Report Number: WUCS-93-20

1993-01-01

Improving the Speed of A Distributed Checkpointing Algorithm

Sachin Garg and Kenneth F. Wong

This paper shows how Koo and Toueg's distributed checkpointing algorithm can be modified so as to substantially reduce the average message volume. It attempts to avoid O(n{squared}) messages by using dependency knowledge to reduce the number of checkpoint request messages. Lemmas on consistency and termination are also included.

Improving the Speed of A Distributed Checkpointing Algorithm

Sachin Garg
Kenneth F. Wong

WUCS-93-20

April 1993

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis, MO 63130-4899

Improving the Speed of A Distibuted Checkpointing Algorithm

Sachin Garg
sachin@wuccrc.wustl.edu

Kenneth F. Wong
kenw@wuccrc.wustl.edu

Computer and Communications Research Center
Washington University
One Brookings Drive, Campus Box 1115
St. Louis, Missouri 63130

Presented at
Sixth International Conference on
Parallel and Distributed Systems
October 14-16, 1993
Louisville, Kentucky

Abstract

This paper shows how Koo and Toueg's distributed checkpointing algorithm can be modified so as to substantially reduce the average message volume. It attempts to avoid $O(n^2)$ messages by using dependency knowledge to reduce the number of checkpoint request messages. Lemmas on consistency and termination are also included.

Improving the Speed of A Distributed Checkpointing Algorithm*

Sachin Garg
sachin@wuccrc.wustl.edu

Kenneth F. Wong
kenw@wuccrc.wustl.edu

Computer and Communications Research Center
Washington University
One Brookings Drive, Campus Box 1115
St. Louis, Missouri 63130

Abstract

This paper shows how Koo and Toueg's distributed checkpointing algorithm can be modified so as to substantially reduce the average message volume. It attempts to avoid $O(n^2)$ messages by using dependency knowledge to reduce the number of checkpoint request messages. Lemmas on consistency and termination are also included.

Key Words: Checkpointing, distributed systems, fault-tolerance, performance.

1 Introduction

The possibility of tackling very large, computationally intensive problems by coupling large communities of distributed processors through a high-speed network is fast becoming a reality [7]. The computing sites may consist of computational resources from several vendors, and communication between sites may require message transmission over long distances (thousands of miles) through several intermediate hops. Clearly, computing in this environment is much more precarious and we can expect higher resource failure rates than in a standard multiprocessor. Thus, a fundamental problem which must be addressed in this environment is that of providing effective computational progress in the face of resource failures.

One approach to providing higher reliability is to have each site periodically checkpoint (save its state) onto stable storage. When a failure occurs, each site can resume computing after it restores its system state by reading the latest checkpoint from its checkpoint storage. However, this simple view of program resumption can only work effectively if the checkpoint is properly coordinated across the processor community.

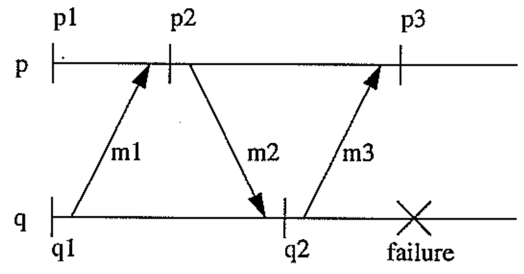

Figure 1: The Domino Effect

The most serious problem with uncoordinated checkpointing is that many generations of checkpoints may need to be stored in order to recover to a consistent system state. In the worst case, the *domino effect* occurs, forcing the system to roll back to the very beginning of program execution [9].

Figure 1 is a time diagram that illustrates this situation for two processes p and q. In the time diagram, each process is represented by a horizontal time line, and each message from process p to process q is represented by an arrow from process p's time line to process q's time line at the points when a message is sent and received. A small vertical line that intersects a single time line indicates a checkpoint. When a failure occurs at time point X, q rolls back to q_2, a state of process q. But since p_3 indicates the reception of message m_3, process p depends on process q, and p must roll back to p_2. This dependence ripples back to the initial checkpoints p_1 and q_1. The problem is that there is no checkpoint set $\{p_i, q_j\}$ in Figure 1 which is independent of each other.

One approach to avoiding this situation, is to coor-

*This research has been sponsored in part by funding from the NSF under Grant CCR-9021041.

dinate the checkpoints so as to maintain at least one set of consistent checkpoints at all times. These algorithms are based on the Chandy-Lamport algorithm [1], and they guarantee that each process need not rollback further than to the latest checkpoint [8, 9]. Most of the algorithms that use low amounts of storage use $O(n^2)$ messages, either intrinsically or as the interval between checkpoints gets large. We show how these algorithms can be modified to avoid $O(n^2)$ messages.

This paper is organized as follows. Section 2 introduces definitions and discusses related work. Section 3 discusses policies for maintaining consistent checkpoint sets. Section 4 describes our algorithm. Section 5 compares the performance of our algorithm with that of Koo and Toueg. Finally, Section 6 contains conclusions and discusses future work.

2 Definitions and Related Work

We assume that there are N processors, each with one process. The processes communicate with each other through virtually lossless, FIFO channels. Moreover, they exhibit fail-stop behavior; i.e. no byzantine failures can occur in the system. Failures are short-lived so that it makes sense to wait for failures to be repaired and the computation to resume from the latest set of checkpoints. Each process periodically takes a checkpoint of its state in coordination perhaps with other processes. The coordination guarantees that the latest checkpoints are consistent so that recovery involves only restoring the state of each process from these checkpoints.

A *checkpoint set* is a set of checkpoints, one per process. A single process checkpoint forms a local state, and a checkpoint set forms a global state of the system. A checkpoint set is said to be *consistent* if the global state does not contain a situation in which process p receives a message m from process q that has not yet been sent by q. In Figure 1, $\{p_3, q_2\}$ is inconsistent since m_3 has been received by p, but not sent by q. While $\{p_1, q_1\}$ is consistent.

An easy way to visually identify consistency and inconsistency is to use a time diagram like Figure 1. A *recovery line* is a line in the time diagram which intersects each time line exactly once at a checkpoint [8]. The region to the right of the recovery line represents the future, and the region to the left of the recovery line represents the past relative to the recovery line. A recovery line is consistent if no message crosses from right to left; that is, no message is transmitted from the future to the past. Messages that cross a recovery line from left to right are *cross-cut messages* and can be handled by the underlying message system which guarantees virtually lossless behavior. Upon a failure, all cross-cut messages are treated as *"lost"* messages [9].

All distributed checkpointing schemes resume computation from a consistent recovery line but take different approaches to finding a consistent one. These differences arise from different assumptions about:

1. the computation model,
2. the frequency of failures, and
3. the degree of process interaction between checkpoint intervals.

For example, database applications use a transaction-oriented computation model. This simplifies the solution to some degree. Much work has been done on checkpointing and rollback-recovery in transaction based systems [3, 4, 6]. Scientific computations running on a distributed set of processors use a computation model that can contain higher degrees of process interaction which imposes stringent requirements on the checkpointing protocol. Such an environment requires fast checkpointing and recovery to maintain good computational progress.

Checkpointing schemes fall into two broad categories: synchronous (coordinated) and asynchronous (uncoordinated). In the *asynchronous* approach, processes take checkpoints independent of each other and log messages (either incoming or outgoing) [2, 5, 12]. After a failure, processes affected by the failure must exchange dependency information to locate a consistent recovery line including the messages that must be replayed. The message logging scheme is used to prevent the occurrence of the domino effect.

In the *synchronous* approach the processes coordinate among themselves while saving their respective states to produce a globally consistent set of checkpoints [1, 8, 9, 11, 13]. After a failure, processes simply rollback to the latest set of checkpoints since those form a consistent recovery line. Some of these algorithms have also been concerned with checkpointing a minimum number of processes to maintain consistency [8, 9].

Both asynchronous and synchronous algorithms have their relative advantages and disadvantages. Asynchronous schemes are, in effect, application transparent (i.e., no coordination between checkpoint algorithm and application) [12]. The basic assumption is that failures will be rare, and therefore, the total checkpoint time will be low and the expensive recovery process will be used infrequently. This simplifies the checkpointing process and keeps the checkpointing overhead low, but at the expense of complicating the recovery process and consuming larger amounts of disk storage. But the high recovery time makes such schemes unsuitable for scientific computations.

In contrast, synchronous algorithms are relatively complex and expensive since they require coordination between processes to determine the recovery line. Storage is minimal since each process keeps at most two checkpoints [9]. The recovery algorithm is straightforward in the sense that it only involves rolling back to the latest checkpoint. Inherent in the algorithm is the assumption that processes can fail at any time, and thus a consistent state must reside on stable storage at all times. Our algorithm is a modification of a synchronous algorithm. But unlike most of these algorithms, it attempts to avoid $O(n^2)$ behavior.

3 Policies

This section examines the policies that must be enforced to maintain consistent recovery lines. They appear in various forms in the existing synchronous checkpoint algorithms. Four questions are posed and answered:

1. Which processes must be included in a new checkpoint set; i.e., take checkpoints?
2. When can a process that is participating in a new checkpoint continue normal processing?
3. If the checkpoint candidates are known, is the order of the checkpoints important?
4. How should failures during the checkpoint process be handled?

The policies below are stated assuming that that p and q are processes.

Policy #1 (checkpoint set): If p receives a message m from q and then takes a checkpoint, q must take a checkpoint if q sent m after its latest checkpoint.

This policy insures that the recovery line running through the two checkpoints will be consistent. In Figure 2, the checkpoint set $\{p_2, q_1\}$ is inconsistent. But policy #1 forces q to take checkpoint q_2. Process p depends on process q in the interval $< p_1, p_2 >$. Process p's *dependency graph* is the transitive closure of this *depends on* relation starting at process p's checkpoint and can be used to form a consistent recovery line. The set of processes in the dependency graph forms p's *dependency set.*

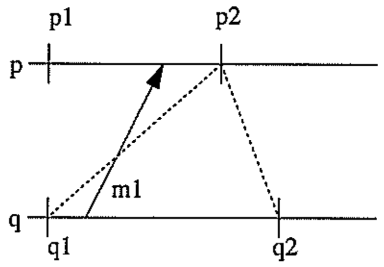

Figure 2: Policy #1

Policy #2 (resumption of normal processing): If p takes a checkpoint, p can not send application messages to any process q in its dependency set that has not taken a checkpoint since the last sending of q's message to p.

This policy guards against the situation where p sends an application message to q and q receives the message m before it takes a checkpoint, leading to an inconsistent recovery line. In a sense, this is just a restatement of policy #1 since from q's perspective, p is in its dependency set and therefore p must take a checkpoint after sending m.

But in its application, it implies that as soon as a process begins a new set of checkpoints, the processes participating in the checkpoints should not send out messages to members of the checkpoint set until all members have completed their checkpoints. If this policy were not followed, then policy #1 could result in a ping ponging effect in which a checkpoint could force the process sending a message after taking a checkpoint to immediately take another checkpoint.

Another implication of policy #2 is that the checkpoint algorithm should use a two-phase protocol in which the first phase takes tentative checkpoints, and the second phase commits (accepts) or undoes the checkpoints. Once the checkpoints have been committed or undone, application message sending can resume.

Policy #3 (checkpoint order): If p and q are in a checkpoint set, the order in which they take their checkpoints is unimportant for consistency so long as the other policies are followed.

Policies #1 and #2 guarantee that the recovery line will be consistent. However, it is natural to perform the checkpoints as the dependency graph is formed. Furthermore, if we perform the checkpoints while the dependency graph is formed, processes at the leaves of the dependency graph can commit their checkpoints immediately [8].

Policy #4 (failures during checkpointing): A new checkpoint set can not be committed until all of the processes in the checkpoint set have taken their checkpoints.

If a failure occurs before the processes have finished checkpointing, the partial new recovery line when combined with the existing recovery line will form an inconsistent recovery line.

The Koo and Toueg algorithm uses these policies in a two-phase-commit protocol. A process called the *initiator* (or coordinator) initiates the formation of a new recovery line. The essential elements of the algorithm for each process except the initiator are:

1. Phase I
   (a) Upon receipt of a request to take a tentative checkpoint (TCP), take a TCP if it hasn't already been done, determine the direct dependents, and request that they take a TCP.
   (b) Wait for a success or failure reply from the direct dependents.
   (c) Reply to the TCP requester with a "willing to checkpoint" message if a TCP has been done and the dependents have sent "willing to checkpoint" messages.
2. Phase II
   (a) Wait for a commit or undo request from a parent process and carry out the request.

In this algorithm, the dependency graph is implicitly formed by the first step in phase I. Eventually, a process executing the first step in phase I will either have no dependents or will request a dependent process to take a TCP that has already taken a TCP. In the latter case, the process will not take another TCP, but will immediately return a "willing to checkpoint" to the requester. The algorithm degenerates to $O(n^2)$ messages for large checkpoint intervals when there is high connectivity between processes because many processes will get TCP requests from multiple processes. Our algorithm tries to avoid this case.

4 Our Algorithm

Our algorithm attempts to eliminate the multiple TCP requests that a process receives in the Koo and Toueg algorithm in phase I and replaces the commit/undo notification messages with a single broadcast in phase II. During the creation of the dependency graph in phase I, partial knowledge of the dependency graph is piggy backed on top of each TCP (tentative checkpoint) message. For example, since the initiator i knows that it will send TCP requests to its dependents, it piggy backs this list of dependents on top of the TCP request that is sent to each of its dependents. The initiator's dependents now know which processes will be taking TCPs. If a dependent has dependents that are on this list, it need not send a TCP request to those dependents.

In phase II, the commit/undo requests are sent to the checkpoint participants using a single multicast instead of following the structure of the dependency graph. This avoids the inefficiencies that can result from pathological dependency graphs (e.g., ring).

The checkpoint algorithm for the initiator i and all other processes p is shown in Figure 3. The notation $< x, y, z >$ represents a message with the three components x, y, and z. The symbols $D^{(i)}$, $R_q^{(p)}$, and $S_q^{(p)}$ denote quantities that are defined below. Their appearance in the algorithm can be thought of as a macro call to the definitions below.

Let N be the number of processes, and let q be an arbitrary process. Each message has a numeric label. Each process q maintains two sets of message labels: $R_k^{(q)}, k \in \{1..N\}$ and $S_k^{(q)}, k \in \{1..N\}$. $R_k^{(q)}$ is the last message label that process q received from process k. $R_k^{(q)}$ is incremented each time q receives a message from k. $S_k^{(q)}$ is the last message label that process q sent to process k. $S_k^{(q)}$ is incremented each time q sends a message to k. $R_k^{(q)}$ and $S_k^{(q)}$ are reset to 0 after q takes a tentative checkpoint.

$D^{(p)}$ is the *direct dependency set* of process p. It is the set of processes that have sent at least one message to p since the last checkpoint; that is,

$$D^{(p)} = \{q | R_q^{(p)} > 0\} \qquad (1)$$

(lines i1-i2) After the initiator takes a tentative checkpoint, it sends messages to each process p in its direct dependency set requesting it to take a tentative checkpoint, and telling it the initiator's dependents and the number of messages that it received from process p. **(line i3)** It then waits for a reply from each dependent p about its willingness to checkpoint ($\omega^{(p)}$) and a set $W^{(p)}$ indicating the set of processes which p knows to have taken a tentative checkpoint. The initiator computes $\omega^{(i)}$, its willingness to commit to the checkpoint. **(lines i5)** By definition it is willing to commit if all of its direct dependents are willing to checkpoint. **(lines i6-i9)** It then broadcasts the decision to the processes in the set $W^{(i)}$. $W^{(i)}$ will contain the set of all processes that took a tentative

```
At the initiator i:
i1    Take a tentative checkpoint (TCP);
i2    send <TCP request,$D^{(i)} \cup \{i\}$,$R_p^{(i)}$> to $p \in D^{(i)}$;
i3    wait for reply $< \omega^{(p)}, W^{(p)} >$ from all $p \in D^{(i)}$;
i4    let $W^{(i)} = \bigcup_{p \in D^{(i)}} W^{(p)}$;
i5    let $\omega^{(i)} = true$ if $\omega^{(p)} = true$ for all $p \in D^{(i)}$;
i6    if $\omega^{(i)} = true$ then
i7        Broadcast <commit> to all $p \in W^{(i)}$;
i8    else
i9        Broadcast <undo> to all $p \in W^{(i)}$;
At each other process p:
Upon receipt of message <request TCP,$K^{(pp)*}$,$R_p^{(pp)}$>
from process pp:
p1    if $S_{pp}^{(p)} \geq R_p^{(pp)} \neq 0$ then
p2        let $R_q^{(p)} = S_q^{(p)} = 0, 1 \leq q \leq N$;
p3        Take a tentative checkpoint;
p4        let $T^{(p)} = D^{(p)} - K^{(pp)*}$;
p5        let $K^{(p)*} = K^{(pp)*} \cup D^{(p)}$;
p6        if $T^{(p)} \neq \phi$ then
p7            send <request TCP, $K^{(p)*}$, $R_q^{(p)}$> to all $q \in T^{(p)}$;
p8            wait for reply $< \omega^{(q)}, W^{(q)} >$ from all $q \in T^{(p)}$;
p9            let $W^{(p)} = \left(\bigcup_{q \in T^{(p)}} W^{(q)}\right) \cup \{p\}$
p10       else
p11           $W^{(p)} = \{p\}$;
p12   let $\omega^{(p)}$ = true if $\omega^{(q)}$ = true for all $q \in T^{(p)}$;
p13   send $< \omega^{(p)}, W^{(p)} >$ to pp;
Upon receipt of message <commit> or <undo>:
p14   if commit then
p15       Commit TCP;
p16   else
p17       Undo TCP;
```

Figure 3: Checkpoint Algorithm

checkpoint.

(lines p1-p3) The other processes respond to tentative checkpoint (TCP) requests. Each process p that is asked to participate in the checkpointing will get a TCP request indicating $K^{(pp)*}$ (which processes are known by the requesting process pp (parent process) to be getting TCP requests) and $R_p^{(pp)}$, the number of messages that pp received from p. If a process has already taken a tentative checkpoint but has not committed it, the message counters will be 0 resulting in process p replying that it is willing to checkpoint. Otherwise, process p will reset its message counters and take a tentative checkpoint. **(line p4)** After the tentative checkpoint, process p computes $T^{(p)}$, the subset of its direct dependency set that it thinks has not taken a tentative checkpoint. Process pp, process p's parent, has sent process p, $K^{(pp)*}$, the set of processes that pp knows will receive TCP requests. Process p does not need to send another TCP request to these processes. **(lines p6-p13)** Whether $T^{(p)}$ is empty or not, process p computes $W^{(p)}$, the set of processes that replies to process pp with this set and its decision on its willingness to commit the checkpoint.

(lines p14-p17) In phase II, each process p that replied with a willingness to commit the tentative checkpoint receives the decision and either commits or undoes the tentative checkpoint.

5 Performance Evaluation

We compare the performance of our algorithm with Koo's algorithm. In order to provide an estimate of the overhead involved in the two algorithms, we simulated both algorithms and measured the number of messages for a wide range of checkpoint intervals and computation configurations. Simple analytic results are also presented for one configuration which captures the dynamics of the two checkpoint algorithms.

The simulated system has N processors, each containing one process. In each simulation run, the message pattern generated by each process p is uniformly distributed over a set of processes F_p called the *fanout set* of process p. A new set F_p is chosen randomly from the N processes for each run. The *fanout* of process p is f_p, the cardinality of F_p. For large checkpoint intervals, processes with a large fanout will send messages to a large number of processes; while processes with a small fanout are restricted to sending messages to only a small number of processes. The fanout in combination with the size of the checkpoint interval controls the connectivity of the dependency graph.

Each process generates messages randomly (at exponentially distributed time intervals) with identical mean intermessage times. The receiver process is determined by picking uniformly among the members of the fanout set. Our results in this section are based on fixed checkpoint intervals. However, the results for Poisson checkpoint intervals are statistically no different.

The checkpoint coordinator is also chosen randomly from the N processes. The mean number of messages generated by each process in a checkpoint interval m is the ratio of the checkpoint interval and the mean intermessage time.

In each simulation run, a single random message graph is generated and multiple checkpoint intervals are simulated to obtain average values for the number of messages required to complete the checkpoint. Figure 4 shows the results of our simulations for $N = 16$ processes, fanouts of 4, 8, 12, and 15, and mean num-

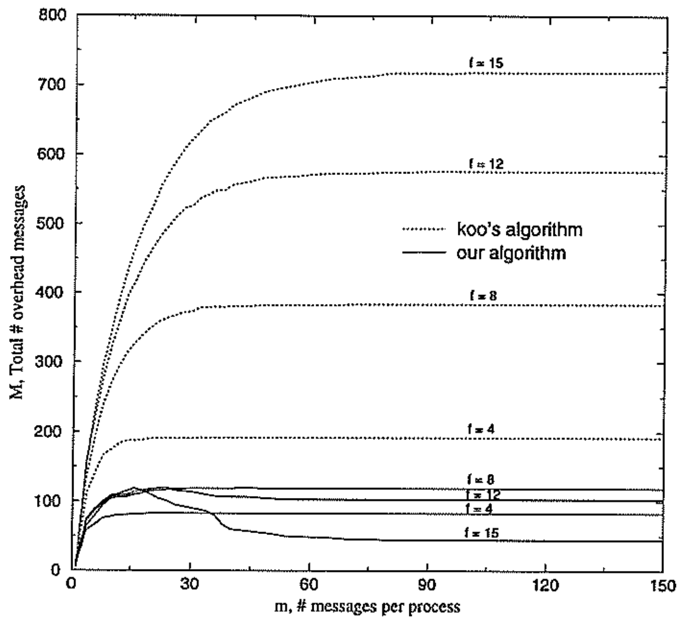

Figure 4: Comparison of Checkpoint Message Volumes

ber of messages sent by each process in a checkpoint interval varied from 1 to 150.

The message volume for Koo's algorithm behaves as expected when the fanout is $f = N - 1 = 15$ and the checkpoint interval is large. The number of messages should be $720 = 3(15) + 3(15)(15)$ since all processes (including the coordinator) will have received a message from all other processes with high probability. In Koo's algorithm a process sends out three types of checkpoint messages: 1) take a tentative checkpoint, 2) willing to checkpoint, and 3) commit or undo the tentative checkpoint. The coordinator will send or receive 15 sets of messages, one per process. These processes in turn will each send or receive 15 sets of messages.

In general, for $f = N - 1$ and for a checkpoint interval in which each process has sent out m messages, each process p will receive at least one message from process q with probability

$$P_1 = (1 - (1 - 1/(N - 1))^m) \tag{2}$$

If we assume the dependency graph is a two level tree, the mean checkpoint message volume M is

$$M = 3(N - 1)P_1 + 3((N - 1)P_1)^2 \tag{3}$$

since the fanout of each process is $N - 1$. The term $3(N-1)P_1$ is the message volume associated with the coordinator since each process has a fanout of $(N - 1)$ and sends at least one message to the coordinator with probability P_1. The term $3((N-1)P_1)^2$ is the message volume associated with the processes in the coordinator's dependency set and can be derived through a symmetry argument.

For a large checkpoint interval (and therefore large number of messages m), P_1 is near 1. It is clear that when the fanout is $N-1$ the message volume is $O(N^2)$ for large checkpoint intervals. But as the checkpoint interval decreases, P_1 will also decrease. As the fanout f decreases, so will the message volume because of the decreasing probability of each process sending a message to a particular process. For an arbitrary fanout f, the message volume looks like it is $O(f^2)$ for large checkpoint intervals. Large message volumes occur in Koo's algorithm because of processes receiving multiple TCP messages.

Our algorithm, on the other hand, attempts to avoid this situation by sending information to processes informing them of other processes that will be or have already been asked to take a tentative checkpoint. If the coordinator has received messages from a large fraction of the processes (a large fanout), it can tell the processes in its dependency set which processes will be asked to take a tentative checkpoint. These processes can be excluded from consideration by the processes in the coordinator's dependency set. These exclusions continue down the dependency graph.

The behavior of the overhead message volume seems appropriate for fanouts that are 8 or less since the message volume curves should look like scaled down versions of those for Koo's algorithm. But as the fanout increases past 8, the total message volume actually begins to decrease as the number of messages generated by each process increases.

For example, Figure 4 indicates that when the fanout is 15 (direct connectivity to all processes), the message volume begins to decrease when the number of messages generated by each process is around 15 or 20 messages. As the number of messages generated continues to increase, the limiting message volume approaches approximately 45. But this corresponds to the coordinator sending/receiving 3 messages to/from each of the other 15 processes.

A simple model explains this behavior for $f = 15$. The checkpoint message volume consists of approximately three components: 1) the messages associated with the coordinator in phase I, 2) the messages associated with processes in the coordinator's dependency set in phase I, and 3) the commit/undo messages in phase II:

$$\begin{aligned} M &= 2F_1 + 2F_1F_2 + (F_1 + F_2) && (4) \\ &= F_1(2F_2 + 3) + F_2 && (5) \end{aligned}$$

where $F_1 = (N-1)P_1$, $F_2 = ((N-1) - F_1)P_1$, and P_1 is defined above. The first term $2F_1$ is the number of messages associated with the coordinator in phase I and is identical to the estimate for Koo's algorithm. The second term $2F_1F_2$ is the number of messages associated with the next level in the dependency graph. It differs from the term for Koo's algorithm in that F_2 has replaced F_1. But F_2 is the number of processes that have not been notified by the coordinator to take a tentative checkpoint, but will be notified at the second level of the dependency graph. The expression for M reduces to $3(N-1)$ for a large checkpoint interval since $P1$ approaches 1 and F_2 approaches 0.

For fanouts less than $N-1$, information on the entire dependency set is not available to the coordinator leading to less of a reduction in the number of redundant messages. For a small enough fanout or a very small checkpoint interval, our algorithm degenerates to Koo's algorithm.

6 Conclusions and Future Research

We have shown how a synchronous, distributed checkpoint algorithm like Koo and Toueg's can be modified to substantially reduce the checkpoint message volume. In one configuration, the message volume is reduced to $O(n)$. The savings are obtained by avoiding most multiple tentative checkpoint requests to the same process. Simulations indicate this reduction and some simple analytic models agree with this conclusion. However, this message count savings is at the cost of some increased cost in message lengths because of the need for sending partial dependency information to the processes.

We are now studying the effect of these modifications on other performance measures. For example, we are interested in the effect of the computational progress of the application. A small number of messages that can not be transmitted with some degree of parallelism can be as bad a large volume of messages that can be transmitted in a parallel fashion.

Appendix (Proofs)

A proof that the our algorithm produces consistent checkpoints and that it will terminate is given below.

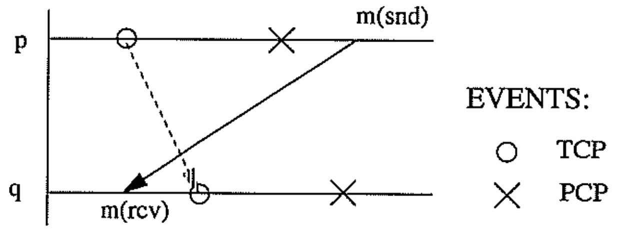

Figure 5: Inconsistent Checkpoint

A Consistency of Checkpoints

Lemma 1 : The nth set of checkpoints taken according to our algorithm is consistent.
Proof: By Induction on n
Let $H(n)$ be the hypothesis that the nth set is consistent. Let $H(0)$ be the base case.
Base Case $H(0)$*:* The starting checkpoint set is consistent since no computation has been performed by the application.
Inductive Step:
Proof (By contradiction): Assume that the set of checkpoints generated by the nth run of the algorithm is inconsistent. Then there exists a message m sent from process p after it made the tentative checkpoint permanent and received by process q before q took its tentative checkpoint. Although inconsistency can happen if the message from p is sent after it took the TCP but before committing, that situation is eliminated since the algorithm does not allow any application messages to be sent until the checkpoint is either committed or undone.

We can determine the order of events in the system as follows. The sending of the message m ($m(snd)$) happened in p after committing the checkpoint (PCP_p). The receiving of m ($m(rcv)$) happened in q before the tentative checkpoint TCP_q. Since the sending of a message occurs before reception of the message [10], $m(snd)$ happened before $m(rcv)$. This implies that PCP_p happened before TCP_q. This is a contradiction. If p and q take tentative checkpoints, the TCPs cannot be committed or undone by p and q until they are notified by the coordinator. Moreover, the coordinator can not send notification until it receives replies from all the processes that took a tentative checkpoint. Hence, p could not have committed its tentative checkpoint unless q had also taken a tentative checkpoint.

By the induction hypothesis, $H(n)$ holds for all n.

B No Lockout

The *"No Lockout"* property ensures that no deadlock can occur in the execution of the checkpointing algorithm. In our case, if the algorithm eventually terminates, the no lockout property is preserved.

Lemma 2 : The Algorithm preserves the *"No Lockout"* property.

Proof: We show that each process p taking part in the checkpointing eventually terminates its execution, and hence the algorithm eventually terminates (when all processes have terminated).

When a process p receives a TCP request, it takes a tentative checkpoint if it has not already done so. In which case, $S_q^{(p)} = 0, 1 \leq q \leq N$ (explicitly done by the algorithm) and hence can not accept any more requests for a tentative checkpoint. Clearly, if it had already taken a tentative checkpoint, it sends the reply and terminates. If it takes a tentative checkpoint, then two cases arise.

Case 1 ($T^{(p)} = \phi$): It has no dependents that won't get a TCP request from some other process. Process p does not wait for any replies. It sends its reply to process pp, the process that sent it the TCP request and terminates.

Case 2 ($T^{(k)} \neq \phi$): In this case, process p has to wait for replies from all processes $q \in T^{(p)}$. Each process q will be in one of the above cases. Case 2 is the only case that can cause a potential problem. But it is a problem only when a process attempts to take more than one tentative checkpoint. This case is prevented by resetting the message counters after taking a tentative checkpoint thereby forcing the process to return a willingness to commit without taking another tentative checkpoint.

Thus, each process eventually terminates execution of the checkpointing algorithm and hence preserves the "No Lockout" property.

References

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

[2] E.N. Elnozahy and W. Zwaenepoel, *"Manetho: transparent rollback-recovery with low overhead, limited rollback, and fast output commit"*, IEEE Trans. Comput., vol. 41, no. 5, May 1992.

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

[4] Jim Gray, et. al., *"The recovery manager of the System R database manager"*, ACM Comp. Surveys, vol. 13, no. 2, pp. 223-242, June 1981.

[5] D. B. Johnson and Willy Zwaenepoel, *"Sender based message logging"*, in Proc. 17th IEEE Symp. on Fault Tolerant Computing, pp. 14-19, June 1987.

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

[7] Alan H. Karp, Ken Miura, and Horst Simon, *"1992 Gordon Bell Prize Winners"*, IEEE Computer, vol. 26, no. 1, pp. 77-82, Jan. 1993.

[8] Junguk L. Kim and Taesoon Park, *"An efficient protocol for checkpointing recovery in distibuted systems"*, to appear in IEEE Trans. Parallel and Distr. Syst.

[9] Richard Koo and Sam Toueg, *"Checkpointing and rollback-recovery for distribute systems"*, IEEE Trans. Software Eng. vol. SE-13, no. 1, pp. 23-31, Jan. 1987.

[10] Leslie Lamport, *"Time, clocks, and ordering of events in a distributed system"*, Comm. ACM, vol. 21, no. 7, pp. 558-565, July 1978.

[11] Kai Li, J. F. Naughton, J. S. Plank, *"An efficient checkpointing method for multicomputers with wormhole routing"*, Intl. Jrnl. Parallel Proc., June 1992.

[12] R. E. Strom and S. Yemini, *"Optimistic recovery in distributed systems"*, ACM Trans. Comput. Syst., vol. 3, no. 3, pp. 204-226, Aug. 1985.

[13] Yuval Tamir and C. H. Sequin, *"Error recovery in multicomputers using global checkpoints"*, Intl. Conf. Parallel Proc., pp. 32-41, Aug. 21-24 1984.