Report Number: WUCS-93-37

1993

Analysis of an Improved Distributed Checkpointing Algorithm

Sachin Garg and Kenneth F. Wong

This paper presents the analysis of an improved distributed checkpointing algorithm. It shows that the message volume of Koo and Toueg's distributed checkpointing algorithm approaches 3fN for large checkpoint intervals where N is the number of processes and processes randomly send messages to f other processes. Thus, the average mesage volume is O(n2). We show how Koo and Toueg's algorithm can be modified so as to avoid this O9n2) overhead and derive an accurate estimate of the message volume. The overhead is reduced by using dependency knowledge to substantially reduce the average message volume.

... Read complete abstract on page 2.

Analysis of an Improved Distributed Checkpointing Algorithm

Sachin Garg and Kenneth F. Wong

Complete Abstract:

This paper presents the analysis of an improved distributed checkpointing algorithm. It shows that the message volume of Koo and Toueg's distributed checkpointing algorithm approaches 3fN for large checkpoint intervals where N is the number of processes and processes randomly send messages to f other processes. Thus, the average mesage volume is O(n2). We show how Koo and Toueg's algorithm can be modified so as to avoid this O9n2) overhead and derive an accurate estimate of the message volume. The overhead is reduced by using dependency knowledge to substantially reduce the average message volume.

Analysis of an Improved Distributed Checkpointing Algorithm

Sachin Garg
Kenneth F. Wong

WUCS-93-37

June 1993

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

Analysis of an Improved 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

Submitted to
Fifth IEEE Symposium on
Parallel and Distributed Processing

Abstract

This paper presents the analysis of an improved distributed checkpointing algorithm. It shows that the message volume of Koo and Toueg's distributed checkpointing algorithm approaches $3fN$ for large checkpoint intervals where N is the number of processes and processes randomly send messages to f other processes. Thus, the average message volume is $O(n^2)$. We show how Koo and Toueg's algorithm can be modified so as to avoid this $O(n^2)$ overhead and derive an accurate estimate of the message volume. The overhead is reduced by using dependency knowledge to substantially reduce the average message volume.

Analysis of an Improved Distributed Checkpointing Algorithm [1]

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

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 [?]. 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.

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 [?]. Figure ?? 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,

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

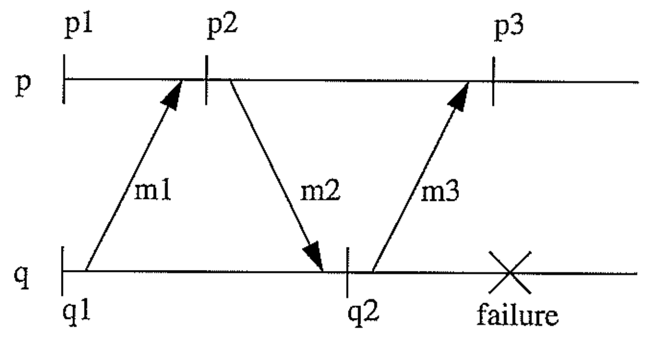

Figure 1: The Domino Effect

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 checkpoint 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 ?? which is independent of each other.

One approach to avoiding this situation, is to coordinate the checkpoints so as to maintain consistent checkpoints. These algorithms are based on the Chandy-Lamport algorithm [?], and they guarantee that each process need not rollback further than to the latest checkpoint [?, ?]. Most of the algorithms that use low amounts of storage use $O(N^2)$ messages as the interval between checkpoints gets large. The Koo and Toueg algorithm (KT algorithm) is such an algorithm. In an earlier paper, we proposed modifications to the KT algorithm and used a simulation model to show that our algorithm significantly reduced the average number of messages [?]. This paper extends that work by presenting analytic models of the average message volume used in the KT algorithm and our algorithm. Our model of the KT algorithm shows that as the checkpoint interval gets large, the average message volume is equal to $3fN$ where N is the number of processes and each process randomly sends messages to f other processes.

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 discusses the analytic models and compares the average message volumes of the KT algorithm and our algorithm. 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 to resume the computation from the latest set of checkpoints. Each process periodically takes a checkpoint of its state in coordination 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 ??, $\{p_3, q_2\}$ 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 ??. A *recovery line* is a line in the diagram which intersects each time line exactly once at a checkpoint [?]. 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. Upon a failure, all cross-cut messages are treated as *"lost"* messages [?].

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 [?, ?, ?]. 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 environments require fast checkpointing and recovery to maintain good computational progress.

Checkpointing schemes fall into two broad categories: synchronous and asynchronous. In the *asynchronous* approach, processes take checkpoints independent of each other and log messages (either incoming or outgoing) [?, ?, ?]. 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 [?, ?, ?, ?, ?]. 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 [?, ?].

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) [?]. 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. 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 substantially reduce the average message volume.

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 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 ??, 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 $\{p_1, p_2\}$. A graph of the *depends on* relation is a *dependency graph* in which an arc <p,q> in the graph indicates p depends on q. Either one can be used to form a consistent recovery line. The *direct*

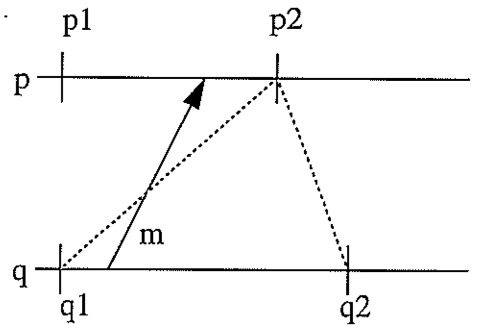

Figure 2: Policy #1

dependents of process p are processes that have sent at least one message to p since the latest checkpoint. They constitute the members of p's *dependency set.*

Policy #2 (resumption of normal processing): If p takes a checkpoint, p can not send application messages to any process q that is a dependent (direct or indirect) unless q has taken a checkpoint since the last sending of q's message to p or any of p's dependents (direct or indirect).

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 is used to take 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 [?].

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.

4 Two Distributed Checkpoint Algorithms

Both the KT algorithm and our algorithm use a two-phase-commit protocol that incorporates the policies described in the preceding section. A process called the *initiator* (or coordinator) initiates the formation of a new recovery line. The essential elements of the KT algorithm for each process p except the initiator are:

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

Step 1.a determines those elements <p,x> of the depends on relation such that process x is a direct dependent of process p. 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 requestor. 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 high message volumes by eliminating the multiple TCP requests that a process can receive in the KT algorithm in phase I and replaces the commit/undo notification messages with a multicast in phase II. During the creation of the depends on relation in phase I, partial knowledge of the depends on relation 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 the identity of some of the processes that 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 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 algorithms for the initiator i and for all other processes p are shown in Figure ??. 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 *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,

$$D^{(p)} = \{q | R_q^{(p)} > 0\} \tag{4.1}$$

(lines i1-i2) After the initiator takes a tentative checkpoint, it sends messages to each process p in its 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 that it is willing to checkpoint ($\omega^{(p)} = true$) and a set $W^{(p)}$ indicating the set of processes which p knows have taken a tentative checkpoint. The initiator computes $\omega^{(i)}$, its willingness to commit to the checkpoint. **(line i5)** By definition it is willing to commit if all of its dependents are willing to checkpoint. **(lines i6-i9)** It then broadcasts the decision to the processes in the set $W^{(i)}$. $W^{(i)}$ contains the set of all processes that took a tentative checkpoint.

The other processes respond to tentative checkpoint (TCP) requests. Each process p that is asked to checkpoint will get a TCP request indicating $K^{(pp)*}$ (the set of processes that the requesting process pp (parent process) knows should 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 commit. Otherwise, process p will reset its message counters and take a tentative checkpoint. **(line p5)** After the tentative checkpoint, it computes $T^{(p)}$, the subset of its dependency set that it thinks has not taken a tentative checkpoint. If pp is process p's parent, and $K^{(pp)*}$ is the set of processes that process pp knows will receive TCP requests, process p does not need to send another TCP request to the processes in $K^{(pp)*}$. **(lines p7-p14)** Whether $T^{(p)}$ is empty or not, process p computes $W^{(p)}$, the set of processes that have taken a tentative checkpoint, and replies to process pp with this set and its decision on its willingness to commit the checkpoint.

(lines p15-p18) 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.

```
At the initiator $i$:
i1   Take a tentative checkpoint (TCP);
i2   send <request TCP,$D^{(i)} \cup \{i\}$,$R_p^{(i)}$> 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   $\omega^{(p)} =$ true;
p2   if $S_{pp}^{(p)} \geq R_p^{(pp)} \neq 0$ then
p3       let $R_q^{(p)} = S_q^{(p)} = 0,\ 1 \leq q \leq N$;
p4       Take a tentative checkpoint;
p5       let $T^{(p)} = D^{(p)} - K^{(pp)*}$;
p6       let $K^{(p)*} = K^{(pp)*} \cup D^{(p)}$;
p7       if $T^{(p)} \neq \phi$ then
p8           send <request TCP, $K^{(p)*}$, $R^{(p)}$> to all $q \in T^{(p)}$;
p9           wait for reply $< \omega^{(q)}, W^{(p)} >$ from all $q \in T^{(p)}$;
p10          let $W^{(p)} = \left(\bigcup_{q \in T^{(p)}} W^{(q)}\right) \cup \{p\}$
p11      else
p12          $W^{(p)} = \{p\}$;
p13  let $\omega^{(p)} =$ true if $\omega^{(q)} =$ true for all $q \in T^{(p)}$;
p14  send $< \omega^{(p)}, W^{(p)} >$ to $pp$;
Upon receipt of message <commit> or <undo>:
p15  if commit then
p16      Commit TCP;
p17  else
p18      Undo TCP;
```

Figure 3: Checkpoint Algorithm

5 Average Message Volume Models

This section compares the performance of the two algorithms described earlier. Appendix I develops an analytic model of the average message volume associated with the two algorithms. This section presents the assumptions upon which these models are based. Then it describes the model of the KT algorithm and shows that for large checkpoint intervals, the average message volume is equal to $3fN$ where N is the number of processes and f is the potential fanout of each process. Finally, it treats the model of our algorithm in a similar fashion. Although there is no closed form expression for the limiting average message volume, the model shows a substantial reduction in the message volume.

The results from the analytic models were compared to those from a simulator [?]. In all cases, the results were within 5% of those from the simulator. Details of the validation can be found in [?].

5.1 Assumptions

We assume that the system has N processors, each containing one process. Each process p generates a message pattern that is uniformly distributed over a set of processes F_p called the *potential fanout set* of process p. The *potential fanout* of process p is f_p, the cardinality of F_p. For simplicity, we assume that all processes have the same potential fanout f. The set of potential fanout sets defines a potential communication pattern between the processes.

At the end of a checkpoint interval, a process will have sent messages to a set of processes called its *actual fanout set* with cardinality equal to the *actual fanout*. For small checkpoint intervals, the actual fanout set will probably be a small subset of the potential fanout set. But as the checkpoint interval is increased, the actual fanout set will eventually be equal to the potential fanout set. The potential fanouts of all of the processes in combination with the size of the checkpoint interval controls the connectivity of the dependency graph.

The checkpoint coordinator is chosen randomly from the N processes. The time between checkpoint initiation is a fixed time interval T. The mean number of messages generated by each process in a checkpoint interval is $m = T/t$, the ratio of the checkpoint interval T and the mean intermessage time t.

5.2 A Model of the KT Algorithm

Figure ?? shows the average message volumes for the KT algorithm for $N = 16$ processes, potential fanouts of f =4, 8, 12, and 15, and mean number of messages sent by each process in a checkpoint interval varied from m =1 to 100. The graph was constructed from the average message volume $M_{KT}(m, f)$, derived in Appendix I and described below.

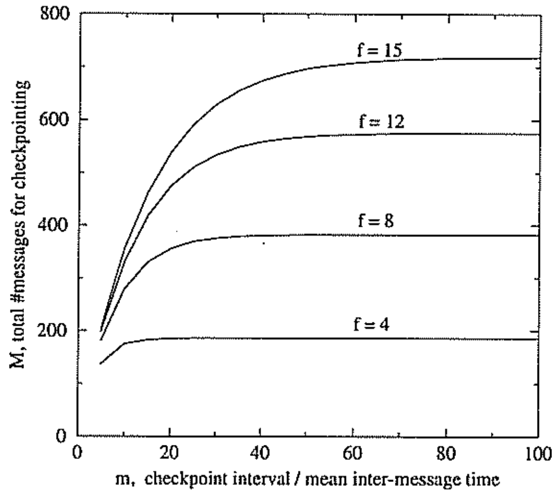

Figure 4: Message Volumes – KT Algorithm

Figure ?? clearly shows a sharp rise in the average message volume $M_{KT}(m, f)$ as the checkpoint interval lengthens. Appendix I shows that a simple approximation for $M_{KT}(m, f)$ is given by:

$$M_{KT}(m, f) = 3dN \tag{5.2}$$

where m is the average number of messages generated by each process, f is the potential fanout of each process, and d is the mean size of each dependency set:

$$d = f(1 - (1 - 1/f)^m) \tag{5.3}$$

For a large number of messages, d will be equal to f, the potential fanout of each process, and the average message volume becomes:

$$\lim_{m \to \infty} M_{KT}(m, f) = 3fN \tag{5.4}$$

Note that when the potential fanout is $f = N - 1$, the limiting average message volume is proportional to N^2 as expected.

5.3 A Model of Our Algorithm

Large message volumes occur in the KT algorithm because of processes receiving multiple "request 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 a large dependency set, 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 when they determine which of their dependents need a "request TCP" message. These exclusions continue down the dependency graph.

Figure ?? shows the average message volumes for both algorithms for $N = 16$ processes, potential fanouts of f =4, 8, 12, and 15, and mean number of messages sent by each process in a checkpoint interval varied from m =1 to 100. The data was generated from the average message volume expression $M(m, f)$ derived in Appendix I and described below.

The behavior of the average 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 the KT algorithm. That is, the sending of partial knowledge about dependencies should reduce the number of redundant TCP requests. 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. This occurs because for a sufficiently large fanout the savings from a reduction in redundant TCP requests is greater than the increase in messages due to a larger fanout.

For example, Figure ?? indicates that when the fanout is 15 (direct connectivity of the coordinator to all processes), the message volume begins to decrease when the number of messages generated by each process is around 15 or 20 messages. This is when the coordinator becomes connected to almost all other processes and most processes will eliminate the sending of redundant "request TCP" messages to its dependents. As the number of messages between checkpoints continues to increase, the limiting message volume approaches approximately 45. But this message volume corresponds to the coordinator sending/receiving 3 messages to/from each of the other 15 processes.

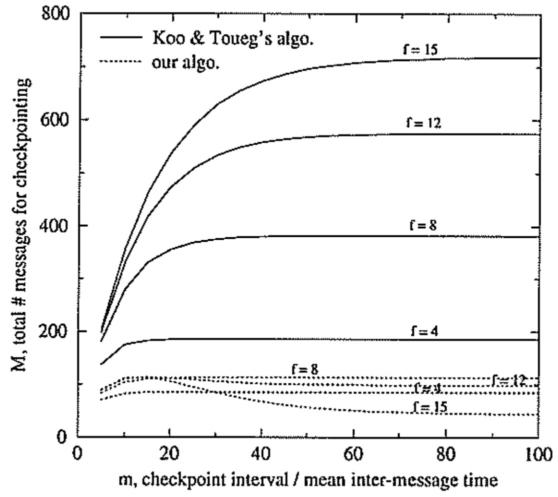

Figure 5: Comparison of Message Volumes

Appendix I shows that the message volume in our algorithm is approximately given by:

$$M(m, f) = 2(\sum_{j=1}^{k} \prod_{i=1}^{j} u'_i) + N \tag{5.5}$$

where k is the smallest integer such that $u'_k = 0$, and u'_k is the average number of processes at a distance k from the coordinator that will receive TCP requests:

$$u'_k = d\left(1 - \frac{\sum_{j=1}^{k-1} \prod_{i=1}^{j} u_i}{N-1}\right) \tag{5.6}$$

where u_k is the number of unique processes at a distance k from the coordinator and is given in the appendix by (??). The processes are unique in that they have not appeared in a dependency set that is within a distance k of the coordinator. The second term in equation ?? is approximately the number of multicast messages in phase II. The first term accounts for the two types of messages ("request TCP" and "willing to checkpoint") in phase I. Unfortunately, there is no simple expression for the limiting value of the average message volume.

6 Conclusions and Future Research

We have developed analytic models of the average message volume for our algorithm and Koo and Toueg's. The models have been verified elsewhere by comparing their results with a simulation model. The analytic model of the KT algorithm shows that for large checkpoint intervals, the average message volume is equal to $3fN$ and therefore exhibits N^2 behavior for large fanouts and large checkpoint intervals. Although a similar limiting result for our algorithm was not available in closed form, data using the model indicates that the average message volume can be reduced substantially by sending partial knowledge about the dependency information to dependents. The savings are obtained by avoiding the sending of most multiple tentative checkpoint requests to the same process. 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. Also, we are examining alternative algorithms.

7 Appendix (Derivation of Analytic Models)

We would like to develop an expression for the mean number of overhead messages $M(m, f)$. $M(m, f)$ is a function of two parameters: m, the mean number of messages per process between checkpoints, and f, the potential fanout of each process. A single mean $M(m, f)$ is an average computed over many checkpoint intervals of length m and many communication patterns in which processes have potential fanouts of f.

Fundamental to the understanding of the performance of both algorithms is the direct dependency graph. Recall that $D^{(p)}$ is the dependency set of process p and is the set of processes that have sent at least one message to p. If P is the set of all processes, then $R_D = \{< x, y > | x \in P, y \in D^{(x)}\}$ is the depends on relation such that $< x, y >$ indicates that process y has sent at least one message to x. The graph of R_D is the dependency graph G_D in which $< x, y > \in R_D$ implies that $< x, y >$ is a directed arc in G_D. In general, G_D is a graph with cycles.

The KT Algorithm

In the KT algorithm, the coordinator sends a "request TCP" message to each of the processes in its dependency set.. Each of these processes, in turn, sends a "request TCP" message to each of their dependents. This recursively continues until a request reaches either a process with an empty dependency set or a process that has already taken a TCP. In a successful checkpoint, each participant replies to the TCP request with a "willing to checkpoint" message and receives a "commit" message.

In order to derive the average message volume used in the KT algorithm, we imagine randomly picking a coordinator process and then following all dependencies in a breadth-first manner through the dependency graph. We count each arc traversal and try a different path when we encounter a process that has been previously visited. This procedure corresponds to a sequential generation of the "request TCP" messages in the KT algorithm.

Before deriving an expression for this counting procedure, we derive the average actual fanout f_{avg} of each process; that is, the number of processes that receives at least one message from an arbitrary process. From symmetry, f_{avg} is identical for all processes. Since each process randomly sends messages with probability $1/f$ to each process in its fanout set, each process has sent messages to f_{avg} processes after m messages:

$$f_{avg} = f(1 - (1 - 1/f)^m) \tag{7.7}$$

$(1-(1-1/f)^m)$ is the probability that at least one of the m messages was sent to a particular process. Since we assumed message targets were drawn independently and uniformly from f possibilities, the expression for f_{avg} follows.

But d, the mean size of each dependency set, is equal to the expression for f_{avg} since from symmetry, d for each process must be identical, and every arc in the dependency graph represents the transmission of at least one message in the opposite direction. So,

$$d = f(1 - (1 - 1/f)^m) \tag{7.8}$$

As expected, d varies from 0 to f as m varies from 0 to ∞.

Suppose we randomly pick process r to be the coordinator. Process r has dependents that are members of a dependency set that is a distance 1 from the coordinator. These processes, in turn, have dependency sets that are a distance 2 from the coordinator and so forth. By definition, the dependents of r are unique and different than r. But the dependency sets at distance k, k>1, have processes that can appear in dependency sets that are a distance k or less from the coordinator. The processes in a dependency set that are at a distance k from the coordinator are said to also be at distance k from the coordinator. Since processes can be in many dependency sets, they can be at many distances from the coordinator.

Let u_k be the number of unique processes in a dependency set that is a distance k from the coordinator; that is, u_k out of the d processes are not in any dependency set that is within a distance k of the coordinator. The remaining $d - u_k$ processes will have been visited by our dependency graph traversal algorithm and are those processes that will have already received a "request TCP" message. u_k satisfies the following balance equation:

$$u_k = d\left(1 - \frac{\sum_{j=1}^{k-1} \prod_{i=1}^{j} u_i + (\prod_{i=1}^{k-1} u_i - 1)u_k)}{N-1}\right) \tag{7.9}$$

The fraction is the probability that a process is in a dependency set within distance k of the coordinator and therefore has already received a "request TCP" message. The denominator of the fraction is $N-1$ because the members of the dependency set can be any of the N processes except the one process upon which the set is defined. The numerator of the fraction is the number of unique processes that appear in the dependency sets within distance k of the coordinator. The summation in the numerator is the number of unique processes within distance $k-1$ of the coordinator. The remaining term in the numerator is the number of unique processes at exactly distance k from the coordinator, but excluding the dependency set of interest. Since each dependency set has on average d processes, the relationship follows.

We can solve (??) for u_k. In order to account for the fact that u_k must eventually vanish, the expression for u_k becomes:

$$u_k = max\left(d\frac{1-\sum_{j=1}^{k-1}\prod_{i=1}^{j} u_i/(N-1)}{1+u_1(\prod_{i=1}^{k-1} u_i - 1)/(N-1)}, 0\right) \tag{7.10}$$

Note that $u_1 = d$ as required, and u_k decreases with increasing distance k from the coordinator.

We now have all of the expressions necessary to write down an expression for the average message volume. In the KT algorithm, a process will send on average d "request TCP" messages, one per dependent. Each of these dependents will reply with a "willing to checkpoint" message. In phase II, a process will end by sending d "commit/undo" messages, one per dependent. That is, there are $3d$ messages transmitted between a process and its dependents. Since the number of unique processes in the dependency graph is $1+\sum_{j=1}^{k}\prod_{i=1}^{j} u_i$ where k is the smallest integer such that u_k is non-zero, the average message volume in the KT algorithm is given by:

$$M_{KT}(m,f) = 3d(1+\sum_{j=1}^{k}\prod_{i=1}^{j} u_i) \tag{7.11}$$

But the parenthesized term for moderate to large values of m will be approximately equal to N. So, a simple approximation to (??) is

$$M_{KT}(m,f) = 3dN \tag{7.12}$$

Our Algorithm

In our algorithm, a process that is at a distance k from the coordinator will send less "request TCP" messages to its dependents than a process that is closer to the coordinator because it will receive accumulated knowledge about which processes will receive a "request TCP" message. Also, "commit/undo" messages are multicasted instead of passed along arcs of the dependency graph.

Let u'_k be the average number of processes at a distance k from the coordinator that will receive TCP requests. Then, the average message volume in our algorithm will be:

$$M(m,f) = 2(\sum_{j=1}^{k}\prod_{i=1}^{j} u'_i) + (\sum_{j=1}^{k}\prod_{i=1}^{j} u_i) \tag{7.13}$$

The first term accounts for the two types of messages in phase I, and the second term accounts for the multicast. As noted in the KT model, the second term can be approximated by N for most values of m, yielding:

$$M(m,f) = 2(\sum_{j=1}^{k}\prod_{i=1}^{j} u'_i) + N \tag{7.14}$$

where an expression for u'_i is derived below.

u'_k is greater than or equal to u_k because u'_k reflects only partial knowledge of which processes are within distance $k-1$ of the coordinator and no knowledge of which processes are at distance k of the coordinator. As an approximation, if we assume that a process at distance k has full knowlege of which processes are within a distance $k-1$ of the coordinator,

$$u'_k = d\left(1 - \frac{\sum_{j=1}^{k-1}\prod_{i=1}^{j} u_i}{N-1}\right) \tag{7.15}$$

The summation is the number of unique processes within distance $k-1$ of the coordinator. The fraction is the probability that one of the dependents of a process will be one of these unique processes. Since each process has d dependents, (??) follows.

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] P. L'Ecuyer and J. Malenfant, *"Computing optimal checkpointing strategies for rollback and recovery systems,"*, IEEE Trans. Comput., vol. 37, no. 4, pp. 154-163 , Apr. 1988.

[3] 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.

[4] 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.

[5] Sachin Garg and Kenneth F. Wong, *"Improving the Speed of a Distributed Checkpointing Algorithm"*, Proc. Sixth Intl. Conf. on Parallel and Distributed Computing Systems, Lexington, Kentucky, October 14-16, 1993.

[6] Sachin Garg and Kenneth F. Wong, *"Validation of Analytic Models of Two Distributed Checkpointing Algorithms"*, Tech. Rep. WUCCRC 93-11, The Computer and Communications Research Center, Washington University, St. Louis, Missouri, April 1993.

[7] 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.

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

[9] 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.

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

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

[12] 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.

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

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

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

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

[17] S. Toueg and O. Babaoglu, *"On the optimum checkpoint selection problem"*, SIAM J. Computing, vol. 13, no. 3, pp. 630-649, Aug. 1984.