Consistent Logical Checkpointing

Nitin H. Vaidya
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
Phone: 409-845-0512
Fax: 409-847-8578
E-mail: vaidya@cs.tamu.edu

Technical Report 94-051
July 1994[*]

Abstract

A "consistent checkpointing" algorithm saves a consistent view of the distributed system state on stable storage. The loss of computation upon a failure can be bounded by taking consistent checkpoints with adequate frequency.

The traditional consistent checkpointing algorithms require the different processes to save their state at about the same time. This causes contention for the stable storage, potentially resulting in large overheads. *Staggering* the checkpoints taken by various processes can reduce the overhead. Some techniques for *staggering* the checkpoints have been proposed previously [9], however, these techniques result in "limited staggering" in that not all processes' checkpoints can be staggered. Ideally, one would like to stagger the checkpoints *arbitrarily.*

This report presents a simple approach to *arbitrarily stagger* the checkpoints. Our approach requires that the processes take consistent *logical* checkpoints, as compared to consistent *physical* checkpoints enforced by existing algorithms. This report discusses the proposed approach and the implementation issues. The proposed approach was discussed briefly in [11].

[*]Section 5 (Related Work) revised October 1994.

1 Introduction

Applications executed on a large number of processors, either in a distributed environment, or on multicomputers such as nCube, are subject to processor failures. Unless some recovery techniques are utilized, a processor failure will require a restart of the application, resulting in significant loss of performance.

Consistent checkpointing is a commonly used technique to prevent complete loss of computation upon a failure [1, 3, 7, 9, 12]. A "consistent checkpointing" algorithm saves a consistent view of the distributed system state on a stable storage. The loss of computation upon a failure is bounded by taking consistent checkpoints with adequate frequency.

The traditional consistent checkpointing algorithms require the different processes to save their state at about the same time. This causes contention for the stable storage, potentially resulting in significant performance degradation [9]. *Staggering* the checkpoints taken by various processes can reduce the overhead of consistent checkpointing. Some techniques for *staggering* the checkpoints have been previously proposed [9], however, these techniques result in "limited staggering" in that not all processes' checkpoints can be staggered. Ideally, one would like to stagger the checkpoints *arbitrarily.* We assume that a processor does not have enough memory to make an "in-memory" copy of entire process state.

This report presents a simple approach to "completely stagger" the checkpoints. Our approach requires that the processes take consistent *logical* checkpoints, as compared to consistent *physical* checkpoints enforced by existing algorithms. This report discusses the proposed approach and the implementation issues. (These were discussed briefly in [11].)

The report is organized as follows. Section 2 discusses the notion of a *logical checkpoint.* Section 3 presents a consistent checkpointing algorithm proposed by Chandy and Lamport [1]. Section 4 presents the basic principle behind the proposed approach; implementation issues are discussed in Section 6. Our approach is closely related to [1, 5, 9, 13], as discussed in Section 5. Section 7 concludes the report.

2 A *Logical* Checkpoint

A process is said to be *deterministic* if its state depends only on its initial state and the messages delivered to it [10]. A *deterministic* process can take two types of checkpoints: a *physical* checkpoint or a *logical* checkpoint. A process is said to have taken a *physical* checkpoint at some time t_1, if the process state at time t_1 is saved on the stable storage. A process is said to have taken a *logical* checkpoint at time t_1, if *enough* information is saved on the stable storage to allow the process state at time t_1 to be recovered.

To the best of our knowledge, the term *logical* checkpoint was first introduced by Wang et al. [13, 14], who also presented one approach for taking a logical checkpoint. Now we present three approaches for taking a logical checkpoint at time t_1. Although the three approaches are equivalent, each approach may be more attractive for some applications than the other approaches. Not all approaches will be feasible on all systems.

- One approach for establishing a logical checkpoint at time t_1 is to take a *physical* checkpoint at some time $t_0 \leq t_1$ and log (on stable storage) all messages delivered to the process between time t_0 and t_1. (For each message, the message log contains the *receive sequence number* for the message as well as the entire message.) This approach is essentially identical to that presented by Wang et al. [13].

  Figure 1 presents an example wherein process P takes a physical checkpoint at time t_0. Messages M1, M2 and M3 are delivered to process P by time t_1. To establish a logical checkpoint of process P at time t_1, messages M1, M2 and M3 are logged on the stable storage. As process P is deterministic, the state of process P can be recovered using the information on the stable storage (i.e., physical checkpoint at t_0 and messages M1, M2 and M3).

  We summarize this approach as:

  physical checkpoint + *message log* = *logical checkpoint*

- The essential purpose behind saving the messages above is to be able to recreate the the state at time t_1, or to be able to "re-perform" the incremental changes made in

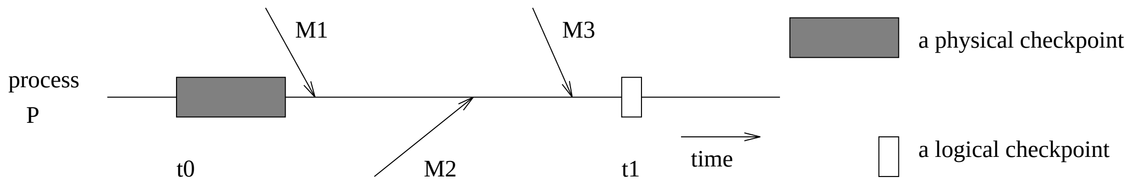

Figure 1: Physical checkpoint + message log = logical checkpoint

process state by each of these messages. This may be achieved simply by taking a *physical* checkpoint at time t_0 and taking an *incremental* checkpoint at time t_1. The incremental checkpoint is taken by logging[1] the changes made to process state between time t_0 and t_1. [2] We summarize this approach as:

physical checkpoint + *incremental checkpoint* = *logical checkpoint*

- The above two approaches take a physical checkpoint *prior* to the desired logical checkpoint, *followed* by logging of additional information (either messages or incremental state change).

  The third approach is the *converse* of the above two approaches. Here, the *physical* checkpoint is taken at a time t_2, where $t_2 > t_1$. In addition, enough information is saved to *un-do* the effect of messages received between time t_1 and t_2. For each relevant message (whose effect must be undone), an *anti-message* is saved on the stable storage. The notion of an *anti-message* here is similar to that used in time warp mechanism [4] (although the implementations could be very different) or that of UNDO records [2] in database systems. Anti-message M* corresponding to a message M can be used to undo the state change caused by message M.

  Figure 2 illustrate this approach. A *logical* checkpoint of process P is to be established at time t_1. Process P delivers messages M4 and M5 between time t_1 and t_2. A *physical* checkpoint is taken at time t_2, and *anti-messages* corresponding to messages M4 and M5 are logged on the stable storage. The anti-messages are named M4* and M5*,

[1]The term *logging* is used to mean "saving on the stable storage".

[2]It is possible to take the so-called *physical* checkpoint also as the incremental change in the process state since the last physical checkpoint of the process was taken. As the interval between two consecutive physical checkpoints is likely to be large, this incremental change is likely to be large. On the other hand, the interval $t_0 - t_1$ will be relatively small, potentially reducing the state affected during that interval.

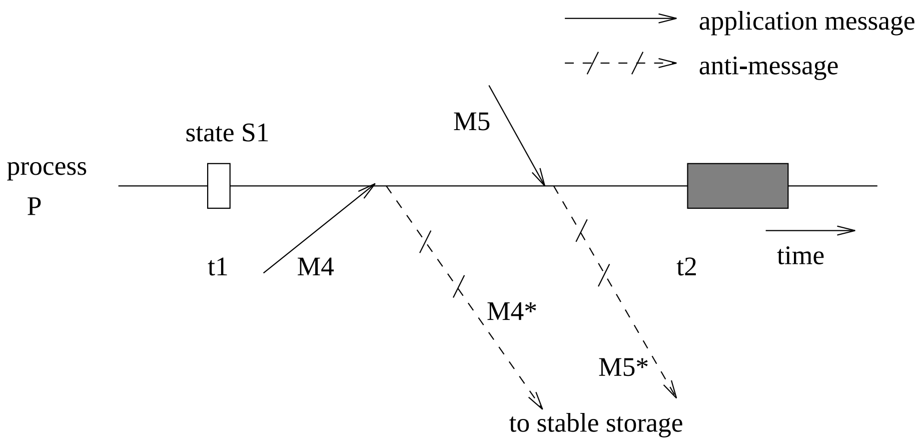

Figure 2: Anti-message log + physical checkpoint = logical checkpoint

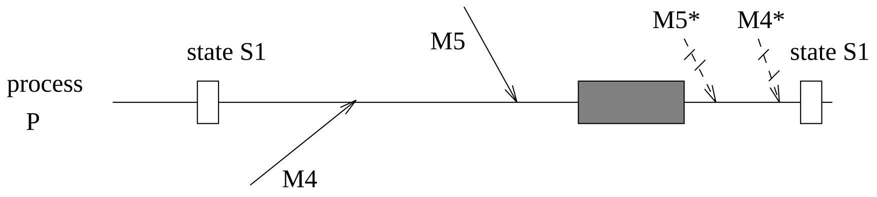

Figure 3: Recovering a logical checkpoint using anti-messages

respectively.

To recover the state, say S1, of process P at time t_1, the process is initialized to the physical checkpoint taken at time t_2 and then anti-messages M5* and M4* are sent to the process. The order in which the anti-messages are delivered is reverse the order in which the messages were delivered. As shown in Figure 3, the final state of process P is identical to the state (or logical checkpoint) at time t_1.

We summarize this approach as:

anti-message log + *physical checkpoint* = *logical checkpoint*

An important issue is that of forming the "anti-messages". The anti-messages can possibly be formed by the application itself, or they may consist of a copy of the (old) process state *modified* by the message. We have, as yet, not experimented with anti-messages. Therefore, practicality of this idea is open to debate.

Note that a physical checkpoint is trivially a logical checkpoint, however, the converse is not true.

3 Chandy-Lamport Algorithm [1]

Chandy and Lamport [1] presented an algorithm for taking a consistent checkpoint of a distributed system. Although the proposed approach can potentially be used with any consistent checkpointing algorithm, for brevity, we limit our discussion to the Chandy-Lamport algorithm.

Assume that the processes communicate with each other using unidirectional communication *channels*; a bidirectional channel can be modeled as two unidirectional channels. The communication graph is assumed to be strongly connected. The algorithm presented next is essentially identical to Chandy-Lamport [1] and assumes that a certain process is designated as the *coordinator*. This algorithm is also presented in [9].

Algorithm: The coordinator process, say P, initiates the consistent checkpointing algorithm by sending *marker* messages on each channel, incident on, and directed away from P and immediately takes a checkpoint.

A process, say Q, on receiving a *marker* message along a channel c takes the following steps:

```
if Q has not taken a checkpoint then
begin
    Q sends a marker on each channel, incident on, and directed away from Q
    Q takes a checkpoint
    Q records the state of channel c as being empty
end
else Q records the state of channel c as the sequence of messages received along c
    after Q had taken a checkpoint and before Q received the marker along c.
```

4 Consistent Logical Checkpointing

The proposed algorithm can be summarized as follows:

staggered physical checkpoints + *consistent message logging* = *consistent logical checkpoints*

The basic idea is to coordinate *logical* checkpoints rather than *physical* checkpoints. In this section, we assume that the first approach, described in Section 2, for taking logical checkpoints is being used. Thus, a logical checkpoint is taken by logging all the messages delivered to a process since its most recent physical checkpoint.

Algorithm

1. *Physical checkpointing phase:* A checkpoint coordinator sends a *take_checkpoint* message to each process. Each process, sometime after receiving this message, takes a *physical* checkpoint and sends an acknowledgement to the coordinator. The checkpoints taken by the processes are *staggered* by allowing an appropriate number[3] of processes to take checkpoints at any time (this can be done using any l-mutual exclusion algorithm). The checkpoints taken by the processes need not be consistent. The processes take checkpoints as soon as possible after receiving the *take_checkpoint* message (subject to the staggering constraint).

   After a process takes the checkpoint, it can continue execution. A process stores each message delivered to it, after its physical checkpoint, into a volatile buffer. Overflows in this buffer are spilled into the stable storage. (Alternatively, the process may asynchronously log these messages to the stable storage even if the buffer is not full.)

   Number of messages required in this phase can be reduced, as discussed in Section 6.

2. *Consistent message logging phase:* When the coordinator receives acknowledgement messages from all the processes indicating that they have taken a physical checkpoint, the coordinator initiates the *consistent message logging* phase, essentially, by initiating the Chandy-Lamport algorithm [1]. The only difference is that when the original algorithm requires a process to take a "checkpoint", the process takes a *logical* checkpoint

[3]For instance, the number of processes allowed to checkpoint simultaneously may be equal to the number of disks.

by logging the messages delivered since the physical checkpoint taken in the previous phase. As required by the Chandy-Lamport algorithm, messages representing the channel states at the time the consistent checkpoint is taken are also logged.

When the Chandy-Lamport algorithm is complete, the processes can discontinue logging received messages (specifically, when a process has logged the state of a channel c, it need not log further messages received on channel c).

This algorithm establishes a consistent recovery line consisting of one *logical* checkpoint per process.

The above algorithm reduces the contention for the stable storage by completely staggering the physical checkpoints. However, contention is now introduced in the second phase of the algorithm when the processes coordinate message logging. This contention can be reduced by using the limited staggering techniques proposed in [9]. Our scheme will perform well if message volume is relatively small compared to checkpoint sizes. A few variations to the above algorithm are possible, as discussed in Section 6

Figure 4 illustrates the algorithm assuming that the system consists of three processes, and that at most one process can write to the stable storage at any time. Process P acts as the coordinator and initiates the checkpointing phase by sending the *take_checkpoint* messages. Processes P, Q and R take staggered checkpoints. When process P receives acknowledgements from processes Q and R, it initiates the *consistent message logging* phase consisting essentially of an execution of the Chandy-Lamport algorithm. Process P sends marker messages to Q and R and then takes a *logical* checkpoint by logging messages M0 and M2 to the stable storage. When process Q receives the marker message from process P, it sends markers to P and R and then takes a logical checkpoint by logging message M1 to the stable storage. Similarly, process R takes a logical checkpoint by logging message M3 to the stable storage. Messages M4 and M5 are logged by the Chandy-Lamport algorithm as the state of the channels at the time the consistent (logical) checkpoints were taken.

Recovery: After a failure, each process rolls back to its recent physical checkpoint and re-executes (using the logged messages) to restore the process state to the logical checkpoint that is a part of the most recent consistent recovery line.

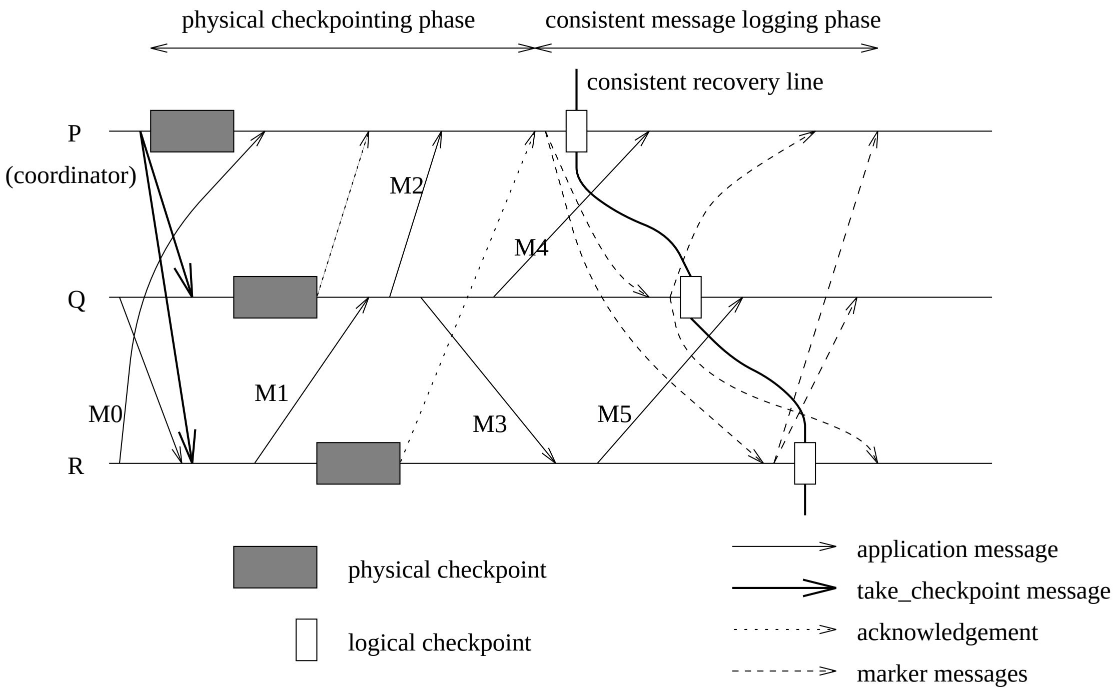

Figure 4: An example

5 Relation to Existing Work

The algorithm presented above is closely related to [1, 5, 8, 9, 13]. Our algorithm is designed to bound the rollback distance, similar to the traditional coordinated checkpointing algorithms. It may be noted that, after a failure, a process rolls back to a *physical* checkpoint and then executes to restore a logical checkpoint. Thus, the overhead of recovery (or rollback distance) is determined by when *physical* checkpoints are taken.

Wang et al. [13, 14] introduced the notion of a logical checkpoint. They determine a recovery line consisting of consistent logical checkpoints, *after* a failure occurs. This recovery line is used to recover from the failure. Their goal is to determine the "latest" consistent recovery line using the information saved on the stable storage. During failure-free operation each process is allowed to independently take checkpoints and log messages. On the other hand, our scheme *coordinates* logical checkpoints *before* a failure occurs. These logical checkpoints are used to recover from a *future* failure. One consequence of this is that we do not need to log all messages, only those message are logged which make the logical checkpoints consistent.

Plank [9] presents two coordinated checkpointing algorithms (one similar to Chandy-Lamport [1]) that attempt to stagger the checkpoints. However, it is possible that some checkpoints taken by these algorithms cannot be staggered. The degree of staggering is affected by the timing of application message delivery. In contrast, our algorithm allows arbitrary staggering of the *physical* checkpoints.

Long et al. [8] discuss an *evolutionary* checkpointing approach that is similar to consistent logical checkpointing. The fundamental difference between the two approaches is that our approach staggers the physical checkpoints, while the scheme in [8] does not allow staggering. By enforcing staggering, our approach is expected to perform much better than [8]. Long et al. also assume *synchronized communication*, no such assumption is made in the proposed approach.

Johnson [5] presents an algorithm that forces the processes to log message on the stable storage or to take a physical checkpoint. The goal of his algorithm is to make the state of a *single* process committable (primarily, to allow it to commit an output). Also, his algorithm does not control the time at which each process takes the checkpoint. Our algorithm is designed to bound the rollback distance (and not for output commits) and it makes recent states of *all* processes committable. The same result can be achieved by executing Johnson's algorithm simultaneously for *all* processes. The implementation will not bound the rollback distance, however, as the timing of the physical checkpoints is not controlled by his algorithm. Additionally, Johnson's algorithm can result in all messages being logged (as processes may choose to log messages asynchronously), our algorithm logs messages only until the *consistent message logging* phase is completed.

6 Implementation Issues

Many variations of the algorithm presented earlier are possible. Utility of these variations depends on the nature of the application and the execution environment. In the following, we discuss some of the implementation issues.

Checkpointing versus message logging:

- If a process receives too many messages after taking the physical checkpoint in the first phase of the algorithm, then it may decide to take a physical checkpoint in the second phase (rather than logging messages). This makes the physical checkpoint taken by the process in the first phase redundant. However, this modification may reduce the overhead when checkpoint size is smaller than what the message log would be.

- A process may decide to not take the physical checkpoint in the first phase, if it a priori knows that its message log will be large. In this case, the process would take a physical checkpoint in the second phase.[4]

- The coordinator may initiate the *consistent message logging* phase even before all processes have taken the physical checkpoint. In this case, consider a process Q that receives a marker message before Q has taken the physical checkpoint (in the first phase). Then, process Q can take a physical checkpoint in the second phase rather than logging messages to establish a logical checkpoint (essentially, process P can pretend that it decided to not take a physical checkpoint in the first phase).[5]

  If the coordinator initiates consistent message logging phase early, then it can cause increase in stable storage contention as some checkpoints may not be staggered anymore.

Staggering in the consistent message logging phase: During the *consistent message logging* phase, the processes will save their message logs to the stable storage at about the same time. To minimize stable storage contention during this phase, the technique presented by Plank [9] can be used. Essentially, Plank's method (as applied to our algorithm) would suggest that the processes should delay the *sending* of a *marker* message until after the process has itself established a logical checkpoint. (As pointed out earlier, this results in a limited amount of staggering.) We suggest another approach for limited staggering. This

[4]Johnson [5] suggested a scheme where each process uses a similar heuristic to decide whether to log messages or not.

[5]Recollect that a *physical* checkpoint is also trivially a *logical* checkpoint. So the process here is actually taking a logical checkpoint, but not by logging messages.

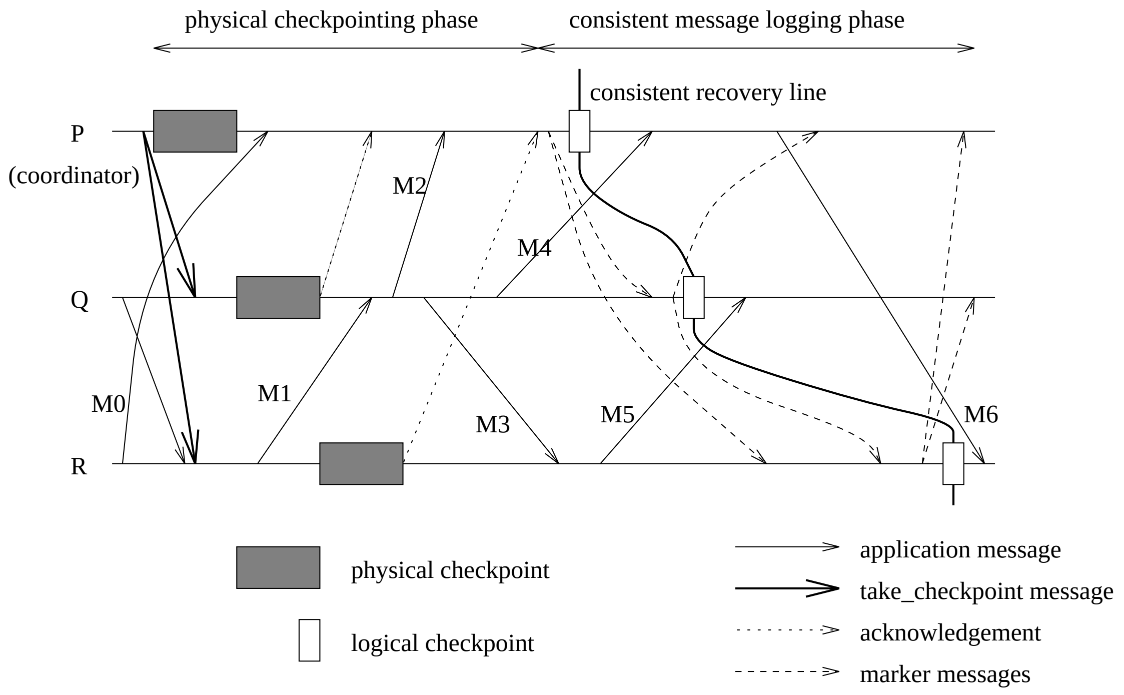

Figure 5: Delaying the delivery of the marker messages

approach delays the *receipt* of a marker message, unlike the delayed sending of the marker messages suggested by Plank.

The Chandy-Lamport algorithm requires that a process take a checkpoint (if it has not already done so) when it receives a marker message on any channel. It is possible to introduce some staggering by delaying the delivery of the marker message. Delivery of a marker message can be delayed until a message subsequent to the marker message (on the same channel) needs to be delivered. Figure 5 illustrates this. As shown in this figure, process R can delay taking logical checkpoint even though it has received markers from both P and Q. The logical checkpoint of process P can be delayed until message M6 from process P needs to be delivered to process R. As the marker precedes message M6, delivery of the marker cannot be delayed any further.

Another related point is that, in our algorithm, messages are logged either because they are needed to establish a logical checkpoint or because they represent the channel state at the time the consistent logical checkpoints are taken. It is possible to log these two types of messages together (rather than separately, as implied by the algorithm description in Section 4).

Number of messages required in the physical checkpointing phase: The description of the algorithm in Section 4 implies that, for N processes, $2(N-1)$ messages (*take_checkpoint* and acknowledgement) are required in this phase, possibly in addition to the messages required to ensure staggered checkpointing (mutual exclusion). However, the number of messages can be reduced significantly as illustrated with an example in Figure 6. Assume that there are three processes and at most one can write to the stable storage at any time. The processes can form a "cycle", position in the cycle determining when a process takes the physical checkpoint. As shown in the figure, process P takes a physical checkpoint, and sends a *take_checkpoint* message to process Q (but to no other process). On receiving this message, process Q takes a physical checkpoint, and then sends a *take_checkpoint* message to R. On receiving the *take_checkpoint* message, process R takes a physical checkpoint, and then sends an *acknowledgement* message to process P. Receipt of this acknowledgment suffices to guarantee that all processes have taken a physical checkpoint. Also, the cyclic arrangement ensures that only one process takes a physical checkpoint at any time. This approach can be easily modified when multiple processes can write to the stable storage simultaneously (e.g., form multiple cycles).

Approach for taking a logical checkpoint: The discussion so far assumed that a logical checkpoint is taken by taking a physical checkpoint and logging subsequently received messages. It is easy to see that the proposed algorithm can be modified to allow a process to use any of the three approaches presented earlier (in Section 2) for establishing a logical checkpoint. In fact, different processes may simultaneously use different approaches for establishing a logical checkpoint. Details of the modified algorithm will be presented in a future revision of this report.

Synchronizing pair of processes: Plank [9] suggests that staggered checkpoints can *increase* the overhead, if the application does a lot of synchronization. While we agree

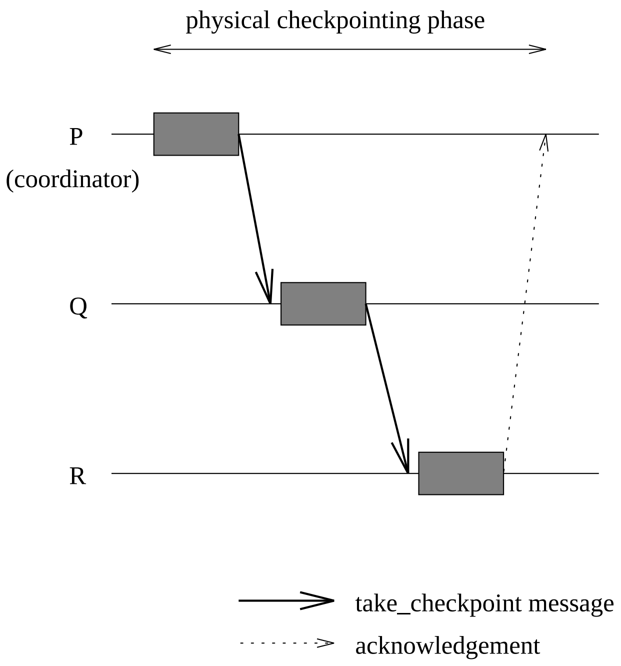

Figure 6: Reducing the messages required in phase 1

with this suggestion as such, it may be possible to overcome the problem is some systems. Figure 7 (based on an example in [9]) illustrates two processes (P and Q) that communicate as follows: Whenever one process sends a message to the other, the other acknowledges with another message. The first process stalls until the acknowledgement is received. During normal operation, the synchronization operation does not introduce any (significant) stalls. When the checkpoints are staggered, however, the first process may have to wait for the second process to complete a checkpoint before the acknowledgement will be sent. With non-staggered checkpoints, the duration of the wait could be much smaller.

A solution to this problem may be to allow a process to deliver (and respond to) an "urgent" message even when it is taking a physical checkpoint. However, delivery of the message may modify the state that is being checkpointed. Therefore, the checkpointing procedure should include these modifications (incremental change) in the final checkpoint. Figure 8 illustrates this. In this case, the physical checkpoint stored on the stable storage will reflect the state of process P *after* message M was delivered.

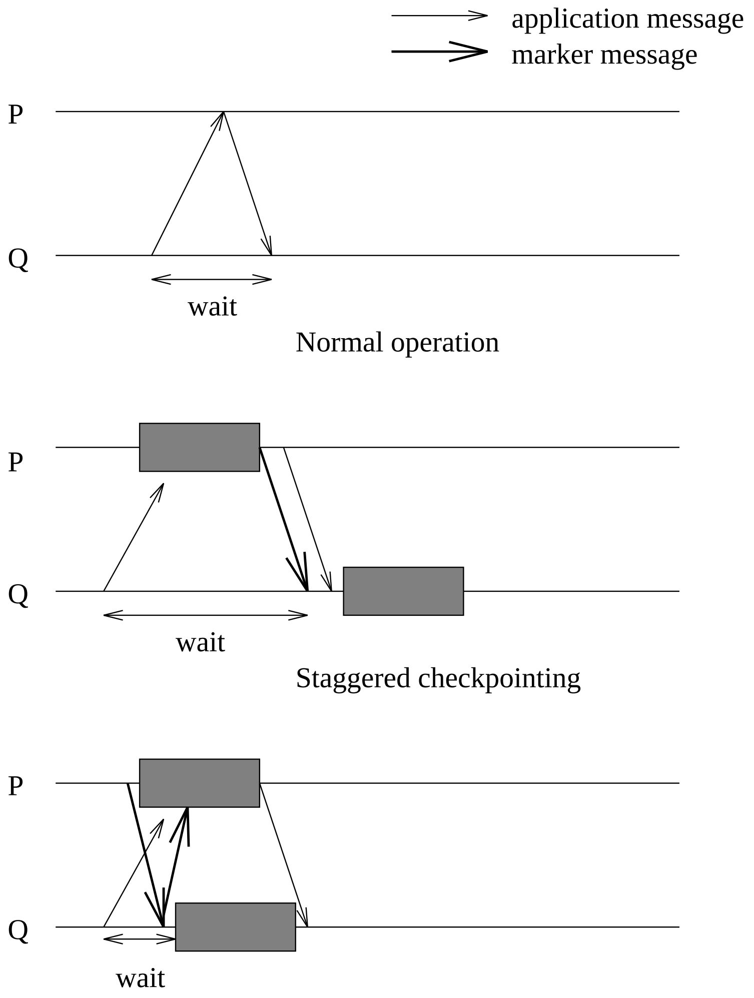

Figure 7: Staggering checkpoints with a synchronizing pair of processes (based on [9])

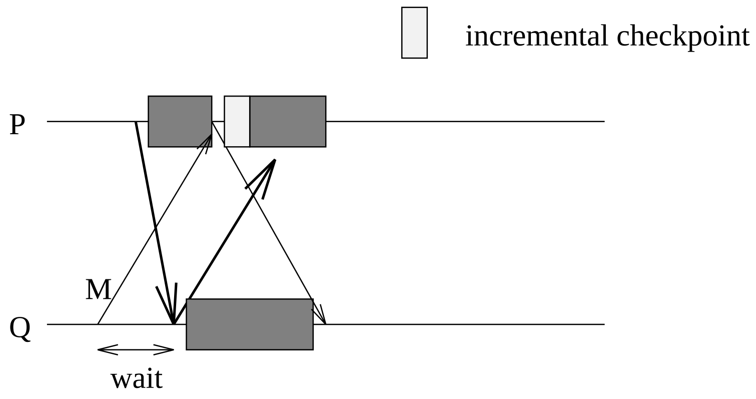

Figure 8: Delivering a message while checkpointing

7 Summary

This report presents an algorithm for taking consistent *logical* checkpoints. The proposed approach ensures that the *physical* checkpoints taken by various processes are completely staggered to minimize the contention in accessing the stable storage. The report also suggests a number of variations of the proposed approach. Performance evaluation of the proposed algorithm is a subject of future work [6]. The experimental results will be made available in a future revision of this report.

References

[1] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states in distributed systems," *ACM Trans. Comp. Syst.*, vol. 3, pp. 63–75, February 1985.

[2] C. J. Date, *An Introduction to Database Systems.* Addison-Wesley, 1986.

[3] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," in *Symposium on Reliable Distributed Systems*, 1992.

[4] D. Jefferson, "Virtual time," *ACM Trans. Prog. Lang. Syst.*, vol. 3, pp. 404–425, July 1985.

[5] D. B. Johnson, "Efficient transparent optimistic rollback recovery for distributed application programs," in *Symposium on Reliable Distributed Systems*, pp. 86–95, October 1993.

[6] S. Kaul (Advisor: N. Vaidya), M.S. thesis, in progress, Texas A&M University.

[7] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. Softw. Eng.*, vol. 13, pp. 23–31, January 1987.

[8] J. Long, B. Janssens, and W. K. Fuchs, "An evolutionary approach to concurrent checkpointing," *submitted to IEEE Transactions on Parallel and Distributed Systems.*

[9] J. S. Plank, *Efficient Checkpointing on MIMD Architectures.* PhD thesis, Dept. of Computer Science, Princeton University, June 1993.

[10] R. E. Strom and S. A. Yemini, "Optimistic recovery: An asynchronous approach to fault-tolerance in distributed systems," *Digest of papers: The* 14^{th} *Int. Symp. Fault-Tolerant Comp.*, pp. 374–379, 1984.

[11] N. H. Vaidya, "Some thoughts on distributed recovery," Tech. Rep. 94-044, Computer Science Department, Texas A&M University, College Station, June 1994. Available via anonymous ftp from ftp.cs.tamu.edu in directory /pub/vaidya.

[12] Y. M. Wang and W. K. Fuchs, "Lazy checkpoint coordination for bounding rollback propagation," in *Symposium on Reliable Distributed Systems*, pp. 78–85, October 1993.

[13] Y. M. Wang, Y. Huang, and W. K. Fuchs, "Progressive retry for software error recovery in distributed systems," in *Digest of papers: The* 23^{rd} *Int. Symp. Fault-Tolerant Comp.*, pp. 138–144, 1993.

[14] Y. M. Wang, A. Lowry, and W. K. Fuchs, "Consistent global checkpoints based on direct dependency tracking." To appear in *Inform. Process. Lett.*