A Low-Overhead Recovery Technique Using Quasi-Synchronous Checkpointing

D. Manivannan and Mukesh Singhal
Computer and Info. Science Dept.
The Ohio State University
Columbus, OH 43210
(email:{manivann,singhal}@cis.ohio-state.edu)

Abstract

In this paper, we propose a quasi-synchronous checkpointing algorithm and a low-overhead recovery algorithm based on it. The checkpointing algorithm preserves process autonomy by allowing them to take checkpoints asynchronously and uses communication-induced checkpoint coordination for the progression of the recovery line which helps bound rollback propagation during a recovery. Thus, it has the easiness and low overhead of asynchronous checkpointing and the recovery time advantages of synchronous checkpointing. There is no extra message overhead involved during checkpointing and the additional checkpointing overhead is nominal. The algorithm ensures the existence of a recovery line consistent with the latest checkpoint of any process all the time. The recovery algorithm exploits this feature to restore the system to a state consistent with the latest checkpoint of a failed process. The recovery algorithm has no domino effect and a failed process needs only to rollback to its latest checkpoint and request the other processes to rollback to a consistent checkpoint. To avoid domino effect, it uses selective pessimistic message logging at the receiver end. The recovery is asynchronous for single process failure. Neither the recovery algorithm nor the checkpointing algorithm requires the channels to be FIFO. We do not use vector timestamps for determining dependency between checkpoints since vector timestamps generally result in high message overhead during failure-free operation.

Keywords: Distributed checkpointing, failure recovery, fault-tolerance.

1. Introduction

In a distributed system, the states of processes depend on one another due to inter-process communication. So, when a process P rolls back after a failure, the processes that have states directly or transitively dependent on P’s state are forced to rollback. The use of checkpoints on a stable storage and rollback-recovery protocols are well established techniques for dealing with process failures in a distributed system. When a failure occurs, a rollback protocol uses the checkpoints and message logs to restore the system to a consistent global state [12]. In the literature, several checkpointing schemes have been proposed for distributed systems. They can be broadly classified into two categories - asynchronous and synchronous.

In asynchronous checkpointing [3], processes take checkpoints periodically without any coordination with others. To recover from a failure, a process communicates with other processes to determine if their local states are causally related. If they are, processes that received messages which are responsible for causal dependencies, roll back to eliminate these causal dependencies. This process is repeated until the local states of all the processes are free from causal dependencies. This approach allows maximum process autonomy and has low checkpointing overhead. However, this approach may suffer from the domino effect, in which the processes roll back recursively while determining a consistent set of checkpoints. To reduce domino effect, Kim et al. [9] and Venkatesh et al. [17] use the dependency tracking and insert checkpoints before processing a new message that introduces dependency. Message logging [6, 7, 14, 16] and message reordering [19] have been suggested in the literature to cope with the domino effect.

In synchronous checkpointing schemes, domino-free recovery is achieved by sacrificing process autonomy and incurring extra message overhead during checkpointing. In this approach, processes synchronize their checkpointing activities so that a globally consistent set of checkpoints is always maintained in the system [10]. The storage requirement for the checkpoints is minimum because each process keeps only one checkpoint in the stable storage at any given time. Process execution may have to be suspended during the checkpointing coordination as in [8, 10], resulting in performance degradation.

Message logging along with checkpointing is generally used to restore the system to a consistent state in the event of a failure. Message logging schemes can be broadly classified in to two categories - optimistic and pessimistic. In pessimistic message logging, messages received by a process are stored in the stable storage before they are processed. Thus, the stable logged information across the processes help make the recovery faster. The drawback of pes-
The rest of the paper is organized as follows. In the next section, we present the system model. In Section 3, we present the quasi-synchronous checkpointing algorithm. In Section 4, we present a basic recovery algorithm and prove its correctness; the recovery algorithm is based on the quasi-synchronous checkpointing algorithm. The recovery algorithm is fully asynchronous for single process failure— a failed process only needs to rollback to its latest checkpoint and request the other processes to roll back. The rollback distance of a process due to a failure is bounded. This helps a process to determine and discard the garbage checkpoints from the stable storage asynchronously. To handle the different types of messages, we use neither fully optimistic message logging nor pessimistic message logging. In our approach, messages are logged selectively but pessimistically, which means processes log only those messages that would be required for replay when a failure occurs.

The rest of the paper is organized as follows. In the next section, we present the system model. In Section 3, we present the quasi-synchronous checkpointing algorithm. In Section 4, we present a basic recovery algorithm and prove its correctness; the recovery algorithm is based on the quasi-synchronous checkpointing algorithm. The recovery algorithm is fully asynchronous for single process failure— a failed process only needs to rollback to its latest checkpoint and request the other processes to roll back. The rollback distance of a process due to a failure is bounded. This helps a process to determine and discard the garbage checkpoints from the stable storage asynchronously. To handle the different types of messages, we use neither fully optimistic message logging nor pessimistic message logging. In our approach, messages are logged selectively but pessimistically, which means processes log only those messages that would be required for replay when a failure occurs.

The rest of the paper is organized as follows. In the next section, we present the system model. In Section 3, we present the quasi-synchronous checkpointing algorithm. In Section 4, we present a basic recovery algorithm and prove its correctness; the recovery algorithm is based on the quasi-synchronous checkpointing algorithm. The recovery algorithm is fully asynchronous for single process failure— a failed process only needs to rollback to its latest checkpoint and request the other processes to roll back. The rollback distance of a process due to a failure is bounded. This helps a process to determine and discard the garbage checkpoints from the stable storage asynchronously. To handle the different types of messages, we use neither fully optimistic message logging nor pessimistic message logging. In our approach, messages are logged selectively but pessimistically, which means processes log only those messages that would be required for replay when a failure occurs.

The rest of the paper is organized as follows. In the next section, we present the system model. In Section 3, we present the quasi-synchronous checkpointing algorithm. In Section 4, we present a basic recovery algorithm and prove its correctness; the recovery algorithm is based on the quasi-synchronous checkpointing algorithm. The recovery algorithm is fully asynchronous for single process failure— a failed process only needs to rollback to its latest checkpoint and request the other processes to roll back. The rollback distance of a process due to a failure is bounded. This helps a process to determine and discard the garbage checkpoints from the stable storage asynchronously. To handle the different types of messages, we use neither fully optimistic message logging nor pessimistic message logging. In our approach, messages are logged selectively but pessimistically, which means processes log only those messages that would be required for replay when a failure occurs.
receives a message, it takes a checkpoint before processing the message if the sequence number of its current checkpoint is less than the checkpoint sequence number received with the message. Now, we present the quasi-synchronous checkpointing algorithm formally.

### 3.2. The Quasi-synchronous Algorithm

**Data Structures at Process** $P_i$

$\text{sn}_i : \text{integer}(= 0);$

next$_i : \text{integer}(= 1);$

When it is time for process $P_i$ to increment next$_i$,

$\text{next}_i := \text{next}_i + 1; /* next$_i$ incremented periodically */$

When it is time for process $P_i$ to take a basic checkpoint

If next$_i > \text{sn}_i$ then

$\text{Take checkpoint } C; /* it had not already taken a */$

$\text{C.sn} := \text{next}_i; /* forced checkpoint } C \text{ with } C \text{.sn} > \text{next}_i; /*$

$\text{sn}_i := \text{C.sn}; /* number for the checkpoint taken */$

$\text{next}_i := C \text{.sn}; /* update } sn_i */$

When process $P_i$ sends a message $M$

$M \text{.sn} := \text{sn}_i; /* sequence number of the current checkpoint */$

send $(M).$

When Process $P_j$ Receives a message $M$ from process $P_i$

If $M \text{.sn} > \text{sn}_j$ then

$\text{Take checkpoint } C; /* the message if } M \text{.sn} > \text{sn}_j */$

$\text{C.sn} := M \text{.sn}; /* C \text{.sn} and } C \text{.sn } */$

$\text{sn}_j := C \text{.sn}; /*$ Process the message.

Now, we explain the Quasi-synchronous checkpointing algorithm (hereafter referred to as QSA) with an example.

![Figure 1. Example illustrating the QSA](image)

A system consisting of three processes $P_1$, $P_2$, and $P_3$ is shown in Figure 1. The basic checkpoints are shown in the figure as "[" and the forced checkpoints are shown as "["]. The sequence numbers assigned to checkpoints are also shown in the figure. Here, each process $P_i$ increments its variable next$_i$ every $x$ time units. Process $P_3$ takes a basic checkpoint every $x$ time units, and $P_2$ takes a basic checkpoint every $2 \times x$ time units, and $P_1$ takes a basic checkpoint every $3 \times x$ time units. Message $M_0$ forces $P_3$ to take a forced checkpoint with sequence number 2 before processing $M_0$. As a result, $P_3$ skips taking a basic checkpoint with sequence number 2. Message $M_1$ forces $P_2$ to take a forced checkpoint with sequence number 3 before processing $M_1$ because $M_2 \text{.sn} = 3$ and $M_3 \text{.sn} = 2 < 3$ while receiving the message. Similarly, message $M_2$ forces $P_1$ to take a checkpoint before processing the message, and $M_4$ forces $P_2$ to take a checkpoint before processing the message. However, $M_3$ does not force the receiving process to take a checkpoint before processing it. Note that there may be gaps in the sequence numbers assigned to checkpoints. For example, process $P_1$ does not have checkpoints with sequence numbers 1 and 2.

### 4. Basic Recovery Algorithm

For the recovery, we assume that, if a process $P_i$ fails, then no other process fails until the system is restored to a consistent state. The recovery algorithm is fully asynchronous; in other words, a failed process rolls back to its latest checkpoint and informs other processes about its failure and resumes its normal computation without waiting for any reply from the other processes. In addition to the integer variables $\text{sn}_i$ and next$_i$, each process $P_i$ has two other integer variables $\text{inc}_i$ and $\text{rec}\_\text{line}_i$. The values of all these variables are kept in the stable storage so that they are available in the event of process failure. $\text{inc}_i$ the incarnation number of process $P_i$ and $\text{rec}\_\text{line}_i$ contains the recovery line number. Initially, $\text{inc}_i = 0$, and $\text{rec}\_\text{line}_i = 0 \forall i$. With each message $M_i$, the current values of the three variables $\text{inc}_i$, $\text{sn}_i$, and $\text{rec}\_\text{line}_i$ are piggybacked. The values of these variables piggybacked with $M$ are denoted by $M \text{.inc}$, $M \text{.sn}$, and $M \text{.rec}\_\text{line}$, respectively. We now present the basic recovery algorithm incorporating the quasi-synchronous checkpointing. Hereafter, we refer to the basic recovery algorithm presented below as BRA.

**The Basic Recovery Algorithm**

**Data structures at Process** $P_i$

$\text{sn}_i : \text{integer}(= 0); /* sequence number of current checkpoint */$

next$_i : \text{integer}(= 1); /* sequence number to be assigned to the */$

$\text{inc}_i : \text{integer}(= 0); /* next basic checkpoint, initialized to 1 */$

$\text{rec}\_\text{line}_i : \text{integer}(= 0); /* sequence number of current incarnation */$

When it is time for process $P_i$ to increment next$_i$,

$\text{next}_i := \text{next}_i + 1; /* next$_i$ incremented at periodic time */$

When it is time for process $P_i$ to take a basic checkpoint

If next$_i > \text{sn}_i$ then

$\text{Take checkpoint } C; /* it had not already taken a */$

$\text{C.sn} := \text{next}_i; /* forced checkpoint } C \text{ with } C \text{.sn} > \text{next}_i; /*$

$\text{sn}_i := \text{C.sn}; /* number for the checkpoint taken */$

$\text{next}_i := C \text{.sn}; /* update } sn_i */$

When process $P_i$ sends a message $M$

$M \text{.sn} := \text{sn}_i; /* sequence number of the current checkpoint */$

send $(M).$

When process $P_j$ receives a message $M$

If $M \text{.sn} > \text{sn}_j$ then

$\text{Take checkpoint } C; /* the message if } M \text{.sn} > \text{sn}_j */$

Process $P_i$ now has two new messages $M_i$ and $M_j$. It is clear from the figure that $M_i$ is piggybacked with $M_j$.

If $M \text{.inc} > \text{inc}_i$ then

$\text{inc}_i := M \text{.inc}; /*$ piggybacked with message $M$ */

$\text{rec}\_\text{line}_i := M \text{.rec}\_\text{line}; /*$ current recovery line number */

$\text{send } (M);$

When process $P_j$ receives a message $M$

If $M \text{.sn} > \text{sn}_j$ then

$\text{Take checkpoint } C; /* the message if } M \text{.sn} > \text{sn}_j */$

$\text{RollBack}(P_i);$

$\text{rec}\_\text{line}_i := M \text{.rec}\_\text{line}; /*$ rollback message */
We prove Theorem 1 that $S_i$ is a consistent global checkpoint. Here, $m_i = \text{rec.line}_i$.

**Proof:** From Observation 1, we have

$$\forall i, 1 \leq i \leq N : \ m_i \geq \text{rec.line}_i. \quad (1)$$

Suppose $S$ is not a consistent global checkpoint. Then, there exists a message $M$ sent by some process $P_j$ to some process $P_k$ such that $\text{send}(M) \neq C_{j,m_j}$ but $\text{receive}(M) = C_{j,m_j}$. So,

$$\text{M.sn} \geq m_j \text{ from Observation 2} \quad (2)$$

$$\text{M.sn} < m_k \text{ from Observation 3} \quad (3)$$
From equations 1, 2, and 3, we get,
\[
\text{rec.line}_i < m_j < M.sn < m_k
\]
(4)

From Observation 1, all the checkpoints taken by process \( P_k \) prior to checkpoint \( C_{k,m_k} \) have sequence numbers less than \( \text{rec.line}_i \). Since \( M.sn \geq \text{rec.line}_i \), \( P_k \) must have processed \( M \) only after taking a checkpoint with sequence number \( \geq \text{rec.line}_i \) (from Observation 4). Since \( \text{receive}(M) \in C_{k,m_k} \), \( M \) must have been received by \( P_k \) before the checkpoint \( C_{k,m_k} \) was taken. Hence, there exists a checkpoint \( C_{k,m'_k} \) of \( P_k \) that was taken before \( C_{k,m_k} \) such that \( m'_k \geq \text{rec.line}_i \). This contradicts the fact that all the checkpoints taken by process \( P_k \) prior to checkpoint \( C_{k,m_k} \) have sequence numbers less than \( \text{rec.line}_i \). So, our assumption that \( S \) is not a consistent global checkpoint in wrong. Hence, the theorem. □

Under BRA, processes roll back to a consistent global checkpoint in the event of a process failure. As a result of the roll back, the reception of some messages might have been undone while the corresponding send event might not have been undone. So, even though the processes roll back to a consistent global checkpoint, it may not leave the system in a consistent state. In the next section, we present a method for restoring the system to a consistent state handling all the types of messages that arise during recovery in the event of a failure.

5. Comprehensive Recovery

In this section, we describe how we can modify the BRA to restore the system to a consistent state after rolling back the processes to a consistent global checkpoint. Rolling back the processes to a consistent global checkpoint may result in undoing the send and/or receive events of many messages. This may result in several abnormal situations which must be dealt with correctly in order to restore the system to a consistent state. Classifying messages into various types will help clarify the message management issues involved in recovery. So, before explaining the modifications required for the BRA, we classify the messages that need to be handled during recovery into different types.

5.1. Message Classification

We use Figure 2 to help clarify the message classification. Suppose process \( P_4 \) in Figure 2 fails at the point marked by \( X \) and rolls back to its latest checkpoint \( C_{1,8} \). As a result, processes \( P_2 \), \( P_3 \), and \( P_4 \) roll back to the checkpoints \( C_{2,9}, C_{3,8}, \) and \( C_{4,8} \) respectively under BRA. The recovery line corresponding to this failure is shown in the figure.

Lost messages: Messages whose send events are not undone but whose receive events are undone due to rollback. This type of messages arise when a process rolls back to a checkpoint prior to the reception of the message while the sender does not roll back to a checkpoint prior to the send event of that message. In Figure 2, \( M_1 \) is a lost message.

Delayed Messages: Messages that were sent before the rollback whose receive events are not recorded because the messages were received either while the receiving process was down or received after the the rollback of the receiving process. For example, messages \( M_2 \) and \( M_3 \) in Figure 2 are examples of delayed message.

Orphan messages: Messages which have been received and whose send has been undone due to rollback but whose receive has not been undone. Orphan messages do not arise if processes roll back to a consistent global checkpoint. So, under BRA, orphan messages do not arise.

Duplicate Messages: This happens due to message logging and replaying of the messages in the log during recovery. For example, in Figure 2, message \( M_4 \) was sent and received before the rollback. Due to rollback, \( P_4 \) has undone the receive of \( M_4 \) and \( P_3 \) has undone the send of \( M_4 \). So, after \( P_4 \) rolls back to \( C_{4,8} \), it should not replay the message \( M_4 \) since \( P_3 \) has undone the send event of \( M_4 \); if \( P_4 \) replays \( M_4 \), then \( M_4 \) will be a duplicate message, since it will be sent again by \( P_3 \).

Since the BRA rolls back the processes to a consistent global checkpoint, orphan messages do not arise. Since we assume that inactive or failed processes can be detected by other processes, a message lost because of the failure of the receiving process will be resent. Thus, we need only handle delayed messages that are received after a failed process recovers from failure, lost messages, and duplicate messages.

5.2. Modifications to the BRA for Message Handling

We modify the BRA so that when a process rolls back, it will replay only lost messages from its message log and handle the delayed messages appropriately. To accomplish this, we allow processes to log received messages selectively and replay logged messages selectively after a rollback. We explain later how messages to be logged and replayed are determined. We use Figure 3 to explain what messages are replayed when a process rolls back, and how a received message is handled.

Handling Replay of Messages

First, we explain which messages are replayed from stable storage when a process rolls back. In Figure 3, suppose pro-
cess $P_i$ fails at the point $X$ and initiates recovery. Then, it rolls back to its latest checkpoint $C_{1,10}$, increments $inc_1$ to 1, sets $rec.line_1 := 10$, and sends $rollback(1, 10)$ message to all the other processes. When $P_2$ receives this rollback message, it sets $inc_2 := 1$, $rec.line_2 := 10$, and rolls back to $C_{2,12}$ since $C_{2,12}$ is the earliest checkpoint of $P_2$ whose sequence number is $\geq 10$. In the figure, messages sent before the rollback are shown by bold arrows and messages sent after the rollback are shown by broken arrows. So, $M.inc = 0$ for all the messages shown by bold arrows and $M.inc = 1$ for all the messages shown by broken arrows.

When a process rolls back, it must replay from its log all those messages whose receive was undone as a result of the rollback but whose send will not be undone; in other words, a process must replay only those messages that originated to the left of the the current recovery line and delivered to the right of the current recovery line. For example, after rolling back to $C_{2,12}$, $P_2$ must replay the messages $M_4$ and $M_5$ but must not replay $M_3$; this is because the send of $M_3$ has not been undone by $P_2$, the send of $M_5$ has not been undone by $P_1$, and the send of $M_3$ was undone by $P_2$ due to its rollback to the checkpoint $C_{4,10}$. After all the processes roll back in response to the $rollback(inc_i, rec.line_j)$ message sent by $P_i$, note that $rec.line_i = rec.line_j$ for all $j$. This common value, we call the recovery line number. For example, the recovery line number for the rollback shown in the figure is 10. It is easy to determine if a message $M$ was sent before the current recovery line by looking at $M.sn$. In fact, a message $M$ was sent before the current recovery line if and only if $M.sn$ is less than the recovery line number of the current recovery line. So in general, the rule for replaying messages after a process rolls back is:

**Message replaying rule:** After a process $P_j$ rolls back to checkpoint $C$, it replays a message $M$ from its message log if and only if it was received after the checkpoint $C$ was taken and $M.sn < rec.line_j$.

### Handling Received Messages

Next, we discuss how a received message is handled by a process. Suppose process $P_j$ receives a message $M$ from process $P_i$. If $P_j$ is replaying messages from its message log as a result of a rollback at that time, then it buffers the message $M$ and process it only after finishing replaying. Otherwise, the following three cases arise:

**case(i):** $M$ was sent in a previous incarnation (i.e., $M.inc < inc_j$) (In other words, $M$ is a delayed message). This means that $P_i$ was not aware of the current recovery at the time of sending the message $M$. In this case, if $M.sn < rec.line_j$ then $M$ is first logged in the message log and then processed; otherwise, it is discarded because $P_j$ will rollback to the earliest checkpoint whose sequence number is $\geq rec.line_j$, and hence will undo the send of $M$. If $M.sn < rec.line_j$, then the message needs to be logged before being processed because if $P_j$ has to rollback to the current checkpoint due to the failure of some process and if the recovery line number of that rollback is greater than $M.sn$, then $M$ will have to be replayed. In the figure, $M_6$ is logged and then processed by $P_j$, whereas $M_8$ is discarded by $P_j$ because $M_8.sn = 11 > rec.line_{10} = 10$ and $M_8.inc = 0 < inc_{10} = 1$. So, in general, a message sent in a previous incarnation must be logged and processed if it was sent prior to the current recovery line; otherwise, the message should be discarded because it is a delayed message. Thus, we follow the following rule for deciding delayed messages:

**Rule for Determining Delayed messages:** A message $M$ received by a process $P_j$ is a delayed message if $M.inc < inc_j$. A delayed message $M$ is processed by $P_j$ only if $M.sn < rec.line_j$; otherwise, it is discarded.

**case(ii):** $M$ was sent in the current incarnation (i.e., $M.inc = inc_j$). In this case, if $M.sn < sn_j$, then $M$ is logged before being processed; this is because if $P_j$ needs to rollback to the current checkpoint, $M$ will have to be replayed if the recovery line number of that rollback is $> M.sn$. For example, in the figure, message $M_7$ was sent by process $P_j$ after it had recovered and it was received by $P_2$ after $P_2$ rolled back and replayed all the messages that need to be replayed. In that case, $M_7.inc = inc_2 = 1$ and $M_7.sn = 10 < sn_2 = 12$; $M_7$ has to be logged before being processed because if $P_2$ has to rollback to checkpoint $C_{2,12}$ due to its failure or due to the failure of some other process in the future, then $P_2$ will have to replay $M_7$ if the recovery line number of that rollback is $> 10$. Note that if $M.sn > sn_j$ then, message $M$ forces $P_j$ to take a checkpoint with sequence number $M.sn$ before processing it and if $M.sn = sn_j$, the message is processed without logging or taking checkpoint. Thus, we have the following message logging rule:

**Message Logging Rule:** When a process $P_j$ receives a message $M$, it logs the message before processing it if ($M.inc < inc_j$ and $M.sn < rec.line_j$) or ($M.inc = inc_j$ and $M.sn < sn_j$).

**case(iii):** $M$ was sent in a future incarnation (i.e., $M.inc > inc_j$). In this case, $P_j$ sets $rec.line_j := M.rec.line$ and $inc_j := M.inc$, and rolls back to the earliest checkpoint whose se
In the figure, suppose $M_s$ had been received by $P_4$ before $P_4$ received the rollback(1, 10) message from process $P_1$, then that would be an example of a message received from a future incarnation.

6. An Overhead Analysis

In checkpointing algorithms, generally three kinds of overheads exist. First is the extra control messages required for checkpoint coordination, second is the control information sent on computation messages, and the third is the number of checkpoints that need to be taken (i.e., the checkpointing overhead). During the normal operation, each computation message is piggybacked with three integers. There is no other message overhead. So, we only analyze the checkpointing overhead involved. The checkpointing overhead depends on the run-time communication pattern as well as the basic checkpointing pattern.

Let $basicnum$ denote the total number of checkpoints the processes would take for a basic checkpointing pattern; let $quasimnum$ denote the total number of checkpoints the processes would take for the same computation and basic checkpointing pattern when the QSA is used. We define the induction ratio $R$ for a computation as

$$ R = \frac{quasimnum}{basicnum} $$

We use $R$ to analyze the checkpointing overhead of the QSA. One would desire $R$ to be as close to 1 as possible. If $R = 1$, then there is no extra checkpointing overhead induced by the QSA. If $R > 1$, then there is additional checkpointing overhead.

We analyze the checkpointing overhead under two given basic checkpointing patterns: (i) Each process takes basic checkpoints at fixed time intervals periodically, the time interval being same for all processes. (ii) Each process takes basic checkpoints at its own pace. The results of the analysis in these two cases are given in Assertions 1 and 2, respectively below. The proofs of these assertions can be found in [11].

**Assertion 1:** Assume that under a basic checkpointing pattern, each process takes a basic checkpoint at the end of every $x$ time units, and the local clocks of the processes can drift by at most $\delta$ where $\delta < \frac{1}{2} \times x$. Then the QSA has no additional checkpointing overhead for this basic checkpointing pattern. Thus, $R = 1$ and the algorithm ensures that the recovery line stays at the end of the computation.

**Assertion 2:** If processes take basic checkpoints at their own pace, then the induction ratio $R \leq \lfloor Q \rfloor$, where $Q$ is the maximum of the ratios of the lengths of the basic checkpoint intervals of any two processes. Moreover, no process needs to rollback to a distance more than $\lfloor Q \rfloor$ under the BRA.

Selective message logging reduces message logging overhead. Since the QSA ensures the existence of a recovery line consistent with the latest checkpoint, failed process starts from the latest checkpoint. From Assertion 2, a failed process may force other processes to roll back to a maximum distance of $Q$ checkpoints, where $Q$ is the maximum ratio of the checkpoint intervals of all the processes in the worst case. This bound on the rollback distance helps the process to decide garbage checkpoints asynchronously.

7. Comparison With Existing Work

The checkpointing algorithms proposed in [10, 2] have a two-phase structure. This causes processes to suspend the normal computation for making checkpoint decisions which greatly increases the overhead during normal computation. The QSA does not cause any such overhead and avoids domino effect completely during recovery. In Acharya et al.'s [1] asynchronous checkpointing algorithm for mobile computing systems, a process takes a checkpoint whenever a message reception is preceded by a message transmission. This might force the processes to take as many checkpoints as the number of messages if the message reception and transmission are interleaved, which would result in high checkpointing overhead.

Wang et al. [18] proposed lazy checkpoint coordination for bounding rollback propagation. Like our approach, their technique requires the checkpoint number being piggybacked on the computation messages so that the receiving processes can take an extra checkpoint when required. However, their approach has higher checkpointing overhead since processes do not skip basic checkpoints if they have taken forced checkpoints.

Many of the existing recovery algorithms use vector timestamps to track dependency between checkpoints and events. Vector timestamps generally result in high message overhead during failure-free operation. Elnozahy and Zwaenepoel [4] discuss the Manetho rollback recovery protocol, which is based on antecedence graphs. The recovery method is complex, but only failed process needs to rollback. Richard III and Singhal [5] proposed a recovery algorithm based on vector timestamps. Our recovery algorithm does not require vector timestamps. Channels need not be FIFO. Recovery is fully asynchronous for single process failure. Recovery requires only one rollback message to be sent to the other processes. Maximum number of rollbacks of a process per failure is 1. When a process whose latest checkpoint has maximum sequence number fails, it will not force any other process to rollback. If the process whose latest checkpoint has lowest sequence number fails, it will force the other processes to roll back to a distance of at most $Q$ checkpoints where $Q$ is the maximum of the ratios of the checkpoint interval lengths of any two processes.

The optimistic recovery algorithm proposed by Strom and Yemini [16] suffers from domino effect. The recovery protocols proposed by Peterson and Kearns [12] is synchronous and tolerates single process failure. It requires the channels to be FIFO. Recovery proposed by Sistla and Welch [14] is synchronous for single process failure, requires the channels to be FIFO and uses vector timestamps. Smith and Johnson [15] proposed an asynchronous recovery algorithm for multiple process failures; however, the size of the vector clock is $\mathcal{O}(N^2)$ which results in high overhead during failure free operation.
8. Conclusion

We presented a novel quasi-synchronous checkpointing algorithm and a comprehensive recovery algorithm based on it. The checkpointing algorithm has the easiness of asynchronous checkpointing and the advantages of synchronous checkpointing. Communication-induced checkpointing coordination guarantees the progression of the recovery line. The algorithm has no additional control message overhead and has nominal checkpointing overhead. It also guarantees the existence of a recovery line consistent with the latest checkpoint of any process all the time. The recovery algorithm presented exploits this property of the checkpointing algorithm to restore the system to a consistent state asynchronously, in the case of single process failure. Unlike some of the existing recovery algorithms, our recovery algorithm does not use vector timestamps for tracking dependency between checkpoints. The overhead involved in the recovery is very low since messages are logged and replayed selectively and there is no explicit synchronization overhead involved during recovery.

References


