Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Systems
October 16-19, 1996 Chicago, Illinois, USA

96-nc

A Nonblocking Consistent Checkpointing Algorithm for Distributed Systems *

Guohong Cao
Computer and Information Science
The Ohio-State University
Columbus, OH43201
gcao@cis.ohio-state.edu

Naphtali Rishe
School of Computer Science
Florida International University
Miami, FL 33199
rishen@fiu.edu

Abstract

Consistent checkpointing simplifies failure recovery and eliminates the domino effect in case of failure by preserving a consistent global checkpoint in stable storage. However, the approach suffers from high overhead associated with the checkpointing process. This paper presents an efficient non-block scheme to address this problem. In the proposed scheme, a checkpoint sequence number vector is used to identify orphan messages; as a result, processes involved in checkpointing need not to be blocked. Based on inter-process dependencies created since the last checkpointing, our scheme only forces a minimal set of processes to take their checkpoints. It is shown that the proposed algorithm ensures the global state consistency of the distributed system.

1 Introduction

The parallel processing capacity of a network of workstations is seldom exploited in practice. This is due in part to the difficulty of building application programs that can tolerate the failures that are common in such environments. Consistent checkpointing is an attractive approach for transparently adding fault tolerance to distributed applications without requiring additional programmer effort. With consistent checkpointing, the state of each process in a 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 state recorded by the checkpoints of the processes. More specifically, the failed processes are restarted on any available machine and their address space is restored from their latest checkpoints on stable storage. Other processes may have to rollback to their latest checkpoints on stable storage in order to remain consistent with the recovering processes.

A system state is said to be consistent if it contains no *orphan message*; i.e., a message whose receiving event is recorded in the state of the destination process, but its sending event is lost [3, 8, 13]. In order to record a consistent global checkpoint in stable storage, processes must be synchronized during checkpointing. In other words before 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 the previous work in consistent checkpointing has focused on minimizing the number of processes that must participate in taking a consistent checkpoint [4, 8, 9] or to reduce the number of messages required to synchronize the consistent checkpoint [14, 15]. However these algorithms (called blocking algorithm) force all relevant processes in the system to freeze their computations during the checkpointing process. A checkpointing process includes the time to trace the dependency tree and save the state of processes on stable storage, which needs a long time. Therefore, blocking algorithms dramatically reduce the performance of the system [2, 5].

Recently, some nonblocking algorithms [5, 12] have received considerable attention. They avoid the need for processes to be blocked during checkpointing by using a checkpointing sequence number to identify orphan messages. However, these algorithms [5, 12] 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, bottle-neck, etc. If they are modified to permit other sites to initiate a checkpoint, which makes them truly distributed, the new algorithm suffers from another problem as follows. 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 every process can initiate checkpointing, the network would be flooded with control messages and processes might waste their time making unnecessary checkpoints.

In this paper, we provide an efficient non-block distributed checkpointing algorithm to reduce the overhead associated with the checkpointing process. The proposed

*This research was supported in part by NASA (under grant NAGW-4080), ARO (under grant DAH04-96-1-0049), BMDO (under ARO grant DAAH04-0024), NATO (under grant HTECH.LG-931449), and NSF (under grant CDA-9313624 for CATE Lab).

algorithm avoids the need for processes to be blocked during checkpointing and forces only a minimal set of processes to take their local checkpoints, based on inter-process dependencies created since the last checkpointing.

The rest of the paper is organized as follows. Section 2 describes the system model. In Section 3, we present the algorithm. The correctness proof is provided in Section 4. In Section 5, we compare the proposed algorithm with the existing algorithms. Section 6 concludes the paper.

2 Computation Model

The system is composed of a set of communicating processes executing on a collection of fail stop processors. The processes are connected by a communication network that is not subject to network partitions, and the processes can only communicate with each other through message passing. It is assumed that the communication system is reliable; i.e. a message sent will be received correctly in finite time. However, messages may be duplicated or delivered out-of-order.

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*.

3 The Algorithm

Our algorithm has two phases. In the first phase, an initiator makes a tentative checkpoint and forces every process on which it causally depends to take a tentative checkpoint. After the initiator receives acknowledgments from all the processes on which it depends, the algorithm enters the second phase in which all these processes change their tentative checkpoints to be permanent.

3.1 Basic Ideas

Because the 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 inconsistency. Most of algorithms [5, 12] use a Checkpoint Sequence Number (*csn*) to avoid inconsistency. More specifically, a process takes a checkpoint if it receives an application message whose appended *csn* is greater than the local *csn*. However this scheme only works when every process in the computation can receive each checkpoint request and then increase its own *csn*.

Since our algorithm forces only the causally dependent processes to take checkpoints, the *csn* of some processes may be out-of-date, and hence insufficient to avoid inconsistency. To deal with this problem, each process has an array to save the *csn* of all processes in the computation, where $csn[i]$ is the expected *csn* of P_i. Note that P_i's $csn[i]$ may be different from P_j's $csn[j]$ if there is no communication between them during several checkpoint periods. By using the *csn* and the initiator identification number (*id*), we can avoid inconsistency and unnecessary checkpoints during the checkpointing.

Huang's algorithm [6] has been modified to detect the termination of the first phase in our algorithm. When a process (the initiator) initiates a checkpointing, it sets its weight to 1, then sends checkpoint *request* message to all the processes on which it depends. Each *request* message carries a portion of the weight of the sender, which is decreased by an equal amount after sending a *request*. When a process P_i receives a *request* from P_j, P_i forwards the *request* to all the sites on which it depends, but P_j does not depend. Similarly, P_i also appends a portion of its received weight to the outgoing *request*. Finally, P_i takes a tentative checkpoint, and sends a *reply* message appending the remaining portion of its received weight to the initiator. Receiving a reply, the initiator adds the appended weight to its own weight. If the sum is equal to 1, the first phase is finished. In this way, the termination information needs not to be propagated along a tree rooted at the initiator. They send it directly to the initiator.

3.2 Data Structures

The following terms and notation are used in our algorithms:

R_c: an array maintained at each process. It is used to save the casually dependent information among processes. The array has n bits, representing n processes. If one process P_i depends on P_j (i.e., P_j sends a message to P_i), the bit j of P_i's dependent vector will be 1; otherwise, it is 0. Any time a site sends a computation message, it appends the R_c to the message. As a result, the receiver updates its local R_c based on dependent vector piggybacked with the computation message.

R_l: similar to R_c, but it saves the dependent information of the last checkpoint period.

R_t: similar to R_l. Besides setting all the bits corresponding to those in R_l to 1, it also sets all the bits corresponding to the processes on which it transitively depends to 1.

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

first: a boolean array of size n maintained by each process. The array is initialized to all zeroes each time a checkpoint at that process is taken. When a process P_i sends a computation message to process P_j, it sets $first[j]$ to 1.

csn: an array of n checkpoint sequence number (*csn*) at each process. Each checkpoint sequence number is represented by an integer. For process P_i, $csn[j]$

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 sequence number $csn[j]$. Note that, $csn[i]$ is the checkpoint sequence number of P_i.

trigger: a tuple (pid, inum) maintained by each process. *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 every system message and the first computation message that a process sends to every other process after taking local checkpoint.

propagate: a boolean to decide if there is a need to propagate the checkpoint *request*. It is initialized to 0, and set to 1 after a checkpoint is triggered by a computation message.

request: a system message to request the receiver to take a checkpoint.

reply: a system message sent to the initiator after the sender has finished its checkpointing.

The *csn* is initialized to an array of 1's at all processes. The trigger tuple at process P_i is initialized to $(i, 1)$. The weight at a process is initialized to 0. When a process P_i sends any computation message, it appends its $csn[i]$ and the R_c to the message.

3.3 Checkpointing Algorithm

Any site can initiate a checkpointing, and the algorithm does not require any process to suspend its underlying computation. When a process P_i initiates a checkpointing, it takes a local checkpoint, increments its checkpoint sequence number, sets weight to 1, and stores its own identifier and the new checkpoint sequence number in its trigger. Then it sends checkpoint *request* to all the processes, such that $R_c[j]$=1 and resumes its computation. Each request message carries the trigger of the initiator, the R_t and a portion of the weight of the initiator, whose weight is decreased by an equal amount.

When a process P_i receives a *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 and forwards the *request* to all the processes on which it depends, but P_j does not depend (P_j has sent *request* to the processes on which it depends). 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 the *request*, P_i does not need to take a checkpoint because it has already taken a checkpoint for this checkpointing initiation. A checkpoint may be triggered by a computation message. In this situation, the checkpoint *request* is not propagated. Therefore, when P_i receives a system checkpoint *request*, it needs to check whether it has propagated the checkpoint *request* or not. If propagate==0, P_i has propagated the *request*, so it only sends a reply to the initiator with the received weight. Otherwise, P_i reset propagate to 0 and forwards the *request* to all the processes on which it depends, but P_j does not depend. Then, P_i sends a *reply* to the initiator with the remaining weight.

When P_i receives a computation message from P_j, P_i compares the $P_j.csn[j]$ with its local $csn[j]$. If $P_j.csn$ is less than or equal to $P_i.csn[j]$, 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's checkpoint. Therefore, the message must have a trigger tuple. P_i first updates its $P_i.csn[j]$ to the $P_j.csn[j]$, then do the follows 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.pid* == *own_.trigger.pid* ∧ *msg_trigger.inum* > *own_trigger.inum*, it means that P_j has sent the message after taking a new checkpoint, while P_i has not taken a checkpoint for this checkpointing. Therefore, P_i takes a checkpoint before processing this message. P_i does not immediately propage this checkpoint *request*; however, it sets *propagate* to 1. When P_i receives the *request* later, from the initiator or other processes which forwards the initiator's *request*, it propages the *request*. Note that, P_i only takes a tentative checkpoint, which can not be made permanent until P_i receives a *request* from other processes.
- If *msg_trigger.pid* ≠ *own_trigger.pid*, P_i executes as follows: If P_i has not processed any message satisfying the condition *msg_trigger.pid* ≠ *own_trigger.pid* since its last local checkpoint, or if the initiator casually depends on P_i (R_t[i]==1), P_i takes a checkpoint, sets *propagate* to 1, and sets *own_trigger* to be *msg_trigger* before processing the message. Otherwise, if R_t[i]=0, and P_i has already processed a message from any process satisfying the condition *msg_trigger.pid* ≠ *own_trigger.pid* since its last local checkpoint, no new local checkpoint is needed.

In order to clearly present the algorithm, we assume that at any time, at most one checkpointing is in progress. Techniques to handle concurrent initiators of checkpointing by multiple processes can be found in [8, 11].
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;
    process_set: set of integers;
    Rc, Ri, Rt, first: bit array of size n;
```

Actions taken when P_i sends computation message to P_j:
if first[j]=0 *then* {
  first[j]=1;
  send(P_i, message, R_c, R_t, csn[i], own_trigger);}
else send(P_i, message, R_c, csn[i], NULL);

Actions for the initiator P_j:
own_trigger.pid=P_j; own_inum=csn[j]; clear R_t;
clear process_set;
take_cp(R_c, R_l, R_c, P_j, own_trigger);
prop_cp(R_l, R_t, P_i, msg_trigger, 1.0)
resume normal computation;

Other processes, P_i, on receiving checkpoint request from P_j:
receive(P_j, *request*, m.R_t, recv_csn, msg_trigger,
  recv_weight);
if msg_trigger==own_trigger *then* {
  if propagate==1 *then*
    prop_cp(R_l, R_t, P_i, msg_trigger, recv_weight)
  send(P_i, *reply*, recv_weight) to initiator; }
else {csn[j]=recv_csn;
  take_cp(R_c, R_l, m.R_t, P_i, msg_trigger);
  prop_cp(R_l, R_t, P_i, msg_trigger, recv_weight) }
resume normal computation;

Actions for process P_i, on receiving computation message from P_j:
receive((P_j, m, m.R_c, m.R_t, recv_csn, msg_trigger);
if recv_csn $\leq$ csn[j] *then* process the message and exit;
else {
  csn[j]=recv_csn;
  if msg_trigger.pid==own_trigger.pid *then*
   {*if* msg_trigger.inum==own_trigger.inum
   then process the message;
   else { take_cp(R_c, R_l, P_i, msg_trigger, m.R_t, P_i,
     msg_trigger);
    process the message; rfirst=1; propagate=1;}}
  else {*if* (rfirst ==0) OR (m.R_t[i]==1) *then*
    { take_cp(R_c, R_l, m.R_t,P_i, msg_trigger);
    process the message;
    rfirst=1; propagate=1; }
   else process the message; }}

take_cp(R_c, R_l, m.R_t, P_i, msg_trigger)
{take local checkpoint;
propagate =0; rfirst=0; increment(csn[i]);
own_trigger=msg_trigger; R_t=m.R_t;
R_l=R_c; reset R_c and first; }

prop_cp(R_l, m.R_t, P_i, msg_trigger, recv_weight)
{R_t = R_l OR m.R_t;
for all processes P_k, such that R_l[k]==1 *and* m.R_t[k] $\neq$ 1
  {weight=weight/2; send_weight=weight;
  send(P_i, *request*, R_t, csn[i], own_trigger, send_weight);}
propagate=0;
send(P_i, *reply*, recv_weight) to initiator; }

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) to P_k; }

Actions for other process P_j:
receive (make_permanent)
make the tentative checkpoint permanent.

3.4 An Example

The basic idea of the algorithm can be better understood by an example presented in Figure 1. In Figure 1, P_1 initiates a checkpointing by taking its own checkpoint and sends checkpoint *request* to P_2 and P_3, since P_1 depends on P_2 and P_3. When P_1's *request* reaches P_2, P_2 takes a checkpoint, then it sends message $m4$ to P_3. When $m4$ arrives at P_3, P_3 takes a checkpoint before processing the message because $m4$ is the first message received by P_3 such that *msg_trigger.pid* $\neq$ *own_trigger.pid*.

P_4 has not communicated with other processes before it takes a local checkpoint. Later, it sends a message $m5$ to P_3. Because P_4 has taken a checkpoint, its checkpoint sequence number is larger than P_3 expected. However, $m5$ is not the first computation message received by P_3 with a larger checkpoint sequence number than expected. Therefore, a checkpoint is not needed. Another reason for P_3 not taking a new checkpoint is that it may lead to an *avalanche effect*, in which processes in the system recursively ask others to take checkpoints. For example, if P_3 takes a checkpoint after it receives $m5$, then it requires P_2 to take another checkpoint. If P_2 has received messages from other processes after it sends $m4$, then those processes have to take checkpoints. This chain may never end.

When the *request* sent by P_1 arrives at P_3, P_3 does not need to take another checkpoint because the *msg_trigger* is equal to *own_trigger*. However, it needs to propagate this checkpoint *request* to P_5, because its current checkpoint is triggered by a computation message $m4$ and P_3 depends on P_5. In [10], P_3 first propagates the *request* when it receives $m4$, then propagates again when it receives the *request* from P_1. But our algorithm only propagates once. Note that the propagation is transitive, therefore our algorithm significantly reduces the message complexity.

Suppose P_4 takes another checkpoint after it receives $m6$, it sends a checkpoint *request* to P_3. If the channel is not FIFO, there is a possibility that $m7$ arrives at P_3 earlier than the *request*. In [10], P_3 does not take checkpoint until it receives the *request*, which results in inconsistency ($m7$ will be an orphan). In our algorithm, because P_3 causally depends on P_4, it takes a checkpoint before processing $m7$.

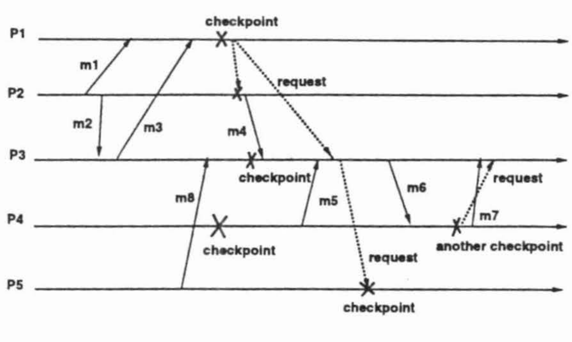

Figure 1: An example of checkpointing

4 Correctness Proof

Lemma 1 If process P_i takes a checkpoint and P_i depends on P_j, then P_j takes a checkpoint for the same checkpointing initiation.

Proof. If P_i is the initiator, to initiate a checkpointing, it sends *request* to all process on which it depends. If P_i is not the initiator and takes a checkpoint on receiving a *request* from P_k, then for the process P_j on which P_i depends, there are two possibilities:

Case 1: If m.R_t[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_t[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 directly depends receives at least one checkpoint *request*. There are two possibilities when P_j receives the first checkpoint *request*:

1: P_j has not taken its checkpoint when the first *request* for this initiation arrives: P_j takes its checkpoint on receiving the *request*.

2: P_j has taken a checkpoint for this checkpoint initiation when the first checkpoint *request* arrives: this *request* and all subsequent *request* messages for this initiation are ignored.

Hence, when a process takes a checkpoint, every process on which it is directly dependent takes a checkpoint. □

Applying the transitivity property of the dependence relation, we conclude that every process on which the initiator is dependent, directly or transitively, 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.

Theorem 1 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. In this case, P_i depends on P_j. From Lemma 1, 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 $P_j \Longrightarrow$ receive(m) at $P_i \Longrightarrow$ checkpoint taken at $P_i \Longrightarrow$ *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 sending of m is recorded at P_j. A contradiction.

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. Therefore, when m arrives at P_i, its checkpoint sequence number is greater than $P_i.csn[j]$. As a result, P_i takes its checkpoint before processing m. In other words, reception of m is not recorded in the checkpoint of P_i. A contradiction.

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, the sequence number of m is greater than $P_i.csn[j]$ and then we have a similar contradiction. □

The checkpointing algorithm terminates within a finite time. The proof is similar to [10] and [6].

5 Related Work

The first consistent checkpointing algorithm was presented in [1]. However, the algorithm assumes that all communications between processes are atomic, which is too restrict. The Koo and Toueg algorithm [8] relaxes this assumption, and only requires message exchange between processes that have dependency relationship, thus reducing the number of messages required. Later, Leu and Bhargava [9] presented another algorithm, which is resilient to multiple process failures, and does not assume that the channel is FIFO, which is necessary in [8]. These two algorithms have a common drawback in that they 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 [4] proposed an algorithm, which addresses both orphan message and lost inconsistencies.

In these consistent checkpointing algorithms, the processes are blocked when taking checkpoint and during rollback recovery. The blocking dramatically reduces the performance of the system [2, 5]. Kim and Park [7] attemped to solve this problem. Their basic idea is: A process takes a checkpoint when it knows that all processes on which it computationally depends have taken their checkpoints, and hence the process need not always wait for the decision made by the checkpoint initiator's decision. However, based on their algorithms, the processes in the system are still often need to be blocked.

In [16], when a process makes 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 every message sent, which may introduce some performance degradation, and require the system to be deterministic.

The Elnozahy-Johnson-Zwaenepoel algorithm [5] 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 of the processes in the computation. The algorithm proposed by Silva and Silva [12] uses the same idea as [5], except that the processes which did not communicate with others during a previous checkpoint period do not need to take a new checkpoint. Both algorithms [5, 12] 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, bottleneck, etc. If they are modified to permit other sites to initiate a checkpoint, which makes them truly distributed, the new algorithm suffers from another problem as follows: 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 every process can initiate checkpointing, the network would be flooded with control messages and processes might waste their time making unnecessary checkpoints.

The Prakash-Singhal [10] is also a non-block algorithm. However their algorithm is designed for mobile computing system and has FIFO assumption. Moreover, if a checkpoint is triggered by a computation message, their algorithm propagates the checkpoint *request* to all dependent processes twice in order to detect the termination of the checkpointing. The proposed algorithm is designed for general distributed system, and it does not have the FIFO assumption. Furthermore, our algorithm only propagates the checkpoint *request* once, which significantly reduces message overhead.

6 Conclusions

A distributed system is a collection of processes that communicate with each other by exchanging messages. Scalability in distributed system requires some effective approach to deal with failure. We present an efficient non-block scheme to address this problem. More specifically, a checkpoint sequence number vector is used to identify orphan messages, so processes involved in checkpointing need not to be blocked. Based on inter-process dependencies created since the last checkpointing, our scheme only forces a minimal set of processes to take their local checkpoints.

In this paper, we only presented a checkpointing algorithm. It is easy to see that a similar recovery algorithm can also be constructed. If our consistent checkpoint algorithm is used in recovery algorithms based on message logging, the algorithm does not require garbage collection of obsolete checkpoints, thus saving a lot of stable storage.

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

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] Ravi Prakash and Mukesh Singhal. "Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems". *Technical Report, the Ohio-state University.*

[11] 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.

[12] 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.

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

[14] 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, 1989.

[15] 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.

[16] 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.