On Consistent Checkpointing in Distributed Systems

Guohong Cao, Mukesh Singhal
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43201
E-mail: {gcao, singhal}@cis.ohio-state.edu

Abstract

Consistent checkpointing simplifies failure recovery and eliminates the domino effect in case of failure by preserving a consistent global checkpoint on the stable storage. However, the approach suffers from high overhead associated with the checkpointing process. Two approaches are used to reduce the overhead: one is to minimize the number of synchronization messages and the number of checkpoints; the other is to make the checkpointing process non-blocking. These two approaches were orthogonal in previous years until the Prakash-Singhal algorithm [17] 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. In this paper, we identify two problems in their algorithm [17] and prove that there does not exist a non-blocking algorithm that forces only a minimum number of processes to take their checkpoints. Based on the proof, we present an efficient algorithm that neither forces all processes to take checkpoints, nor blocks the underlying computation during checkpointing. Correctness proofs are also provided.

Key words: Distributed system, consistent checkpointing, dependency, non-blocking.

1 Introduction

The parallel processing capacity of a network of workstations is seldom exploited in practice. This is in part due to the difficulty in building application programs that can tolerate fully failures that are common in such environment. Consistent checkpointing is an attractive approach for transparently adding fault tolerance to distributed applications without requiring additional programmer efforts. The state of each process in a system is periodically saved on the 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 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 the stable storage. Other processes may have to rollback to their latest checkpoints on the 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 [3, 10, 21]. In order to record a consistent global checkpoint on the stable storage, processes must synchronize their checkpointing activities. In other words, when a process takes a checkpoint, it asks (by sending checkpoint requests to) all relevant processes to take checkpoints. Therefore, consistent checkpointing suffers from high overhead associated with the checkpointing process.

Much of previous work in consistent checkpointing has focused on minimizing the number of processes that must participate in taking a consistent checkpoint [5, 10, 13] or to reduce the number of messages required to synchronize the recording of a consistent checkpoint [22, 23]. However, these algorithms (called *blocking algorithm*) force all relevant processes in the system to block their computations during the checkpointing process. Checkpointing includes the time to trace the dependency tree and to save the states of processes on the stable storage, which may be long. Therefore, blocking algorithms may dramatically degrade the performance of the system [2, 6].

Recently, nonblocking algorithms [6, 19] 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 [6, 19] 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, these algorithms [6, 19] require all processes in the computation to take checkpoints during checkpointing, even though most of them may not be necessary.

Prakash-Singhal algorithm [17] 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 have found two problems in their algorithm. In this paper, we identify these two problems and formally prove that there does not exist a non-blocking algorithm that forces only a minimum number of processes to take their checkpoints. Based on this proof, we propose an efficient non-blocking algorithm, which minimizes the number of checkpoints during checkpointing.

The rest of the paper is organized as follows. Section 2 presents preliminaries. In Section 3, we prove that no non-blocking algorithm forces only a minimum number of processes to take checkpoints. Section 4 presents the algorithm. The correctness proofs are provided in Section 5. In Section 6, we compare the proposed algorithm with the existing algorithms. Section 7 concludes the paper. The 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: each process progresses at its own speed and messages are exchanged through reliable FIFO channels, whose transmission delays are finite but arbitrary.

The messages generated by the underlying distributed application will be referred to as *computation messages.* Messages generated by the processes to advance checkpoints will be referred to as *system messages.*

Each checkpoint taken by a process is assigned a unique sequence number. The $i^{th}(i \leq 0)$ checkpoint of process P_p is assigned a sequence number i and is denoted by $C_{p,i}$ (but sometimes we denote checkpoints as simply A, B, or C for clarity). 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 [15]. The i^{th} *checkpoint interval* of process P_p denotes all the computation performed between its i^{th} and $(i+1)^{th}$ checkpoint, including the i^{th} checkpoint but not the $(i+1)^{th}$ checkpoint.

2.2 Minimizing Dependency Information

In order to record the dependency relationship among processes, we use a similar approach as the Prakash-Singhal algorithm [17], where each process P_i maintains a boolean vector R_i. The vector has n bits, representing n processes. At P_i, the vector is initialized to 0.

When process P_i sends a message to P_j, it appends R_i to the message m. Receiving a message, P_j uses the appended $m.R$ to update its own R_j as follows: $R_j[k] = R_j[k] \vee m.R[k]$, where $1 \leq k \leq n$. Thus, if the sender of a message depends on a process P_k before sending the message, the receiver also depends on P_k on receiving the message. In Figure 1, P_2 depends on P_1 because of $m1$. When P_3 receives $m2$, P_3 (transitively) depends on $P1$.

The dependency 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 Figure 1, the vertical line S_1 represents the global consistent checkpoints at the beginning of the computation. Later, when P_1 initiates a new checkpoint collection, it only sends checkpoint requests to P_2 and P_3. As a result, only P_1, P_2, and P_3 take new checkpoints. P_4 and P_5 continues their computation without taking new checkpoints.

Unlike [7] and [14], which use a vector of n integers to record dependency information, we use bit vector to record dependency information, which significantly reduces space overhead and computation requirements.

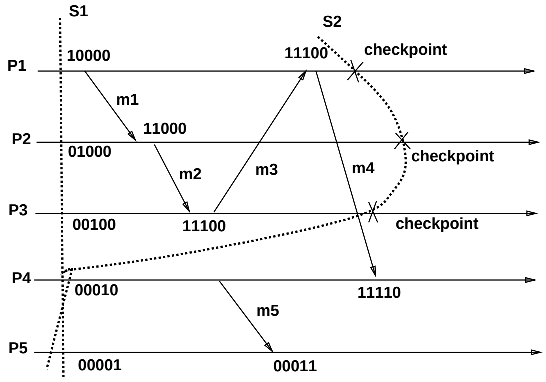

Figure 1: Checkpointing and dependency information

2.3 Basic Idea of Non-blocking Algorithms

Most of the existing consistent checkpointing algorithms [5, 10, 13] rely on the two-phase protocol and save two kinds of checkpoints on the 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. A process may say "no" depending on its underlying computation. When the initiator receives acknowledgments from all relevant processes, the algorithm enters the second phase. If the initiator learns that all the processes have successfully taken tentative checkpoints, it asks them to make their tentative checkpoints permanent; otherwise, it requires them to discard their tentative checkpoints. A process, on receiving the message from the initiator acts accordingly. Note that after a process takes a tentative checkpoint in the first phase, it remains blocked until it receives the decision from the initiator in the second phase.

Since a non-blocking algorithm does not require any process to suspend its underlying computation, it is possible for a process to receive a message from another process, which is already running in a new checkpoint interval, resulting in an inconsistency. For example, in Figure 2, P_2 initiates a checkpointing. 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. It is possible that P_3 receives $m1$ before it receives the checkpoint request from P_2. Then, P_3 processes $m1$ when it receives $m1$. When P_3 receives the checkpoint request from P_2, it takes a checkpoint. Then, $m1$ becomes an orphan.

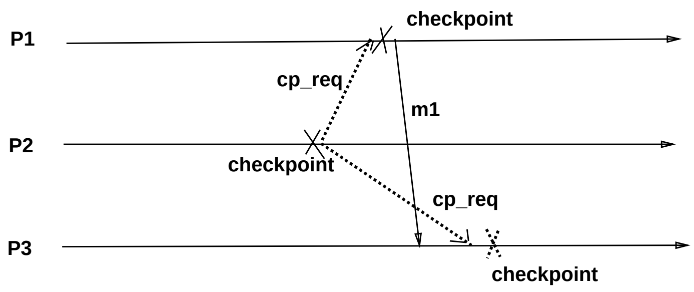

Figure 2: Missing a checkpoint

Most of the non-blocking algorithms [6, 19] use a Checkpoint Sequence Number (csn) to avoid inconsistency. More specifically, a process is forced to take a checkpoint if it receives a computation message whose appended csn is greater than its local csn. In Figure 2, P_1 increase its csn after it takes a checkpoint and appends the new csn to $m1$. When P_3 receives $m1$, it takes a checkpoint before processing $m1$ since the csn appended with $m1$ is larger than its local csn.

This scheme works only when every process in the computation can receive each checkpoint request and then increase its own csn. Since the Prakash-Singhal algorithm [17] only forces part of processes to take checkpoints, the csn of some processes may be out-of-date, and hence is insufficient to avoid inconsistency. the Prakash-Singhal algorithm attempts to solve this problem by having each process maintain an array to save the csn, where $csn[i]$ 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 is no communication between them during several checkpoint intervals. By using csn and the initiator identification number, they claim that their non-blocking algorithm can avoid inconsistency and minimize the number of checkpoints during checkpointing. However, we have found two problems in their algorithm (see Appendix). Next, we prove that there does not exist a non-blocking algorithm that forces only a minimum number of processes to take their checkpoints.

3 Proof of Impossibility

Definition 1 If P_p sends a message to 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 $P_p \prec^i_j P_q$.

Definition 2 If $P_p \prec^i_j P_q$, and $P_q \prec^j_k P_r$, 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 $P_p \overset{*\,i}{\prec}_k P_r$ (sometimes we simply call it "P_r transitively z-depends on P_p" if there is no confusion).

Proposition 1

$$P_p \prec^i_j P_q \Longrightarrow P_p \overset{*\,i}{\prec}_j P_q$$
$$P_p \overset{*\,i}{\prec}_j P_q \wedge P_q \overset{*\,j}{\prec}_k P_r \Longrightarrow P_p \overset{*\,i}{\prec}_k P_r$$

The definition of "z-depend" here is different from the concept of "depend" used in the literature. In Figure 3, since P_2 sends $m1$ before it receives $m2$, there is no dependent relation between P_1 and P_3. However, according to our definition:

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

In other words, P_1 transitively z-depends on P_3.

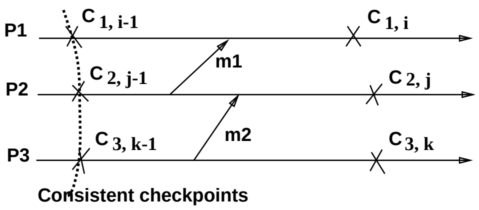

Figure 3: The difference between depend and z-depend

Definition 3 A *min-process* checkpointing algorithm is an algorithm satisfying the follows: when a process P_p initiates a new checkpointing 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 $P_q \overset{*\,j-1}{\prec}_{i-1} P_p$.

In consistent checkpointing, to keep consistency, the initiator forces all the dependent processes to take checkpoints. Each process, which takes a checkpoint, recursively forces all dependent processes to take checkpoints. The Koo-Toueg algorithm [10] uses this scheme, and it forces only a minimum number of processes to take checkpoints.

We illustrated the only difference between depend and z-depend using Figure 3. Next, we use Figure 3 to show that the Koo-Toueg algorithm [10] is a min-process algorithm. In Figure 3, according to [10], if P_1 takes $C_{1,i}$, P_1 asks P_2 to take $C_{2,j}$, and P_2 asks P_3 to take $C_{3,k}$. In the min-process checkpointing algorithm, since $P_2 \prec^{j-1}_{i-1} P1$ and $P_3 \overset{*}{\prec}{}^{k-1}_{i-1} P1$, P_2 takes $C_{2,j}$ and P_3 takes $C_{3,k}$. Assume P_2 receives P_1's checkpoint request before it receives $m2$. According to [10], if P_1 takes $C_{1,i}$, P_1 asks P_2 to take $C_{2,j}$. But P_2 does not ask P_3 to take a new checkpoint. In the min-process checkpointing algorithm, since there is only $P_2 \prec^{j-1}_{i-1} P1$, P_2 takes $C_{2,1}$ and P_3 does not take a new checkpoint.

The following is a formal proof of the above description. To simplify the proof, we use "$P_p \vdash^{j}_{i} P_q$" to represent the follows: P_q 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

$$P_p \vdash^{j}_{i} P_q \Longrightarrow P_p \overset{*}{\prec}{}^{j}_{i} P_q$$
$$P_p \prec^{j}_{i} P_q \Longrightarrow P_p \vdash^{j}_{i} P_q$$

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. We know that the Koo-Toueg algorithm [10] forces only a minimum number of processes to take checkpoints, so we need to prove the follows: in [10], when a process P_p initiates a new checkpointing 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 $P_q \overset{*}{\prec}{}^{j-1}_{i-1} P_p$.

Necessary: in [10], when a process P_p initiates a new checkpoint $C_{p,i}$, it recursively asks all dependent processes to take checkpoints. If a process P_q takes a checkpoint $C_{q,j}$ associated with $C_{p,i}$. Then, there must be a sequence:

$$P_q \vdash^{j-1}_{i_1} P_1 \wedge P_1 \vdash^{i_1}_{i_2} P_2 \wedge \cdots \wedge P_{n-1} \vdash^{i_{n-1}}_{i_n} P_n \wedge P_n \vdash^{i_n}_{i-1} P_p$$
$$\Longrightarrow P_q \overset{*}{\prec}{}^{j-1}_{i_1} P_1 \wedge P_1 \overset{*}{\prec}{}^{i_1}_{i_2} P_2 \wedge \cdots \wedge P_{n-1} \overset{*}{\prec}{}^{i_{n-1}}_{i_n} P_n \wedge P_n \overset{*}{\prec}{}^{i_n}_{i-1} P_p$$
$$\Longrightarrow P_q \overset{*}{\prec}{}^{j-1}_{i-1} P_p$$

Sufficient: if $P_q \overset{*}{\prec}{}^{j-1}_{i-1} P_p$, then there must be a sequence:

$$P_q \prec^{j-1}_{i_1} P_1 \wedge P_1 \prec^{i_1}_{i_2} P_2 \wedge \cdots \wedge P_{n-1} \prec^{i_{n-1}}_{i_n} P_n \wedge P_n \prec^{i_n}_{i-1} P_p$$
$$\Longrightarrow P_q \vdash^{j-1}_{i_1} P_1 \wedge P_1 \vdash^{i_1}_{i_2} P_2 \wedge \cdots \wedge P_{n-1} \vdash^{i_{n-1}}_{i_n} P_n \wedge P_n \vdash^{i_n}_{i-1} P_p$$

Then, when P_p initiates a new checkpoint $C_{p,i}$, P_p asks P_n to take a checkpoint, P_n asks P_{n-1} to take a checkpoint, etc. In the end, P_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 *min-process non-blocking* algorithm is a min-process checkpointing algorithm which does not block the underlying computation.

In a non-blocking algorithm, as discussed in Section 2.3, it is possible for a process to receive a message from another process, which is already running in a new checkpoint interval, resulting in an inconsistency. The following lemma states the necessary and sufficient conditions to avoid inconsistency.

Lemma 2 *In a min-process non-blocking algorithm, assume P_p initiates a new checkpointing 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 $P_q \overset{*\,j-1}{\prec}_{i-1} P_p$.*

Proof. According to the definition of min-process, P_q takes a checkpoint $C_{q,j}$ if and only if $P_q \overset{*\,j-1}{\prec}_{i-1} P_p$. Therefore, we only need to show that P_q should take $C_{q,j}$ before processing m. It is easy to see, if P_q takes $C_{q,j}$ after processing m, m becomes an orphan. □

From Lemma 2, when a process receives a message m, the receiver of m has to know if the initiator of a new checkpointing transitively z-depends on the receiver during the previous checkpoint interval.

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

Proof. The proof is by construction. In Figure 4, assume $m6$ and $m7$ do not exist. P_1 initiates a checkpointing. When P_4 receives $m4$, there is a z-dependency as follows:

$$P_2 \overset{*\,0}{\prec}_0 P_4 \wedge P_4 \overset{*\,0}{\prec}_0 P_1 \Longrightarrow P_2 \overset{*\,0}{\prec}_0 P_1.$$

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

Approach 1: tracing the in-coming messages. P_2 gets the new z-dependency information from P_1, so P_1 has to know the z-dependency information before it sends $m5$ and appends the z-dependency information to $m5$. In Figure 4, P_1 can not get the new z-dependency information ($P_2 \overset{*\,0}{\prec_0} P_1$) unless P_4 notifies P_1 of the new z-dependency information when P_4 receives $m4$; P_4 either broadcasts the z-dependency information (not illustrated in the figure) or sends the z-dependency information by an extra message $m6$ to P_3, which in turn sends to $P1$ 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 in the figure). Then, P_2 still can not get the z-dependency information when it receives $m5$.

Approach 2: tracing the out-coming messages. In this approach, P_2 gets the dependency information from P_5 by extra messages since it sends $m3$ to P_5, so P_5 has to get the dependency 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 can not get the z-dependency information in time since the computation is in progress. □

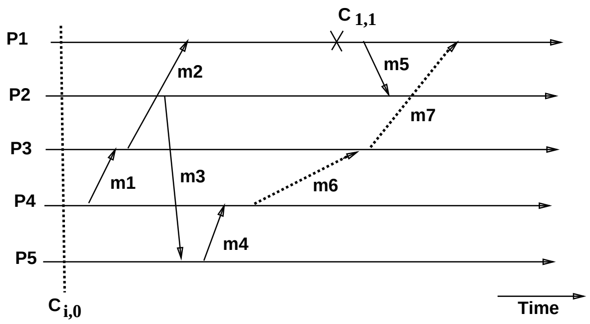

Figure 4: Tracing the dependency

Theorem 1 *No min-process non-blocking algorithm exists.*

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

Corollary 1 *No non-blocking algorithm forces only a minimum number of processes to take checkpoints.*

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

4 A Checkpointing Algorithm

From Theorem 1, no min-process non-blocking algorithm exists. Therefore, there are two directions in designing efficient consistent checkpointing algorithms. First is to relax the non-blocking condition while keeping the min-process property. The other is to relax the min-process condition while keeping the non-blocking property. Based on the empirical study of [2, 6], blocking the underlying computation may dramatically reduce the performance of the system. Therefore, we develop an algorithm that relaxes the min-process condition; that is, it is non-blocking, but it may force extra processes to take checkpoints. It, however, minimizes the number of extra checkpoints.

4.1 Basic Idea

Basic Scheme

A simple non-blocking, but not min-process, scheme for checkpointing is as follows: when a process P_i sends a message, it piggybacks the current value of $csn_i[i]$. When a process P_j receives a 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, and then processes the message. Obviously, this method may result in a large number of forced checkpoints. Moreover, it may lead to an *avalanche effect*, in which processes in the system recursively ask others to take checkpoints.

For example, in Figure 5, to initiate a checkpointing, 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 because $m3.csn > csn_3[4]$. 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 message $m1$ to P_1. P_1 takes $C_{1,2}$ before processing $m1$, because P_0 has taken a

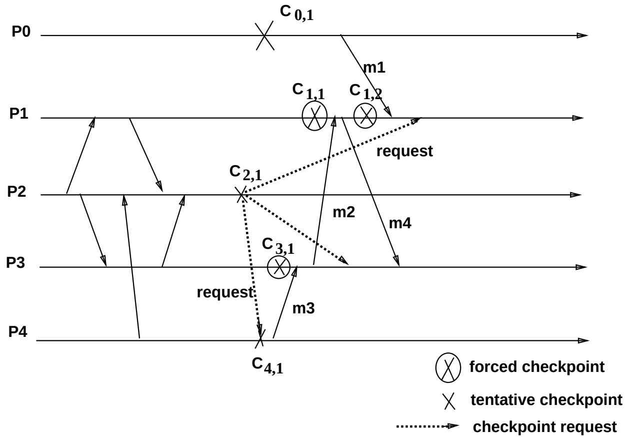

Figure 5: An example of checkpointing

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 $m3$, then those processes would have been forced to take checkpoints. This chain may never end.

In Figure 5, if $m4$ does not exist, it is not necessary for P_1 to take $C_{1,2}$, since P_1 did not send any message after taking $C_{1,1}$. Based on this observation, we get the following scheme.

Revised Basic Scheme

"When a process P_j receives a 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 Figure 5, if $m4$ does not exist, $C_{1,2}$ is not necessary in the revised scheme. However, if $m4$ exists, the revised scheme still results in a large number of forced checkpoints and may result in avalanche effect.

Enhanced Scheme

We now present the basic idea of our scheme that eliminates avalanche effect during checkpointing. From Figure 5, we have two observations:

Observation 1: It is not necessary to take $C_{1,2}$ since P_0 does not transitively z-depend on P_1 during the 0^{th} 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 still unavoidable. We observe that there are two kinds of forced checkpoints. In Figure 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. Therefore, P_1 should keep $C_{1,1}$ when P_1 receives P_2's *request*. P_1 can discard $C_{1,2}$ after a period (defined as the longest checkpoint *request* travel time). If only one checkpointing 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; they are saved on the stable storage. Forced checkpoints are not saved on the stable storage. 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.

Assume 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. Then, P_i forces all the processes, on which P_i depends before it took the forced checkpoint, 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 even not finished its checkpointing after a period defined as before; if so, P_i stops the unnecessary checkpointing, and then continues its computation.

In Figure 5, $C_{1,2}$ is a forced checkpoint that will be discarded in the future. Since $C_{1,2}$ is just 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 performance in checkpointing.

4.2 Data Structures

The following data structures are used in our algorithm:

R: 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 non-negative variable of type real with maximum value of 1. It is used to detect the termination of the checkpointing 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 1.

$trigger_i$: 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. $trigger$ is appended to each system message and the first computation message that a process sends to every other processes after taking a 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 k^{th} 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 0's at all processes. The trigger tuple at process P_i is initialized to $(i, 0)$. The weight at a process is initialized to 0. 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 1 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 is in progress. Techniques to handle concurrent initiations of checkpointing by multiple processes can be found in [16]. As multiple concurrent initiations of checkpointing is orthogonal to our discussion, we only briefly mention the main features of [16]. When a process receives its first request for checkpointing initiated by another process, it takes a local checkpoint and propagates the request. All the 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 [20]. Therefore, the amount of computation lost during rollback, after process failures, is minimized. Next, we present our non-blocking algorithm.

Checkpointing initiation: Any site can initiate a checkpointing. When a process P_i initiates a checkpointing, it takes a local checkpoint, increments its $csn_i[i]$, sets *weight* to 1, and stores its own identifier and the new $csn_i[i]$ in its trigger. Then, it sends checkpoint *request* to each process P_j, such that $R_i[j]$=1 and resumes its computation. Each request message 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 request: 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), P_i does not need to take a checkpoint. But,

- If there is a forced checkpoint which has a trigger equal to $msg\_trigger$, P_i saves the forced checkpoint on the 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 which has a trigger equal to $msg\_trigger$, then a *reply* is sent to the initiator with the weight equal to the weight received in the *request*.

Computation messages received during checkpointing: When P_i receives a computation message 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 the message, and this message is the first computation message sent by P_j to P_i since P_j took its checkpoint. Therefore, the message 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 event. Therefore, no new local checkpoint is needed.

- If $msg\_trigger \neq own\_trigger$, 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 the $msg\_trigger$ 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 1, it concludes that all the processes involved in the checkpointing have taken their tentative local checkpoints. Then, it sends out *commit* messages to all the processes from which it received *reply*s. The processes make their tentative checkpoints permanent on receiving the *commit* message. 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:

The checkpointing algorithm

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 the 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 the 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 $\leq csn_i[j]$ *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 local checkpoint, save it in $CP[i][last+1].forced$;
$CP[i][last+1]$.trigger=msg_trigger; $CP[i][last+1].R = R_i$;
own_trigger=msg_trigger; increment($csn_i[i]$);
reset R_i and $first_i$; }

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 all 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($make\_permanent, msg\_trigger$) to P_k;

Actions for other process P_j:
if Receive($make\_permanent, msg\_trigger$) *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 Figure 5. To initiate a checkpointing, 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 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 message $m1$ to P_1. P_1 takes $C_{1,2}$ 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 the 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 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.

5 Correctness Proof

In Section 2.2, R_i represents all the dependency relations in the current checkpoint period. Due to the introduction of forced checkpoints, R_i represents all dependency 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. So, we used boolean array $temp1$ to recalculate the dependency relations in the current checkpoint period, 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 dependency relations in the current checkpoint period. If there are forced checkpoints in the current checkpoint period, R_i represents the value of $temp1$.

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

Proof. If P_i initiates a checkpointing, it sends checkpoint *request* messages to all process such that $R_i[j]$=1. If P_i is not the initiator and takes a checkpoint on receiving a *request* from a process P_k, then 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, then 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, then 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, every process on which it depends also takes a checkpoint. These dependencies may have been present before the checkpointing was initiated, or may have been created while the consistent checkpointing was in progress. $\square$

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

Proof. Assume the contrary. Then, there must be a pair of processes P_i and P_j such that at least one 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 possible situations 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 $P_j \longrightarrow$ [1] receive(m) at P_i

receive(m) at $P_i \longrightarrow$ checkpoint taken at P_i

checkpoint taken at $P_i \longrightarrow$ *request* sent by P_i to P_j

request sent by P_i to $P_j \longrightarrow$ checkpoint taken at P_j

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

Case 2: P_j's 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 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 [17]. □

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

6 Related Work

The first consistent checkpointing algorithm was presented in [1]. However, it assumes that all communications between processes are atomic, which is too restrictive. The Koo-Toueg algorithm [10] relaxes this assumption. In [10], 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, and 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 [5] proposed an algorithm, which addresses both orphan message and lost messages.

In Koo and Toueg's original scheme [10], if any of the involved processes is not able or not willing to take a checkpoint, then 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 [4, 18] 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 to 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 consistent checkpointing algorithms [1, 4, 5, 9, 10, 13, 18] require processes to be blocked during checkpointing. Checkpointing includes the time to trace the dependency tree and to save the state of processes on the stable storage, which may be long. Therefore, blocking algorithms may dramatically reduce the performance of the system [2, 6].

The Chandy-Lamport algorithm [3] is the earliest non-blocking algorithm for consistent checkpointing. However, in their algorithm, system messages are sent along all the 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 proposed another algorithm [11]. In their algorithm, when a process takes a checkpoint, it piggybacks a checkpoint request to the first message it sends out from each channel. The receiver checks the piggybacked message before processing the message. Their algorithm still requires checkpoint request messages to be sent from each channel, and it also requires all process to take checkpoints, even though most of them are unnecessary.

In [24], when a process takes a checkpoint it may continue its normal operation without blocking, because processes keep track of any delayed message. Their 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 [6] uses the checkpoint sequence number to identify orphan messages, thus avoiding the need for processes to be blocked during checkpointing. However, this approach requires the initiator to communicate with all processes in the computation. The algorithm proposed by Silva and Silva [19] uses a similar idea as [6], except that the processes which did not communicate with others during the previous checkpoint period do not need to take a new checkpoint. Both algorithms [6, 19] 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 most of them are unnecessary. If they are modified to permit more processes to initiate checkpointing, which makes them truly 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, 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 consistent checkpointing algorithms: one is to minimize the number of synchronization messages and the number of checkpoints [1, 4, 5, 9, 10, 13, 18]; the other is to make checkpointing non-blocking [3, 6, 11, 19, 24]. These two approaches were orthogonal in previous years until the Prakash-Singhal algorithm [17] combined them. However, their algorithm has two problems. Therefore, our algorithm is the first correct algorithm to combine these two approaches. In other words, our non-locking algorithm minimizes the number of extra checkpoints in checkpointing.

Netzer and Xu introduced the concept of "zigzag" paths [15] to define the necessary and sufficient conditions of consistent checkpoints. Our definition of "z-depend" catches 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, then there is no "zigzag" path among them, and all processes on which the initiator "transitively z-depends" have taken checkpoints. However, there is distinctive difference between a "zigzag" path and "z-depend". The "zigzag" path is used to evaluate whether the existing checkpoints are consistent, so it is mainly used in independent checkpointing algorithms to find a consistent recovery line. It has almost no use in consistent checkpointing algorithms since a consistent recovery line is guaranteed by the synchronization messages. The "z-depend" is proposed for consistent checkpointing algorithms and it reflects the whole synchronization process of consistent checkpointing algorithms, not only the result. Based on "z-depend", we found and proved the impossibility result in Section 3. It is impossible to prove the result only based on "zigzag" paths.

7 Conclusions

Consistent checkpointing has received considerable attention because of its low stable storage overhead 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 consistent checkpointing. Compared to these consistent checkpointing algorithms, this paper has made three contributions to consistent checkpointing:

1. Using a new concept "z-depend", we proved that no nonblocking algorithm forces only a minimum number of processes to take checkpoints, and point out two directions in designing efficient consistent checkpointing algorithms.

2. Different from the literature, the "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 non-blocking algorithm avoids the avalanche effect and minimizes the number of checkpoints.

3. We identified two problems in the Prakash-Singhal algorithm[17].

From Theorem 1, no min-process non-blocking algorithm exists. Therefore, there are two directions in designing efficient consistent checkpointing algorithms. In this paper, we proposed a non-blocking algorithm, which relaxed the min-process condition while minimizing the number of checkpoints. In the future, we will propose a min-process algorithm, which relaxes the non-blocking condition while minimizing the blocking time.

References

[1] G. Barigazzi and L. Strigini. "Application-Transparent Setting of Recovery Points". *Digest of Papers FTCS-13*, pages 48–55, 1983.

[2] B. Bhargava, S.R. Lian, and P.J. Leu. "Experimental Evaluation of Concurrent Checkpointing and Rollback-Recovery Algorithms". *Proceedings of the International Conference on Data Engineering*, pages 182–189, 1990.

[3] K.M. Chandy and L. Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". *ACM Transactions on Computer Systems*, February 1985.

[4] F. Cristian and F. Jahanian. "A timestamp-based Checkpointing Protocol for long-lived Distributed Computations". *Proc. of IEEE Symp. Reliable Distributed Syst.*, pages 12–20, 1991.

[5] Y. Deng and E.K. Park. "Checkpointing and Rollback-Recovery Algorithms in Distributed Systems". *Journal of Systems and Software*, April 1994.

[6] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. "The Performance of Consistent Checkpointing". *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 86–95, October 1992.

[7] J. Fidge. “Timestamps in Message-Passing Systems that Preserve the Partial Ordering”. *Proceedings of* 11^{th} *Australian Computer Science Conference*, pages 56–66, 1988.

[8] S.T. Huang. “Detecting Termination of Distributed Computations by External Agents”. *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 79–84, 1989.

[9] J.L. Kim and T. Park. “An Efficient Protocol For Checkpointing Recovery in Distributed Systems”. *IEEE Transactions on Parallel and Distributed Systems*, pages 955–960, August 1993.

[10] R. Koo and S. Toueg. “Checkpointing and Rollback-Recovery for Distributed Systems”. *IEEE Transactions on Software Engineering*, pages 23–31, January 1987.

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

[12] L. Lamport. “Time, Clocks and Ordering of Events in Distributed Systems”. *Communication of the ACM*, July 1978.

[13] P.Y. Leu and B. Bhargava. “Concurrent Robust Checkpointing and Recovery in Distributed Systems”. *Pro. 4th IEEE Int. Conf. on Data Eng.*, pages 154–163, 1988.

[14] F. Mattern. “Virtual Time and Global States of Distributed Systems”. *Proceedings of the workshop on Parallel and Distributed Algorithm*, pages 215–226, 1993.

[15] R.H.B. Netzer and Jian Xu. “Necessary and Sufficient Conditions for Consistent Global Snapshots”. *IEEE Transactions on Parallel and Distributed System*, pages 165–169, February 1995.

[16] Ravi Prakash and Mukesh Singhal. “ Maximal Global Snapshot with Concurrent Initiators”. *Proceedings of the Sixth IEEE symposium on Parallel and Distributed Processing*, pages 344–351, October 1994.

[17] Ravi Prakash and Mukesh Singhal. “Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems”. *IEEE Transactions on Parallel and Distributed System*, pages 1035–1048, October 1996.

[18] P. Ramanathan and K.G. Shin. “Use of Common Time Base for Checkpointing and Rollback Recovery in a Distributed System”. *IEEE Transactions on Software Engineering*, pages 571–583, June 1993.

[19] L.M. Silva and J.G. Silva. “Global Checkpointing for Distributed Programs”. *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 155–162, October 1992.

[20] M. Spezialetti and P. Kearns. “Efficient Distributed Snapshots”. *Proceedings of the* 6^{th} *International Conference on Distributed Computing Systems*, pages 382–388, 1986.

[21] R.E. Strom and S.A. Yemini. "Optimistic Recovery In Distributed Systems". *ACM Transactions on Computer Systems*, pages 204–226, August 1985.

[22] Z. Tong, R.Y. Kain, and W.T. Tsai. "A Lower Overhead Checkpointing and Rollback Recovery Scheme for Distributed Systems". *Proceedings of the 8th Symposium on Reliable Distributed Systems*, pages 12–20, October 1989.

[23] K. Venkatesh, T. Radhakrishnan, and H.F. Li. "Optimal Checkpointing and Local Recording for Domino-free Rollback Recovery". *Information Processing Letters*, pages 25:295–303, July 1987.

[24] Z. Wojcik and B.E. Wojcik. "Fault Tolerant Distributed Computing Using Atomic Send Receive Checkpoints". *Proc. 2nd IEEE Symp. on Parallel and Distributed Processing*, pages 215–222, 1990.

APPENDIX

We point out two problems in the Prakash-Singhal algorithm [17] and discuss some possible solutions.

Preliminaries for the Prakash-Singhal Algorithm

Similar to other two-phase algorithms, the Prakash-Singhal Algorithm also has two phases. In the first phase, an initiator takes a tentative checkpoint and forces the checkpointing of other processes on which it depends. In the second phase, all involved processes make their tentative checkpoints permanent.

When a process P_i receives a checkpoint *request* from P_j, it compares the $P_j.trigger$ ($msg\_trigger$) with $P_i.trigger$ ($own\_trigger$). If these two triggers are different, P_i takes a tentative checkpoint, updates its local information, and forwards the *request* to all the processes on which it depends.

If $msg\_trigger$ is equal to $own\_trigger$ when P_i receives the *request*, P_i does not need to take a checkpoint because it has already taken a checkpoint for this checkpoint initiation.

To deal with the problem associated with non-blocking algorithm discussed in Section 2.3, the Prakash-Singhal Algorithm requires each process to include its trigger in the first computation message that it sends to every other process after taking its checkpoint. When a process say P_j receives a computation message m from P_i, it compares the interval_number received in the message with its own interval_vector[i]. If the interval_number received is less than or equal to interval_vector[i], then the message is processed and no checkpoint is taken. If the interval_number of the computation message received is greater than interval_vector[i], it implies that P_i took a checkpoint before sending the message, and this message is the first computation message sent by P_i to P_j since P_i's snapshot. So, the message must have a trigger tuple. P_j updates its interval_vector, and does the following:

- 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 event. Therefore, no new local checkpoint is needed.

- If $msg\_trigger.pid = own\_.trigger.pid \wedge msg\_trigger.inum > own\_trigger.inum$, it means that P_i has sent the message after taking a new checkpoint, while P_j has not taken a checkpoint for this checkpointing. Therefore, P_j takes a checkpoint before processing this message, updates its local information, and propagates the checkpoint request. Note that $rfirst$ is set to 1.

- If $msg\_trigger.pid \neq own\_trigger.pid$, P_j executes as follows: If P_j has not processed any message satisfying the condition $msg\_trigger.pid \neq own\_trigger.pid$ since its last local checkpoint ($rfirst$=0), P_j takes a checkpoint before processing this message, updates its local information, and propagates the checkpoint request. Note that $rfirst$ is set to 1. Otherwise, if P_j has already processed a message from any process satisfying the condition $msg\_trigger.pid \neq own\_trigger.pid$ since its last local checkpoint ($rfirst$=1), no new local checkpoint is needed.

The First Problem

They claim that their algorithm can take a lazy checkpoint. In Section 4.4 of their paper, they gave the following example: As in Figure 6, P_3 sends $m9$ to P_4 after P_3 finishes 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. Since P_1, P_2 and P_3 do not casually depend on P_4, P_5 or P_6, P_1, P_2 or P_3 does not send checkpoint *request* to P_4, P_5, and P_6. The algorithm terminates when the weight of P_2 is equal to 1. Because the computation message $m9$ does not carry any 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 1 before P_6 takes

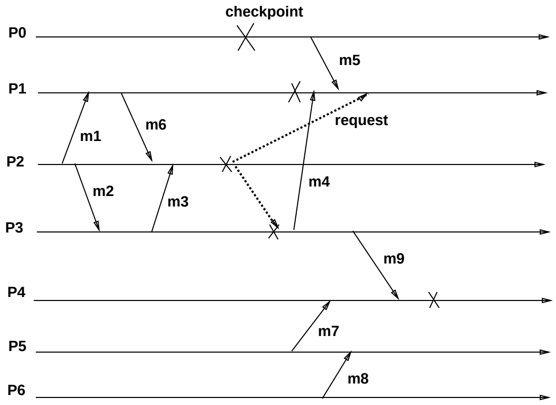

Figure 6: The first problem

its checkpoint, but after P_2 has received the responses from P_4 and P_5. Then, P_2 sends *make_permanent* 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 its local checkpoint, it results in an inconsistency.

Solution to the first problem As presented in our algorithm, we solve this problem by using a forced checkpoint. The process taking 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 checkpoints 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.

The Second Problem

We identify the second problem using a counter-example. In Figure 7, P_2 initiates a checkpointing 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, then P_4 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 $msg\_trigger.pid \neq own\_trigger.pid$. P_3 sets $rfirst$ to 1. For the same reason, P_1 takes a checkpoint before processing $m2$, and P_1 sets $rfirst$ to 1.

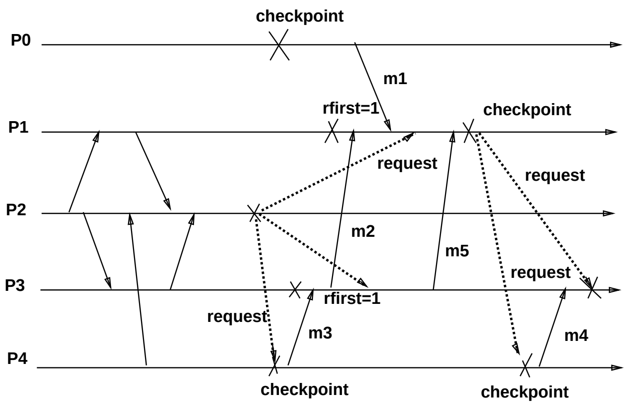

Figure 7: The second problem

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, a checkpoint is not needed.

Suppose P_1 initiates a checkpointing after it receives $m5$. P_1 takes a checkpoint, then it sends checkpoint *request* messages to 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. Thus, $m4$ becomes an orphan.

Possible solutions to the second problem The problem arises due to variable $rfirst$. In Figure 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 inconsistency, there is a need to clear $rfirst$ as soon as possible. So, we have a contradiction.

Another possible solution can be illustrated by Figure 7. When P_4 sends $m4$, it appends the dependency 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 counter-example, such as in Lemma 3. Therefore, to solve the second problem, we have to introduce extra checkpoints.