-state.edu.

Manuscript received 27 Jan. 1997.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 100387.

result, we propose an efficient nonblocking algorithm which significantly reduces the number of checkpoints. Also, we point out future research directions in designing coordinated checkpointing algorithms for distributed computing systems.

The rest of the paper is organized as follows. Section 2 presents preliminaries. In Section 3, we prove that there does not exist a nonblocking algorithm which forces only a minimum number of processes to take their checkpoints. Section 4 presents the nonblocking checkpointing algorithm. Correctness proofs are provided in Section 5. In Section 6, we compare the proposed algorithm with the existing algorithms. Section 7 presents future research directions and concludes the paper. Problems of the Prakash-Singhal algorithm are described in the Appendix.

2 Preliminaries

2.1 System Model

The distributed computation we consider consists of N spatially separated sequential processes denoted by $P_1, P_2, \cdots, P_N$. The processes do not share a common memory or a common clock. Message passing is the only way for processes to communicate with each other. The computation is asynchronous, i.e., each process progresses at its own speed and messages are exchanged through reliable channels, whose transmission delays are finite but arbitrary. The messages generated by the underlying distributed application are referred to as *computation messages*. The messages generated by processes to advance checkpoints are referred to as *system messages*.

Each checkpoint taken by a process is assigned a unique sequence number. The ith ($i \geq 0$) checkpoint of process P_p is assigned a sequence number i and is denoted by $C_{p,i}$. We also assume that each process P_p takes an initial checkpoint $C_{p,0}$ immediately before execution begins and ends with a *virtual* checkpoint that represents the last state attained before termination [16]. The ith *checkpoint interval* of process P_p denotes all the computation performed between its ith and $(i + 1)$th checkpoint, including the ith checkpoint but not the $(i + 1)$th checkpoint.

2.2 Minimizing Dependence Information

In order to record the dependence relationship among processes, we use a similar approach as the Prakash-Singhal algorithm [18], where each process P_i maintains a Boolean vector R_i, which has n bits. $R_i[j] = 1$ represents that P_i depends on P_j. At P_i, the vector is initialized to 0 except $P_i[i]$.

When process P_i sends a computation message m to P_j, it appends R_i to m. After receiving m, P_j includes the dependences indicated in R_i into its own R_j as follows: $R_j[k] = R_j[k] \vee m.R[k]$, where $1 \leq k \leq n$, and $\vee$ is the bitwise inclusive OR operator. Thus, if a sender P_i of a message depends on a process P_k before sending the computation message, the receiver P_j also depends on P_k through transitivity. In Fig. 1, P_2 depends on P_1 because of $m1$. When P_3 receives $m2$, P_3 (transitively) depends on $P1$.

The dependence information can be used to reduce the number of processes that must participate in the checkpointing process and the number of messages required to synchronize the checkpointing activity. In Fig. 1, the vertical line S_1 represents the global consistent checkpoints at the beginning of the computation. Later, when P_3 initiates a new checkpointing process, it only sends checkpoint requests to P_1 and P_2. As a result, only P_1, P_2, and P_3 take new checkpoints. P_4 and P_5 continue their computation without taking new checkpoints.

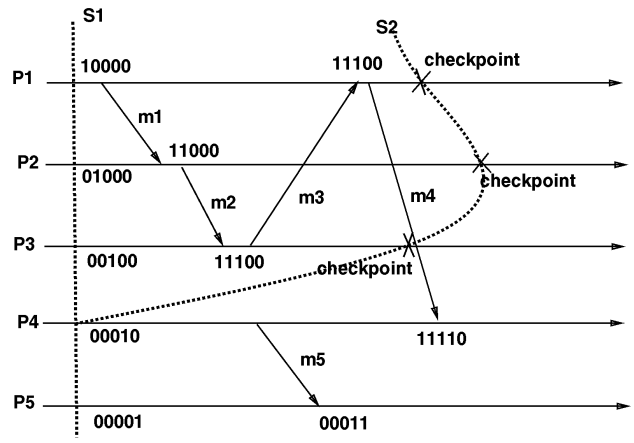

Fig. 1. Checkpointing and dependence information.

2.3 Basic Ideas Behind Nonblocking Algorithms

Most of the existing coordinated checkpointing algorithms [6], [10], [13] rely on the two-phase protocol and save two kinds of checkpoints on stable storage: *tentative* and *permanent*. In the first phase, the initiator takes a tentative checkpoint and forces all relevant processes to take tentative checkpoints. Each process informs the initiator whether it succeeded in taking a tentative checkpoint and blocks until released by the initiator in phase two. A process may refuse to take a checkpoint depending on its underlying computation. After the initiator has received positive acknowledgments from all relevant processes, the algorithm enters the second phase. If the initiator learns that all processes have successfully taken tentative checkpoints, it asks them to make their tentative checkpoints permanent; otherwise, it asks them to discard their tentative checkpoints. A process, on receiving the message from the initiator, acts accordingly.

A nonblocking checkpointing algorithm does not require any process to suspend its underlying computation. When processes do not suspend their computations, it is possible for a process to receive a computation message from another process which is already running in a new checkpoint interval. If this situation is not properly dealt with, it may result in an inconsistency. For example, in Fig. 2, P_2 initiates a checkpointing process. After sending checkpoint requests to P_1 and P_3, P_2 continues its computation. P_1 receives the checkpoint request and takes a new checkpoint, then it sends $m1$ to P_3. Suppose P_3 processes $m1$ before it receives the checkpoint request from P_2. When P_3 receives the checkpoint request from P_2, it takes a checkpoint (see Fig. 2). In this case, $m1$ becomes an orphan.

Most nonblocking algorithms [7], [20] use a Checkpoint Sequence Number (*csn*) to avoid inconsistencies. In these algorithms, a process is forced to take a checkpoint if it receives a computation message whose *csn* is greater than its

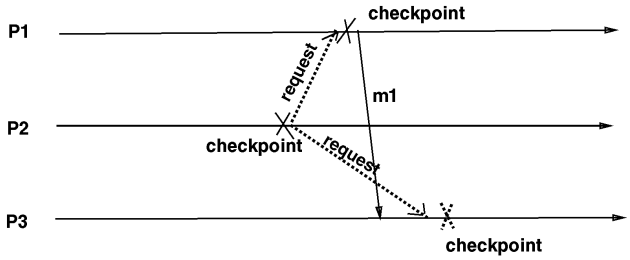

Fig. 2. Inconsistent checkpoints.

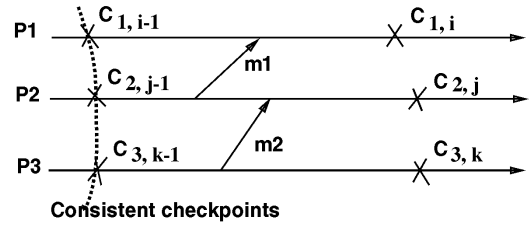

Fig. 3. The difference between causal dependence and z-dependence.

local *csn*. For example, in Fig. 2, P_1 increases its *csn* after it takes a checkpoint and appends the new *csn* to *m*1. When P_3 receives *m*1, it takes a checkpoint before processing *m*1 since the *csn* appended to *m*1 is larger than its local *csn*.

This scheme works only when every process in the computation can receive each checkpoint request and then increases its own *csn*. Since the Prakash-Singhal algorithm [18] forces only a subset of processes to take checkpoints, the *csn* of some processes may be out-of-date and may not be able to avoid inconsistencies. The Prakash-Singhal algorithm attempts to solve this problem by having each process maintain an array to save the *csn*, where $csn_i[i]$ is the expected *csn* of P_i. Note that P_i's $csn_i[i]$ may be different from P_j's $csn_j[i]$ if there has been no communication between P_i and P_j for several checkpoint intervals. By using *csn* and the initiator identification number, they claim that their nonblocking algorithm can avoid inconsistencies and minimize the number of checkpoints during checkpointing. However, we found two problems in their algorithm (see the Appendix). Next, we prove a more general result: "There does not exist a nonblocking algorithm that forces only a minimum number of processes to take their checkpoints."

3 Proof of Impossibility

Before presenting the proof, we define a new concept called "z-dependence," which is more general than causal dependence and can be used to model coordinated checkpointing.

Definition 1. *If a process P_p sends a message to a process P_q during its* i*th checkpoint interval and P_q receives the message during its* j*th checkpoint interval, then P_q* **z-depends** *on P_p during P_p's* i*th checkpoint interval and P_q's* j*th checkpoint interval, denoted as $P_p \prec_j^i P_q$.*

Definition 2. *If $P_p \prec_j^i P_q$ and $P_q \prec_k^j P_r$, then P_r* **transitively z-depends** *on P_p during P_r's* k*th checkpoint interval and P_p's* i*th checkpoint interval, denoted as $P_p \overset{*}{\prec}{}_k^i P_r$ (we simply call it "P_r transitively z-depends on P_p" if there is no confusion).*

Proposition 1.

$$P_p \prec_j^i P_q \Rightarrow P_p \overset{*}{\prec}{}_j^i P_q$$
$$P_p \overset{*}{\prec}{}_j^i P_q \wedge P_q \overset{*}{\prec}{}_k^j P_r \Rightarrow P_p \overset{*}{\prec}{}_k^i P_r.$$

The definition of "z-dependence" here is different from the concept of "causal dependence" used in the literature. We illustrate the difference between causal dependence and z-dependence in Fig. 3. Since P_2 sends *m*1 before it receives *m*2, there is no causal dependence between P_1 and P_3 due to these messages. However, these messages do establish a z-dependence between P_3 and P_1:

$$P_3 \prec_{j-1}^{k-1} P_2 \wedge P_2 \prec_{i-1}^{j-1} P_1 \Rightarrow P_3 \overset{*}{\prec}{}_{i-1}^{k-1} P_1.$$

Definition 3. *A* **min-process** *checkpointing algorithm is an algorithm satisfying the following condition: When a process P_p initiates a new checkpointing process and takes a checkpoint $C_{p,i}$, a process P_q takes a checkpoint $C_{q,j}$ associated with $C_{p,i}$ if and only if $P_q \overset{*}{\prec}{}_{i-1}^{j-1} P_p$.*

In coordinated checkpointing, to avoid an inconsistency, the initiator forces all dependent processes to take checkpoints. For any process, after it takes a checkpoint, it recursively forces all dependent processes to take checkpoints. The Koo-Toueg algorithm [10] uses this scheme, and it has been proven [10] that this algorithm forces only a minimum number of processes to take checkpoints. In the following, we prove that the Koo-Toueg algorithm is a min-process algorithm and a min-process algorithm forces only a minimum number of processes to take checkpoints. To simplify the proof, we use "$P_p \vdash_i^j P_q$" to represent the following: P_q causally depends on P_p when P_q is in the *i*th checkpoint interval and P_p is in the *j*th checkpoint interval.

Proposition 2.

$$P_p \vdash_i^j P_q \Rightarrow P_p \overset{*}{\prec}{}_i^j P_q$$
$$P_p \prec_i^j P_q \Rightarrow P_p \vdash_i^j P_q.$$

Lemma 1. *An algorithm forces only a minimum number of processes to take checkpoints if and only if it is a min-process algorithm.*

Proof. It has been proven [10] that the Koo-Toueg algorithm forces only a minimum number of processes to take checkpoints, thus, we only need to prove the following: In [10], when a process P_p initiates a new checkpointing process and takes a checkpoint $C_{p,i}$, a process P_q takes a checkpoint $C_{q,j}$ associated with $C_{p,i}$ if and only if $P_q \overset{*}{\prec}{}_{i-1}^{j-1} P_p$.

Necessity. In [10], when a process P_p initiates a new checkpoint $C_{p,i}$, it recursively asks all dependent processes

to take checkpoints. For example, P_p asks P_{k_m} to take a checkpoint, P_{k_m} asks $P_{k_{m-1}}$ to take a checkpoint, and so on. If a process P_q takes a checkpoint $C_{q,j}$ associated with $C_{p,i}$, there must be a sequence:

$$P_q \vdash^{j-1}_{s_{k_1}} P_{k_1} \wedge P_{k_1} \vdash^{s_{k_1}}_{s_{k_2}} P_{k_2} \wedge \cdots \wedge P_{k_{m-1}} \vdash^{s_{k_{m-1}}}_{s_{k_m}} P_{k_m} \wedge P_{k_m} \vdash^{s_{k_m}}_{i-1} P_p \quad (1 \le m \le N)$$

$$\Rightarrow P_q \prec^{*}_{s_{k_1}} {}^{j-1} P_{k_1} \wedge P_{k_1} \prec^{*}_{s_{k_2}} {}^{s_{k_1}} P_{k_2} \wedge \cdots \wedge P_{k_{m-1}} \prec^{*}_{s_{k_m}} {}^{s_{k_{m-1}}} P_{k_m} \wedge P_{k_m} \prec^{*}_{i-1} {}^{s_{k_m}} P_p$$

$$\Rightarrow P_q \prec^{*}_{i-1} {}^{j-1} P_p$$

OR

$$P_q \vdash^{j-1}_{i-1} P_p$$

$$\Rightarrow P_q \prec^{j-1}_{i-1} P_p$$

$$\Rightarrow P_q \prec^{*}_{i-1} {}^{j-1} P_p$$

SUFFICIENCY. If $P_q \prec^{j-1}_{i-1} P_p$ when P_p initiates a new checkpoint $C_{p,i}$, P_q takes a checkpoint $C_{q,j}$ associated with $C_{p,i}$; otherwise, if $P_q \prec^{*}_{i-1} {}^{j-1} P_p$, there must be a sequence:

$$P_q \prec^{j-1}_{s_{k_1}} P_{k_1} \wedge P_{k_1} \prec^{s_{k_1}}_{s_{k_2}} P_{k_2} \wedge \cdots \wedge P_{k_{m-1}} \prec^{s_{k_{m-1}}}_{s_{k_m}} P_{k_m} \wedge P_{k_m} \prec^{s_{k_m}}_{i-1} P_p \quad (1 \le m \le N)$$

$$\Rightarrow P_q \vdash^{j-1}_{s_{k_1}} P_{k_1} \wedge P_{k_1} \vdash^{s_{k_1}}_{s_{k_2}} P_{k_2} \wedge \cdots \wedge P_{k_{m-1}} \vdash^{s_{k_{m-1}}}_{s_{k_m}} P_{k_m} \wedge P_{k_m} \vdash^{s_{k_m}}_{i-1} P_p$$

Then, when P_p initiates a new checkpoint $C_{p,i}$, P_p asks P_{k_m} to take a checkpoint, P_{k_m} asks $P_{k_{m-1}}$ to take a checkpoint, and so on. In the end, P_{k_1} asks P_q to take a checkpoint. Then, P_q takes a checkpoint $C_{q,j}$ associated with $C_{p,i}$. □

DEFINITION 4. *A* **nonblocking min-process** *algorithm is a min-process checkpointing algorithm which does not block the underlying computation during checkpointing.*

LEMMA 2. *In a nonblocking min-process algorithm, assume P_p initiates a new checkpointing process and takes a checkpoint $C_{p,i}$. If a process P_r sends a message m to P_q after it takes a new checkpoint associated with $C_{p,i}$, then P_q takes a checkpoint $C_{q,j}$ before processing m if and only if $P_q \prec^{*}_{i-1} {}^{j-1} P_p$.*

PROOF. From the definition of min-process, P_q takes a checkpoint $C_{q,j}$ if and only if $P_q \prec^{*}_{i-1} {}^{j-1} P_p$. Thus, we only need to show that P_q takes $C_{q,j}$ before processing m. It is easy to see, if P_q takes $C_{q,j}$ after processing m, m becomes an orphan (as in Fig. 2). □

From Lemma 2, in a nonblocking min-process algorithm, when a process receives a message m, it must know if the initiator of a new checkpointing process transitively z-depends on it.

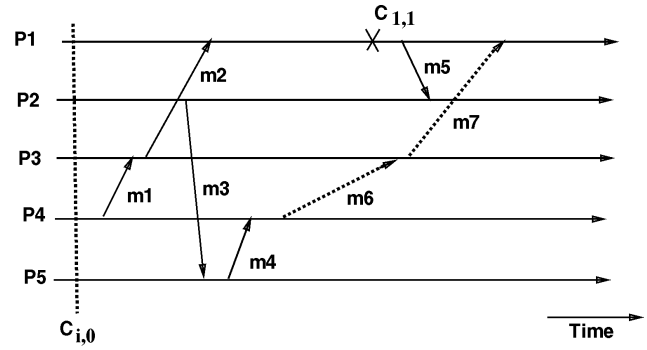

Fig. 4. Tracing the dependence.

LEMMA 3. *In a nonblocking min-process algorithm, there is not enough information at the receiver of a message to decide whether the initiator of a new checkpointing process transitively z-depends on the receiver.*

PROOF. The proof is by construction (using a counter-example). In Fig. 4, assume messages $m6$ and $m7$ do not exist. P_1 initiates a checkpointing process. When P_4 receives $m4$, there is a z-dependence as follows:

$$P_2 \prec^{*}_{0} {}^{0} P_4 \wedge P_4 \prec^{*}_{0} {}^{0} P_1 \Rightarrow P_2 \prec^{*}_{0} {}^{0} P_1.$$

However, P_2 does not know this when it receives $m5$. There are two possible approaches for P_2 to get the z-dependence information:

APPROACH 1 (Tracing the incoming messages). In this approach, P_2 gets the new z-dependence information from P_1. Then, P_1 has to know the z-dependence information before it sends $m5$ and appends the z-dependence information to $m5$. In Fig. 4, P_1 cannot get the new z-dependence information ($P_2 \prec^{*}_{0} {}^{0} P_1$) unless P_4 notifies P_1 of the new z-dependence information when P_4 receives $m4$. There are two ways for P_4 to notify P_1 of the new z-dependence information: First is to broadcast the z-dependence information (not illustrated in the figure); the other is to send the z-dependence information by an extra message $m6$ to P_3, which in turn notifies P_1 by $m7$. Both of them dramatically increase message overhead. Since the algorithm does not block the underlying computation, it is possible that P_1 receives $m7$ after it sends out $m5$ (as shown in the figure). Thus, P_2 still cannot get the z-dependence information when it receives $m5$.

APPROACH 2 (Tracing the outgoing messages). In this approach, since P_2 sends message $m3$ to P_5, P_2 hopes to get the new z-dependence information from P_5. Then, P_5 has to know the new z-dependence information and it would like to send an extra message (not shown in the figure) to notify P_2. Similarly, P_5 needs to get the new z-dependence information from P_4, which comes from P_3, and finally from P_1. Certainly, this requires much more

extra messages than Approach 1. Similar to Approach 1, P_2 still cannot get the z-dependence information in time since the computation is in progress. □

THEOREM 1. *No nonblocking min-process algorithm exists.*

PROOF. From Lemma 2, in a nonblocking min-process algorithm, a receiver has to know if the initiator of a new checkpointing process transitively z-depends on the receiver, which is impossible from Lemma 3. Therefore, no nonblocking min-process algorithm exists. □

COROLLARY 1. *There does not exist a nonblocking algorithm that forces only a minimum number of processes to take their checkpoints.*

PROOF. The proof directly follows from Lemma 1 and Theorem 1. □

3.1 Remarks

Netzer and Xu [16] introduced the concept of "zigzag" paths to define the necessary and sufficient conditions for a set of local checkpoints to lie on a consistent global checkpoint. Our definition of "z-dependence" captures the essence of zigzag paths. If an initiator forces all its transitively z-dependent processes to take checkpoints, the resulting checkpoints are consistent, and no zigzag path exists among them. If the resulting checkpoints are consistent, there is no zigzag path among them, and all processes on which the initiator transitively z-depends have taken checkpoints. However, there is a distinct difference between a zigzag path and z-dependence. A zigzag path is used to evaluate whether the existing checkpoints are consistent; thus, it is mainly used to find a consistent global checkpoint in an uncoordinated checkpointing algorithm. It has almost no use in a coordinated checkpointing algorithm, since a consistent global checkpoint is guaranteed by the synchronization messages. The z-dependence is proposed for coordinated checkpointing and it reflects the whole synchronization process of coordinated checkpointing, e.g., in the proof of Lemma 1, z-dependence is used to model the checkpointing process. Based on z-dependence, we found and proved the impossibility result. It is impossible to prove the result only based on zigzag paths.

Wang [23], [24] considered the problem of constructing the maximum and the minimum consistent global checkpoints that contain a target set of checkpoints. In his work, a graph called rollback dependence graph (or "R-graph") can be used to quickly find zigzag paths. Based on R-graph, Wang developed efficient algorithms to calculate the maximum and the minimum consistent global checkpoints for both general nondeterministic executions and piecewise deterministic executions. Manivannan et al. [14] generalized this problem. They proposed an algorithm to enumerate all such consistent global checkpoints that contain a target set of checkpoints and showed how the minimum and the maximum checkpoints are special cases. Their work [14] is based on a concept called "Z-cone," which is a generalization of zigzag paths. All these works [14], [23], [24] are based on zigzag paths. Similar to [16], these works [14], [23], [24] are not useful in coordinated checkpointing since checkpoints are guaranteed to be consistent in coordinated checkpointing. Thus, the notions of zig-zag paths and Z-cone help find consistent global checkpoints in uncoordinated checkpointing, while the notion of z-dependence helps understand the nature of coordinated checkpointing.

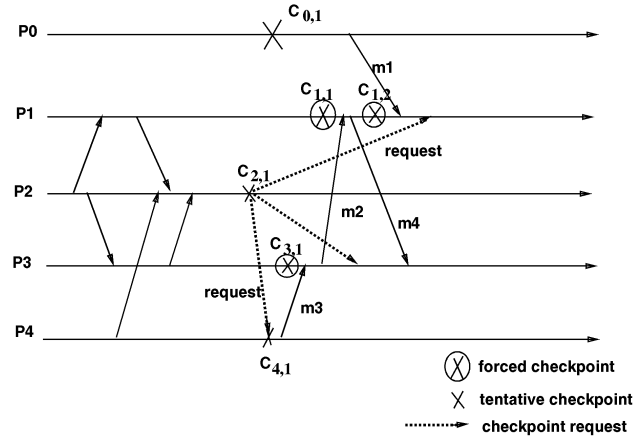

Fig. 5. An example of checkpointing.

4 A CHECKPOINTING ALGORITHM

In this section, we present our checkpointing algorithm, which neither blocks the underlying computation nor forces all processes to take checkpoints.

4.1 Basic Idea

4.1.1 Basic Scheme

A simple nonblocking scheme for checkpointing is as follows. When a process P_i sends a computation message, it piggybacks the current value of $csn_i[i]$ (*csn* is explained in Section 2.3). When a process P_j receives a computation message m from P_i, P_j processes the message if $m.csn \leq csn_j[i]$; otherwise, P_j takes a checkpoint (called *forced checkpoint*), updates its *csn* ($csn_j[i] = m.csn$), and then processes the message. This method may result in a large number of checkpoints. Moreover, it may lead to an *avalanche effect*, in which processes in the system recursively ask others to take checkpoints.

For example, in Fig. 5, to initiate a checkpointing process, P_2 takes its own checkpoint and sends checkpoint *request* messages to P_1, P_3, and P_4. When P_2's *request* reaches P_4, P_4 takes a checkpoint. Then, P_4 sends message $m3$ to P_3. When $m3$ arrives at P_3, P_3 takes a checkpoint before processing it due to $m3.csn > csn_3[4]$. For the same reason, P_1 takes a checkpoint before processing $m2$.

P_0 has not communicated with other processes before it takes a local checkpoint. Later, it sends a computation message $m1$ to P_1. P_1 takes checkpoint $C_{1,2}$ before processing $m1$, because P_0 has taken a checkpoint with checkpoint sequence number larger than P_1 expected. Then, P_1 requires P_3 to take another checkpoint (not shown in the figure) due to $m2$ and P_3 in turn asks P_4 to take another checkpoint (not shown in the figure) due to $m3$. If P_4 had received messages from other processes after it sent $m3$, those processes would have been forced to take checkpoints. This chain may never end.

We reduce the number of checkpoints based on the following observation. In Fig. 5, if $m4$ does not exist, it is not

necessary for P_1 to take $C_{1,2}$ since checkpoint $C_{1,1}$ is consistent with the rest of the checkpoints. Based on this observation, we get the following revised scheme.

4.1.2 Revised Basic Scheme

"When a process P_j receives a computation message m from P_i, P_j only takes a checkpoint when $m.csn > csn_j[i]$ and P_j has sent at least a message in the current checkpoint interval."

In Fig. 5, if $m4$ does not exist, $C_{1,2}$ is not necessary according to the revised scheme. However, if $m4$ exists, the revised scheme still results in a large number of checkpoints and may result in an avalanche effect.

4.1.3 Enhanced Scheme

We now present the basic idea of our enhanced scheme that eliminates avalanche effects during checkpointing. From Fig. 5, we make two observations:

OBSERVATION 1. It is not necessary to take checkpoint $C_{1,2}$ even though $m4$ exists, since P_0 does not transitively z-depend on P_1 during the zeroth checkpoint interval of P_0.

OBSERVATION 2. From Lemma 3, P_1 does not know if P_0 transitively z-depends on P_1 when P_1 receives $m1$.

These observations imply that $C_{1,2}$ is unnecessary but unavoidable. Thus, there are two kinds of forced checkpoints in response to computation messages. In Fig. 5, $C_{1,1}$ is different from $C_{1,2}$. $C_{1,1}$ is a checkpoint associated with the initiator P_2, and P_1 will receive the checkpoint *request* from P_2 in the future. $C_{1,2}$ is a checkpoint associated with the initiator P_0, but P_1 will not receive the checkpoint *request* from P_0 in the future. To avoid an inconsistency, P_1 should keep $C_{1,1}$ when P_1 receives P_2's *request*. However, P_1 can discard $C_{1,2}$ after a period (defined as the longest checkpoint *request* travel time). If only one checkpointing process is in progress, P_1 can also discard $C_{1,2}$ when it needs to make its tentative checkpoint permanent. Thus, our algorithm has three kinds of checkpoints: *tentative*, *permanent*, and *forced*. Tentative and permanent checkpoints are the same as before and they are saved on stable storage. Forced checkpoints do not need to be saved on stable storage. They can be saved anywhere, even in the main memory. When a process takes a tentative checkpoint, it forces all dependent processes to take checkpoints. However, a process taking a forced checkpoint does not require its dependent processes to take checkpoints. Suppose a process P_i takes a forced checkpoint associated with an initiator P_j. When P_i receives the corresponding checkpoint *request* associated with P_j, P_i converts the forced checkpoint into a tentative checkpoint. After that, P_i forces all processes on which it depends to take checkpoints. If P_i does not receive the checkpoint *request* associated with P_j after a period or P_i needs to make its tentative checkpoint permanent, it may discard the forced checkpoint associated with P_j to save storage. Actually, P_i may have not finished taking its checkpointing process after a period defined as before; if so, P_i stops the ongoing checkpointing and resumes its computation.

In Fig. 5, $C_{1,2}$ is a forced checkpoint that will be discarded in the future. Since $C_{1,2}$ is a forced checkpoint, it is not necessary for P_3 to take a new checkpoint when P_1 takes $C_{1,2}$ as in the previous scheme. Thus, our scheme avoids the avalanche effect and improves the performance of checkpointing.

4.2 Data Structures

The following data structures are used in our algorithm:

R_i: A Boolean vector described in Section 2.2. The operator $\ominus$ is defined as follows:

$$i \ominus j = \begin{cases} 1 & \text{if } i = 1 \wedge j = 0 \\ 0 & \text{otherwise} \end{cases}$$

csn_i: An array of n checkpoint sequence numbers (*csn*) at each process P_i. Each checkpoint sequence number is represented by an integer. $csn_i[j]$ represents the checkpoint sequence number of P_j that P_i knows. In other words, P_i expects to receive a message from P_j with checkpoint sequence number $csn_i[j]$. Note that $csn_i[i]$ is the checkpoint sequence number of P_i.

weight: A nonnegative variable of type real with maximum value of 1. It is used to detect the termination of the checkpointing algorithm as in [8].

$first_i$: A Boolean array of size n maintained by each process P_i. The array is initialized to all zeroes each time a checkpoint at that process is taken. When P_i sends a computation message to process P_j, P_i sets $first_i[j]$ to one.

$trigger_i$: A tuple (pid, inum) maintained by each process P_i. *pid* indicates the checkpointing initiator that triggered this process to take its latest checkpoint. *inum* indicates the *csn* at process *pid* when it took its own local checkpoint on initiating the checkpointing process. *trigger* is appended to each system message and the first computation message to which a process sends after taking its local checkpoint.

CP_i: A list of records maintained by each process P_i. Each record has the following fields:

- *forced*: forced checkpoint information of P_i.
- *R*: P_i's own Boolean vector before it takes the current forced checkpoint, but after it takes the previous forced checkpoint or permanent checkpoint, whichever is later.
- *trigger*: a set of *trigger*s. Each trigger represents a different initiator. P_i may have one common forced checkpoint for several initiators.
- *first*: P_i's own *first* before it takes the current forced checkpoint, but after it takes the previous forced checkpoint or permanent checkpoint, whichever is later.

The following notations are used to access these records:

$CP[i][k]$ is the kth record of P_i;

$CP[i][last]$ is the last record of P_i;

$CP[i][last + 1]$ will be a new record appended at the end of P_i's records.

csn is initialized to an array of zeros at all processes. The trigger tuple at process P_i is initialized to $(i, 0)$. The weight at a process is initialized to zero. When a process P_i sends any computation message, it appends its $csn_i[i]$ and R_i to

the message. Also, P_i appends its trigger in the first computation message it sends to every other process after taking its checkpoint. P_i checks if $send_i[j] = 0$ before sending a computation message to P_j. If so, it sets $send_i[j]$ to one and appends its trigger to the message.

4.3 The Checkpointing Algorithm

To clearly present the algorithm, we assume that at any time, at most one checkpointing process is in progress. Techniques to handle concurrent initiations of checkpointing by multiple processes can be found in [17]. As multiple concurrent initiations of checkpointing are orthogonal to our discussion, we only briefly mention the main features of [17]. When a process receives the first request for checkpointing initiated by another process, it takes a local checkpoint and propagates the request. All local checkpoints taken by the participating processes for a checkpoint initiation collectively form a global checkpoint. The state information collected by each independent checkpointing is combined. The combination is driven by the fact that the union of consistent global checkpoints is also a consistent global checkpoint. The checkpoints thus generated are more recent than each of the checkpoints collected independently and also more recent than that collected by [21]. Therefore, the amount of computation lost during rollback, after process failures, is minimized. Next, we present our nonblocking algorithm.

Checkpointing initiation. Any process can initiate a checkpointing process. When a process P_i initiates a checkpointing process, it takes a local checkpoint, increments its $csn_i[i]$, sets *weight* to one, and stores its own identifier and the new $csn_i[i]$ in its trigger. Then, it sends a checkpoint *request* to each process P_j such that $R_i[j] = 1$ and resumes its computation. Each *request* carries the trigger of the initiator, the R_i, and a portion of the weight of the initiator, whose weight is decreased by an equal amount.

Reception of checkpoint requests. When a process P_i receives a *request* from P_j, it compares P_j.*trigger* (*msg_trigger*) with P_i.*trigger* (*own_trigger*). If these two triggers are different, P_i checks if it has sent any message during the current checkpoint interval. If P_i has not sent a message during the current checkpoint interval, it updates its *csn*, sends a *reply* to the initiator with the weight equal to the weight received in the *request*, and resumes its underlying computation; otherwise, P_i takes a tentative checkpoint and propagates the *request* as follows. For each process P_k on which P_i depends but P_j does not (P_j has sent *request* to the processes on which it depends), P_i sends a *request* to P_k. Also, P_i appends the initiator's trigger and a portion of the received weight to all those *request*s. Then, P_i sends a *reply* to the initiator with the remaining weight and resumes its underlying computation.

If *msg_trigger* is equal to *own_trigger* when P_i receives a *request* (implying that P_i has already taken a checkpoint for this checkpointing process), P_i does not need to take a checkpoint. But,

- If there is a forced checkpoint whose trigger is equal to *msg_trigger*, P_i saves the forced checkpoint on stable storage (the forced checkpoint is turned into a tentative checkpoint). Then, P_i propagates the *request* as before.
- If there is not a forced checkpoint whose trigger is equal to *msg_trigger*, a *reply* is sent to the initiator with the weight equal to that received in the *request*.

Computation messages received during checkpointing. When P_i receives a computation message m from P_j, P_i compares $m.csn$ with its local $csn_i[j]$. If $m.csn \leq csn_i[j]$, the message is processed and no checkpoint is taken; otherwise, it implies that P_j has taken a checkpoint before sending m and m is the first computation message sent by P_j to P_i after P_j took its checkpoint. Therefore, the message m must have a trigger tuple. P_i first updates its $csn_i[j]$ to $m.csn$, then does the following depending on the information of P_j.*trigger* (*msg_trigger*) and P_i.*trigger* (*own_trigger*):

- If *msg_trigger* = *own_trigger*, it means that the latest checkpoints of P_i and P_j were both taken in response to the same checkpoint initiation. Therefore, no new local checkpoint is needed.
- If *msg_trigger* ≠ *own_trigger*, P_i checks if it has sent any message during the current checkpoint interval. If P_i has not sent any message during the current checkpoint interval, it only needs to add *msg_trigger* to its forced checkpoint list. If P_i has sent a message during the current checkpoint interval, it takes a forced checkpoint and updates its data structures such as *csn*, *CP*, *R*, and *first*.

Termination and garbage collection. The checkpoint initiator adds weights received in all *reply* messages to its own *weight*. When its weight becomes equal to one, it concludes that all processes involved in the checkpointing process have taken their tentative local checkpoints. As a result, it sends out *commit* messages to all processes from which it received *reply*s. Processes make their tentative checkpoints permanent on receiving the *commit* messages. The older permanent local checkpoints and all forced checkpoints at these processes are discarded because a process will never roll back to a point prior to the newly committed checkpoint. Note that when a process discards its forced checkpoints, it also updates its data structures such as *CP*, *R*, and *first*.

A formal description of the checkpointing algorithm is given below:

type trigger = record (pid, inum: integer;) **end**
var own_trigger, msg_trigger: trigger;
csn: array[1..*n*] of integers;
weight: real;
CP: array[1..*n*] of LIST;
process_set: set of integers;
R, temp1, temp2, first: bit array of size *n*;

Actions taken when P_i sends a computation message to P_j:
if $first_i[j] = 0$
then $first_i[j] := 1$; send(P_i, message, R_i, $csn_i[i]$, own_trigger);
else send(P_i, message, R_i, $csn_i[i]$, NULL);

Actions for the initiator P_j:
take a local checkpoint (on stable storage);
increment($csn_j[j]$); own_trigger := (P_j, $csn_j[j]$); clear process_set;
prop_cp(R_j, NULL, P_j, own_trigger, 1.0);

Other processes, P_i, on receiving a checkpoint request from P_j:
receive(P_j, *request*, *m.R*, recv_csn, msg_trigger, recv_weight);
$csn_j[j]$:= recv_csn;
if msg_trigger = own_trigger
then if $\exists\, k$ (msg_trigger $\in$ $CP[i][k]$)
  then save $CP[i][k]$.*forced* on stable storage;
    prop_cp(R_i, *m.R*, P_i, msg_trigger, recv_weight);
  else send(P_i, *reply*, recv_weight) to the initiator;
else if $first_i[] = 0$
  then send(P_i, *reply*, recv_weight) to the initiator;
  else take a local checkpoint (on stable storage);
    increment($csn_i[i]$); own_trigger := msg_trigger;
    prop_cp(R_i, *m.R*, P_i, msg_trigger, recv_weight);

Actions for process P_i, on receiving a computation message from P_j:
receive(P_j, m, m.*R*, recv_csn, msg_trigger);
if recv_csn $\le$ $csn_i[j]$
then process the message and exit;
else $csn_i[j]$:= recv_csn;
  if msg_trigger = own_trigger
  then process the message;
  else if $first_i[] = 0$
    then if $CP[i] \neq \phi$ **then**
      $CP[i][last]$.*trigger* := $CP[i][last]$.*trigger* $\cup$ msg_trigger
    else take a checkpoint, save it in $CP[i][last + 1]$.*forced*;
      $CP[i][last + 1]$.trigger := msg_trigger;
      $CP[i][last + 1].R := R_i$; reset R_i and $first_i$;
    own_trigger := msg_trigger; increment($csn_i[i]$);
    process the message;

prop_cp(R_i, *m.R*, P_i, msg_trigger, recv_weight)
if $\exists\, k$(msg_trigger $\in$ $CP[i][k]$.trigger)
then temp1 := $CP[i][k].R$;
  for j := 0 to k – 1 **do** temp1 := temp1 $\cup$ $CP[i][j].R$
else temp1 := R_i;
  for j := 0 to *last* **do** temp1 := temp1 $\cup$ $CP[i][j].R$
temp2 := temp1 $\ominus$ *m.R*; temp1 := temp1 OR *m.R*;
weight := recv_weight;
for any processes P_k, such that temp2[k] = 1
  weight := weight / 2;
  send(P_i, *request*, temp1, $csn_i[i]$, msg_trigger, weight);
send(P_i, *reply*, weight) to the initiator; reset R_i and $first_i$;

Actions in the second phase for the initiator P_i:
Receive(P_j, *reply*, recv_weight)
weight := weight + recv_weight;
process_set := process_set $\cup$ P_j;
if weight = 1
then for any P_k, such that $P_k \in$ *process_set*,
  send(*commit*, *msg_trigger*) to P_k;

Actions for other process P_j:
if Receive(*commit*, *msg_trigger*)
then make the tentative checkpoint permanent;
  find the *k*, such that (msg_trigger $\in$ $CP[i][k]$.trigger);
  for i := k + 1 to last **do** $R_j := R_j \cup CP[j][i].R$;
  for i := k + 1 to last **do** $first_j := first_j \cup CP[j][i]$.*first*;
  clear the forced checkpoint list.

4.4 An Example

The basic idea of the algorithm can be better understood by the example presented in Fig. 5. To initiate a checkpoint, P_2 takes its own checkpoint and sends checkpoint *request* messages to P_1, P_3 and P_4, since $R_2[1] = 1$, $R_2[3] = 1$, and $R_2[4] = 1$. When P_2's *request* reaches P_4, P_4 takes a checkpoint. Then, P_4 sends a computation message $m3$ to P_3. When $m3$ arrives at P_3, P_3 takes a forced checkpoint before processing it because $m3.csn > csn_3[4]$ and P_3 has sent a message during the current checkpoint interval. For the same reason, P_1 takes a forced checkpoint before processing $m2$.

P_0 has not communicated with other processes before it takes a local checkpoint. Later, it sends a computation message $m1$ to P_1. P_1 takes a checkpoint $C_{1,2}$ before processing $m1$, because P_0 has taken a checkpoint with checkpoint sequence number larger than P_1 expected and P_1 has sent $m4$ during the current checkpoint interval.

When P_1 receives the *request* from P_2, it searches its forced checkpoint list. Since $C_{1,1}$ is a forced checkpoint associated with P_2, P_2 turns $C_{1,1}$ into a tentative checkpoint by saving it on stable storage. Similarly, P_3 converts $C_{3,1}$ to a tentative checkpoint when it receives the checkpoint *request* from P_2. Finally, the checkpointing process initiated by P_2 terminates when checkpoints $C_{1,1}$, $C_{2,1}$, $C_{3,1}$, and $C_{4,1}$ are made permanent. P_1 discards $C_{1,2}$ when it makes checkpoint $C_{1,1}$ permanent.

4.5 Discussion

Checkpointing includes the time to trace the dependence tree and to save the states of processes on stable storage, which may be long. For a network of powerful workstations connected by a high-speed network, even a small blocking time may result in substantial lost computation, but taking an extra checkpoint does not consume much time on a powerful workstation and this time may be much less than the blocking time. Moreover, based on the empirical study of [3], [7], blocking the underlying computation may dramatically reduce the system performance. Thus, our nonblocking algorithm is much more efficient than blocking algorithms for a network of workstations.

Coordinated checkpointing algorithms suffer from the synchronization overhead associated with the checkpointing process. During the last decade, the communication overhead far exceeded the overhead of accessing stable storage [3]. Furthermore, the memory available to run processes tended to be small. These trade-offs naturally favored uncoordinated checkpointing schemes over coordinated checkpointing schemes. In modern systems, the overhead of coordinating checkpoints is negligible compared to the overhead of saving the states [7], [15]. Using concurrent and incremental checkpointing [7], the overhead of either coordinated or uncoordinated checkpointing is essentially the same. Therefore, uncoordinated checkpointing is not likely to be an attractive technique in practice given the negligible performance gains. These gains do not justify the complexities of finding a consistent recovery line after the failure, the susceptibility to the domino effect, the high stable storage overhead of saving multiple checkpoints of each process, and the overhead of garbage collection. Thus, our coordinated

checkpointing algorithm has many advantages over uncoordinated checkpointing algorithms.

Due to technological advancements, the synchronization cost is negligible compared to the overhead of saving the states on stable storage [7], [15]. Stable storage for checkpoints is provided by a highly available network file server. Checkpoints cannot be saved on a local disk or in local volatile or nonvolatile memory since that will make them inaccessible during an extended outage of the local machine. The cost of saving checkpoints on stable storage therefore includes both the cost of network transmission to the file server and the cost of accessing the stable storage device on the file server. Since most of the processes take checkpoints almost at the same time, the stable storage at the file server is more likely to be a bottleneck. Our algorithm requires minimum number of processes to take tentative checkpoints and thus minimizes the workload at the stable storage server. The "forced checkpoints" *do not* need to be saved on stable storage. They can be saved anywhere, even in the local volatile memory. Thus, taking a "forced checkpoint" avoids the cost of transferring large amounts of data to the stable storage and accessing the stable storage device, and thus it has much less overhead compared to taking a tentative checkpoint on stable storage. Also, by taking "forced checkpoints," our algorithm avoids the avalanche effect and significantly reduces the number of checkpoints.

In our algorithm, the number of "forced checkpoints" depends on the application. A process takes a "forced checkpoint" only when it receives a computation message which has a *csn* larger than the process expects. Further reducing the number of "forced checkpoints" will be our future work. Overall, our algorithm is efficient in the sense that it is nonblocking, requires minimum stable storage, minimizes the number of tentative checkpoints, and avoids the avalanche effect.

5 Correctness Proof

In Section 2.2, R_i represents all dependence relations in the current checkpoint period. Due to forced checkpoints, R_i represents all dependence relations after the last forced checkpoint or tentative checkpoint, whichever is later. Note that a process saves its current R_i and then clears it whenever it takes a forced checkpoint. As a result, we use Boolean array *temp*1 to recalculate the dependence relations in the current checkpoint interval, which is a sequence of union operations depending on the number of forced checkpoints taken. To simplify the proof, we use R_i to represent all dependence relations in the current checkpoint interval. If there are forced checkpoints in the current checkpoint period, R_i represents the value of *temp*1.

LEMMA 4. *If process P_i takes a checkpoint and $R_i[j] = 1$, P_j takes a checkpoint for the same checkpointing initiation.*

PROOF. If P_i is an initiator, it sends checkpoint *request* messages to all processes such that $R_i[j] = 1$. If P_i is not an initiator, it takes a checkpoint on receiving a *request* from a process P_k. For each process P_j such that $R_i[j] = 1$, there are two possibilities:

CASE 1. If m.R[j] = 0 in the *request* received by P_i from P_k, P_i sends a *request* to P_j.

CASE 2. If m.R[j] = 1 in the *request* received by P_i from P_k, a *request* has been sent to P_j by at least one process in the checkpoint *request* propagation path from the initiator to P_k.

Therefore, if a process takes a checkpoint, every process on which it depends receives at least one checkpoint *request*. There are three possibilities when P_j receives the first checkpoint *request*:

1) P_j has taken a tentative checkpoint for this checkpoint initiation when the first checkpoint *request* arrives: This *request* and all subsequent *request* messages for this initiation are ignored.
2) P_j has not taken a tentative checkpoint or a forced checkpoint for this checkpoint initiation when the first *request* arrives: P_j takes a checkpoint on receiving the *request*.
3) P_j has taken a forced checkpoint for this checkpoint initiation when the first checkpoint *request* arrives: P_j turns the forced checkpoint into a tentative checkpoint and all subsequent *request* messages for this initiation are ignored.

Hence, when a process takes a checkpoint, each process on which it depends also takes a checkpoint. These dependencies may have been present before the checkpointing process was initiated, or may have been created while the coordinated checkpointing process was in progress. □

THEOREM 2. *The algorithm creates a consistent global checkpoint.*

PROOF. We prove this by contradiction. Assume that the global state of the system is inconsistent at a time instance. Then, there must be a pair of processes P_i and P_j such that at least one computation message m has been sent from P_j after P_j's last checkpoint and has been received by P_i before P_i's last checkpoint. So, $R_i[j] = 1$ at P_i at the time of taking its checkpoint. From Lemma 4, P_j has taken a checkpoint. There are three possibilities under which P_j's checkpoint is taken:

CASE 1. P_j's checkpoint is taken due to a *request* from P_i. Then:

send(m) at $P_j \rightarrow$[1] receive(m) at P_i
receive(m) at $P_i \rightarrow$ checkpoint taken at P_i
checkpoint taken at $P_i \rightarrow$ *request* sent by P_i to P_j
request sent by P_i to $P_j \rightarrow$ checkpoint taken at P_j

Using the transitivity property of $\rightarrow$, we have: send(m) at $P_j \rightarrow$ checkpoint taken at P_j. Thus, the sending of m is recorded at P_j.

CASE 2. P_j's last checkpoint is taken due to a *request* from a process P_k, $k \neq i$. According to the assumption, P_j sends m after taking its local checkpoint which is triggered by P_k. When m arrives at P_i, there are two possibilities:

CASE 2.1. P_i has taken a forced checkpoint for this checkpoint initiation. Then, P_i processes m as

1. $\rightarrow$ is the "happened before" relation described in [12].

normal. When P_i receives a checkpoint *request* for this checkpoint initiation, P_i turns the forced checkpoint into a tentative checkpoint. As a result, the reception of m is not recorded in the checkpoint of P_i.

CASE 2.2. P_i has not taken a forced checkpoint for this checkpoint initiation. Since csn appended with m is greater than $P_i.csn[j]$, P_i takes a forced checkpoint before processing m. When P_i receives a checkpoint request for this checkpoint initiation, P_i turns the forced checkpoint into a tentative checkpoint. Certainly, the reception of m is not recorded in the checkpoint of P_i.

CASE 3. P_j's checkpoint is taken due to the arrival of a computation message m' at P_j from P_k. Similar to Case 2, we have a contradiction. □

THEOREM 3. *The checkpointing algorithm terminates within a finite time.*

PROOF. The proof is similar to [18]. □

6 Related Work

The first coordinated checkpointing algorithm was presented in [2]. However, it assumes that all communications between processes are atomic, which is too restrictive. The Koo-Toueg algorithm [10] relaxes this assumption. In this algorithm, only those processes that have communicated with the checkpoint initiator either directly or indirectly since the last checkpoint need to take new checkpoints. Thus, it reduces the number of synchronization messages and the number of checkpoints. Later, Leu and Bhargava [13] presented an algorithm which is resilient to multiple process failures. Also, this algorithm does not assume that the channel is *FIFO*, which is necessary in [10]. However, these two algorithms [10], [13] assume a complex scheme (such as slide window) to deal with the message loss problem and do not consider lost messages in checkpointing and recovery. Deng and Park [6] proposed an algorithm to address both orphan message and lost messages.

In Koo and Toueg's algorithm [10], if any of the involved processes is not able to or not willing to take a checkpoint, the entire checkpointing process is aborted. Kim and Park [9] proposed an improved scheme that allows the new checkpoints in some subtrees to be committed while the others are aborted.

To further reduce the system messages needed to synchronize the checkpointing, loosely synchronous clocks [5], [19] are used. More specifically, loosely synchronized checkpoint clocks can trigger the local checkpointing actions of all participating processes at approximately the same time without the need of broadcasting the checkpoint request by an initiator. However, a process taking a checkpoint needs to wait for a period that equals the sum of the maximum deviation between clocks and the maximum time to detect a failure in another process in the system.

All the above coordinated checkpointing algorithms [2], [5], [6], [9], [10], [13], [19] require processes to be blocked during checkpointing. Checkpointing includes the time to trace the dependence tree and to save the state of processes on stable storage, which may be long. Therefore, blocking algorithms may dramatically reduce system performance [3], [7].

The Chandy-Lamport algorithm [4] is the earliest nonblocking algorithm for coordinated checkpointing. However, in their algorithm, system messages (markers) are sent along all channels in the network during checkpointing. This leads to a message complexity of $O(n^2)$. Moreover, it requires all processes to take checkpoints and the channel must be *FIFO*. To relax the *FIFO* assumption, Lai and Yang [11] proposed another algorithm. In their algorithm, when a process takes a checkpoint, it piggybacks a checkpoint request (a *flag*) to the messages it sends out from each channel. The receiver checks the piggybacked message flag to see if there is a need to take a checkpoint before processing the message. If so, it takes a checkpoint before processing the message to avoid an inconsistency. To record the channel information, each process needs to maintain the entire message history on each channel as part of the local checkpoint. Thus, the space requirements of the algorithm may be large. Moreover, it requires all processes to take checkpoints, even though many of them are unnecessary.

In [25], when a process takes a checkpoint, it may continue its normal operation without blocking, because processes keep track of any delayed message. This algorithm is based on the idea of atomic send-receive checkpoints. Each sender and receiver make the balance between the messages exchanged, and keep the set of unbalanced messages as part of checkpoint data. However, this scheme requires each process to log each message sent, which may introduce some performance degradation and require the system to be deterministic.

The Elnozahy-Johnson-Zwaenepoel algorithm [7] uses a checkpoint sequence number to identify orphan messages, thus avoiding the need for processes to be blocked during checkpointing. However, this approach requires all processes to take checkpoints during checkpointing. The algorithm proposed by Silva and Silva [20] uses a similar idea as [7] except that the processes which did not communicate with others during the previous checkpoint interval do not need to take new checkpoints. Both algorithms [7], [20] assume that a distinguished initiator decides when to take a checkpoint. Therefore, they suffer from the disadvantages of centralized algorithms such as one-site failure, traffic bottle-neck, etc. Moreover, their algorithms require almost all processes to take checkpoints, even though many of them are unnecessary. If they are modified to permit more processes to initiate checkpointing processes, which makes them distributed, the new algorithm suffers from another problem; in order to keep the checkpoint sequence number updated, any time a process takes a checkpoint, it has to notify all processes in the system. If each process can initiate a checkpointing process, the network would be flooded with control messages and processes might waste their time taking unnecessary checkpoints.

All the above algorithms follow two approaches to reduce the overhead associated with coordinated checkpointing

algorithms: One is to minimize the number of synchronization messages and the number of checkpoints [2], [5], [6], [9], [10], [13], [19], the other is to make checkpointing nonblocking [4], [7], [11], [20]. These two approaches were orthogonal in previous years until the Prakash-Singhal algorithm [18] combined them. However, their algorithm has some problems and may result in inconsistencies. Therefore, our algorithm is the first correct algorithm to combine these two approaches. In other words, our nonlocking algorithm avoids the avalanche effect and significantly reduces the number of checkpoints.

7 Conclusions

Coordinated checkpointing has received considerable attention because of its low stable storage requirement and low rollback recovery overhead. However, it suffers from high overhead associated with the checkpointing process. A large number of algorithms have been proposed to reduce the overhead associated with coordinated checkpointing. Compared to these works, this paper has made three contributions to coordinated checkpointing:

1) Based on a new concept "z-dependence," we proved a more general result: There does not exist a nonblocking algorithm that forces only a minimum number of processes to take their checkpoints.
2) Different from the literature, a "forced checkpoint" in our algorithm is neither a tentative checkpoint nor a permanent checkpoint, but it can be turned into a tentative checkpoint. Based on this innovative idea, our nonblocking algorithm avoids the avalanche effect and significantly reduces the number of checkpoints.
3) We identified two problems in the Prakash-Singhal algorithm [18].

From Theorem 1, no nonblocking min-process algorithm exists. This implies that there are three directions in designing efficient coordinated checkpointing algorithms. On one extreme, we proposed a nonblocking algorithm, which relaxed the min-process condition while minimizing the number of tentative checkpoints; future work along this direction consists of further reducing the number of "forced checkpoints." The other extreme is to relax the nonblocking condition while keeping the min-process property, that is, we can design a min-process algorithm which tries to minimize the blocking time. Between these two extremes, we can also design blocking non-min-process algorithms that significantly reduce the blocking time as well as the number of checkpoints. Future research in designing coordinated checkpointing algorithms will consider these three approaches.

Since different applications may have different constraints, identifying suitable applications for these three approaches is valuable. For example, in mobile computing systems, due to vulnerability of Mobile Hosts (*MH*s) to catastrophic failures, e.g., loss, theft, or physical damage, the disk storage on an *MH* cannot be considered stable storage. A reasonable solution [1] is to utilize the stable storages at Mobile Support Stations (*MSS*s) to store checkpoints of *MH*s. Thus, to take a checkpoint, an *MH* has to transfer a large amount of data to its local *MSS* over the wireless network. Since the wireless network has low bandwidth, the min-process property is much more important in such environments. Most *MH*s have low computation power, a small amount of blocking time does not mean too much lost computation for them, but taking and transferring unnecessary checkpoints may waste a large amount of computation power, bandwidth, and energy. Moreover, the checkpointing may last much longer than the blocking time. Therefore, the min-process property may be more important than the nonblocking property in designing coordinated checkpointing algorithms for mobile computing systems. However, for a network of powerful workstations connected by a high-speed network, even a small blocking time may result in substantial lost computation, but taking an extra checkpoint does not consume much time on a powerful workstation and this time may be much less than the blocking time. Also, since there are no strict bandwidth and energy consumption requirements, the nonblocking property may be more important than the min-process property in designing coordinated checkpointing algorithms for a network of workstations.

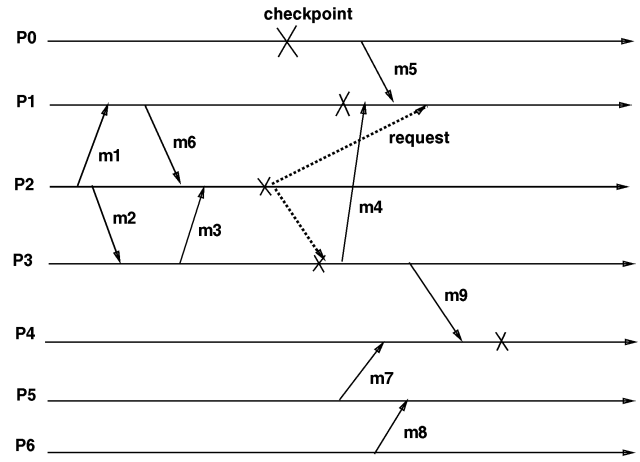

Fig. 6. The first problem.

Appendix

The coordinated checkpointing algorithm proposed by Prakash and Singhal [18] may result in inconsistencies. In the following, we identify two problems in this algorithm and discuss some possible solutions.

The First Problem

Prakash and Singhal [18] claim that their algorithm can take lazy checkpoints. In Section 4.4 of their paper, they gave the following example: As in Fig. 6, P_3 sends $m9$ to P_4 after P_3 finishes taking its checkpoint. When P_4 receives $m9$, it takes a new checkpoint (lazy checkpoint). Then, P_4 asks P_5 to take a checkpoint and P_5 asks P_6 to take a checkpoint. P_1, P_2, or P_3 do not send checkpoint *request* to P_4, P_5, and P_6, since P_1, P_2, and P_3 do not depend on P_4, P_5, or P_6. The checkpointing algorithm terminates when the weight of P_2 is equal to one. Because the computation message $m9$ does not carry any

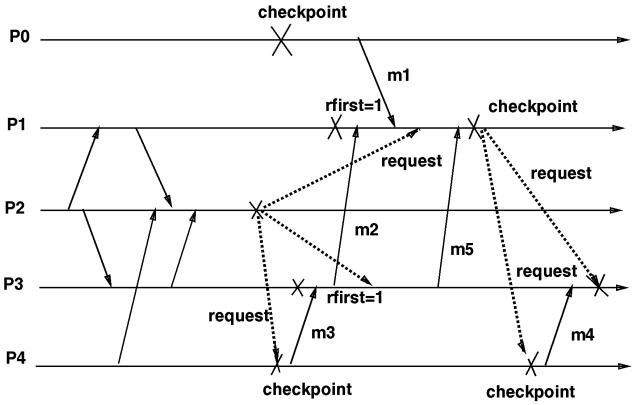

Fig. 7. The second problem.

weight, P_4, P_5, and P_6 send responses to the initiator with weight 0. Therefore, the termination of the algorithm is independent of their responses. Suppose P_2 finds out that its weight becomes equal to one before P_6 takes its checkpoint, but after P_2 has received the responses from P_4 and P_5. Then, P_2 sends *commit* messages to P_4 and P_5, but not P_6. As a result, P_4 and P_5 make their tentative checkpoints permanent. If P_6 fails before it finishes taking its local checkpoint, it results in an inconsistency.

A Solution to the First Problem

As presented in our algorithm, we solve this problem by using a forced checkpoint. The process taking a forced checkpoint does not propagate the checkpoint *request* until it receives the checkpoint *request* associated with this checkpoint initiation. In this example, P_4 only takes a forced checkpoint, it does not require P_5 and P_6 to take checkpoints. The forced checkpoint will be discarded since no process makes it permanent. Note that our algorithm only needs to propagate the checkpoint *request* once, but the Prakash-Singhal algorithm propagates the checkpoint *request* twice. However, lazy checkpointing is eliminated. Certainly, a process can always propagate lazy checkpoints by initiating another round of checkpointing. For example, after P_4 takes a lazy checkpoint, it initiates (as an initiator) another checkpointing process and makes them permanent after P_5 and P_6 have taken their checkpoints.

The Second Problem

We illustrate the second scenario using the example in Fig. 7. P_2 initiates a checkpointing process by taking its own checkpoint and sends checkpoint *request* messages to P_1, P_3, and P_4. When P_2's *request* reaches P_4, P_4 takes a checkpoint and sends message $m3$ to P_3. When $m3$ arrives at P_3, P_3 takes a checkpoint before processing the message, because $m3$ is the first message received by P_3 such that *msg_trigger.pid* ≠ *own_trigger.pid*. P_3 sets *rfirst* to 1. For the same reason, P_1 takes a checkpoint and sets *rfirst* to 1 before processing $m2$.

P_0 has not communicated with other processes before it takes a local checkpoint. Later, it sends a message $m1$ to P_1. Because P_0 has taken a checkpoint, its checkpoint sequence number is larger than P_1 expected. However, $m1$ is not the first computation message received by P_1 with a larger checkpoint sequence number than expected (*rfirst* = 1). Therefore, no checkpoint is taken.

Suppose P_1 initiates a checkpointing process after it receives $m5$. P_1 takes a checkpoint and then it sends checkpoint *request* messages to P_2, P_3, and P_4. When P_1's *request* reaches P_4, P_4 takes a checkpoint. Suppose P_4 sends a message $m4$ to P_3 after it finishes its checkpoint. Because $m4$ is not the first computation message received by P_3 with a larger checkpoint sequence number than expected (*rfirst* is already 1), P_3 does not take a checkpoint when it receives $m4$. Later, when P_3 receives the checkpoint *request* from P_1, it takes a checkpoint. As a result, $m4$ becomes an orphan.

Possible Solutions to the Second Problem

The problem arises due to the variable *rfirst*. In Fig. 7, one possible solution is to let P_3 clear *rfirst* before it receives $m4$. In this case, P_3 will take a checkpoint before processing $m4$. However, things are not that simple. In order to avoid the avalanche effect or unnecessary checkpoints, there is a need to keep the value of *rfirst* as long as possible. On the other hand, to avoid an inconsistency, there is a need to clear *rfirst* as soon as possible. Thus, we have a contradiction.

Another possible solution can be illustrated by Fig. 7. When P_4 sends $m4$, it appends the dependence information to $m4$. When P_3 receives $m4$, it knows that the initiator transitively depends on it, so P_3 takes a checkpoint before processing $m4$. However, we can easily construct a counterexample, such as that in Lemma 3. Therefore, to solve the second problem, we have to introduce extra checkpoints.

REFERENCES

[1] A. Acharya and B.R. Badrinath, "Checkpointing Distributed Applications on Mobil Computers," *Proc. Third Int'l Conf. Parallel and Distributed Information Systems*, Sept. 1994.

[2] G. Barigazzi and L. Strigini, "Application-Transparent Setting of Recovery Points," *Digest of Papers, Proc. 13th Fault Tolerant Computing Symp.* (*FTCS-13*), pp. 48–55, 1983.

[3] B. Bhargava, S.R. Lian, and P.J. Leu, "Experimental Evaluation of Concurrent Checkpointing and Rollback-Recovery Algorithms," *Proc. Int'l Conf. Data Eng.*, pp. 182–189, 1990.

[4] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Computer Systems*, Feb. 1985.

[5] F. Cristian and F. Jahanian, "A Timestamp-Based Checkpointing Protocol for Long-Lived Distributed Computations," *Proc. IEEE Symp. Reliable Distributed Systems*, pp. 12–20, 1991.

[6] Y. Deng and E.K. Park, "Checkpointing and Rollback-Recovery Algorithms in Distributed Systems," *J. Systems and Software*, pp. 59–71, Apr. 1994.

[7] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel, "The Performance of Consistent Checkpointing," *Proc. 11th Symp. Reliable Distributed Systems*, pp. 86–95, Oct. 1992.

[8] S.T. Huang, "Detecting Termination of Distributed Computations by External Agents," *Proc. Ninth Int'l Conf. Distributed Computing Systems*, pp. 79–84, 1989.

[9] J.L. Kim and T. Park, "An Efficient Protocol For Checkpointing Recovery in Distributed Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 8, pp. 955–960, Aug. 1993.

[10] 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.

[11] T.H. Lai and T.H. Yang, "On Distributed Snapshots," *Information Processing Letters*, pp. 153–158, May 1987.

[12] L. Lamport, "Time, Clocks and Ordering of Events in Distributed Systems," *Comm. ACM*, July 1978.

[13] P.Y. Leu and B. Bhargava, "Concurrent Robust Checkpointing and Recovery in Distributed Systems," *Proc. Fourth IEEE Int'l Conf. Data Eng.*, pp. 154–163, 1988.
[14] D. Manivannan, R. Netzer, and Mukesh Singhal, "Finding Consistent Global Checkpoints in a Distributed Computation," *IEEE Trans. Parallel and Distributed Systems*, vol. 8, no. 6, pp. 623–627, June 1997.
[15] G. Muller, M. Hue, and N. Peyrouz, "Performance of Consistent Checkpointing in a Modular Operating System: Results of the FTM Experiment," *Lecture Notes in Computer Science: Proc. First European Conf. Dependable Computing* (*EDCC-1*), pp. 491–508, Oct. 1994.
[16] R.H.B. Netzer and J. Xu, "Necessary and Sufficient Conditions for Consistent Global Snapshots," *IEEE Trans. Parallel and Distributed System*, vol. 6, no. 2, pp. 165–169, Feb. 1995.
[17] R. Prakash and M. Singhal, "Maximal Global Snapshot with Concurrent Initiators," *Proc. Sixth IEEE Symp. Parallel and Distributed Processing*, pp. 344–351, Oct. 1994.
[18] R. Prakash and M. Singhal, "Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems," *IEEE Trans. Parallel and Distributed System*, vol. 7, no. 10, pp. 1,035–1,048, Oct. 1996.
[19] P. Ramanathan and K.G. Shin, "Use of Common Time Base for Checkpointing and Rollback Recovery in a Distributed System," *IEEE Trans. Software Eng.*, vol. 19, no. 6, pp. 571–583, June 1993.
[20] L.M. Silva and J.G. Silva, "Global Checkpointing for Distributed Programs," *Proc. 11th Symp. Reliable Distributed Systems*, pp. 155–162, Oct. 1992.
[21] M. Spezialetti and P. Kearns, "Efficient Distributed Snapshots," *Proc. Sixth Int'l Conf. Distributed Computing Systems*, pp. 382–388, 1986.
[22] R.E. Strom and S.A. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Trans. Computer Systems*, pp. 204–226, Aug. 1985.
[23] Y. Wang, "Maximum and Minimum Consistent Global Checkpoints and Their Application," *Proc. 14th IEEE Symp. Reliable Distributed Systems*, pp. 86–95, Oct. 1995.
[24] Y. Wang, "Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints," *IEEE Trans. Computers*, vol. 46, no. 4, pp. 456–468, Apr. 1997.
[25] Z. Wojcik and B.E. Wojcik, "Fault Tolerant Distributed Computing Using Atomic Send Receive Checkpoints," *Proc. Second IEEE Symp. Parallel and Distributed Processing*, pp. 215–222, 1990.

Guohong Cao received his BS degree from Xian Jiaotong University, Xian, China. He received an MS degree in computer science from Ohio State University in 1997. He is currently a PhD candidate in the Department of Computer and Information Science at Ohio State University. His research interests include distributed computing systems, fault tolerance, mobile computing, and wireless networks.

Mukesh Singhal received a Bachelor of Engineering degree in electronics and communication engineering with high distinction from the University of Roorkee, Roorkee, India, in 1980, and a PhD degree in computer science from the University of Marland, College Park, in 1986. He is an associate professor of computer and information science at Ohio State University, Columbus. His current research interests include operating systems, distributed systems, mobile computing, high-speed networks, computer security, and performance modeling. He has published more than 100 refereed articles in these areas. He has coauthored two books: *Advanced Concepts in Operation Systems*, (McGraw-Hill, 1994) and *Readings in Distributed Computing Systems*, (IEEE CS Press, 1993). He has served on the program committees of several international conferences and symposiums. He was the program co-chair of the Sixth International Conference on Computer Communications and Networks in 1997 (IC3N '97) and is the program chair of the 17th IEEE Symposium on Reliable Distributed Systems, 1998.