1987

Concurrent Checkpointing and Recovery in Distributed Systems

Pei-Jyun Leu

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Report Number:
87-689

CONCURRENT CHECKPOINTING AND RECOVERY IN DISTRIBUTED SYSTEMS

Pei-Jyun Leu
Bharat Bhargava

CSD-TR-689
June 1987
(Revised October 1988)

Concurrent Checkpointing and Recovery in Distributed Systems*

Pei-Jyun Leu and Bharat Bhargava

Department of Computer Sciences

Purdue University

West Lafayette, IN 47907

ABSTRACT

This paper studys concurrency issues in distributed checkpointing and rollback recovery. It transforms the concurrent checkpointing and recovery problem to a transaction processing problem. A new transaction model, which consists of four types of atomic operations and five types of conflicts, is used for distributed checkpointing and recovery. Each transaction is executed by multiple processes in the system. We have shown that the consistency of recovery lines and rollback lines established by checkpoint transactions and rollback transactions can be achieved by enforcing serializability on the corresponding transactions. An algorithm is designed to expand and execute checkpoint transactions or rollback transactions concurrently. The algorithm supports efficient recovery, reduces the response time of checkpoint transactions and rollback transactions, and allows normal messages to be transmitted in any order. We have implemented the algorithm for performance evaluation. The analysis shows that concurrent execution reduces the response time of checkpoint transactions and rollback transactions. The CPU cost is in a linear order of the total number of synchronization messages used. For a checkpoint/rollback transaction with eight participating processes, the CPU cost is significantly smaller than the single checkpoint/rollback cost when the processes are bigger than 12K bytes.

* This work was supported in part by UNISYS, and NASA, and a David Ross fellowship.

** An earlier version of this paper appears in *Proc. IEEE 4th Conf. Data Engineering*, Los Angeles, CA, Feb. 1988.

1. Introduction

Rollback recovery is a technique to eliminate transient errors in a system. Transient errors may occur anywhere during the computation, and may not be detected immediately. The causes of the transient errors can be unstable hardware, software bugs, or illegal operations. To reduce the rollback distance, the system periodically saves correct system states in stable storage [LAMPS79]. When transient errors are captured, the system restores the last checkpointed state, and restarts. Since the computation after the last checkpoint may have been contaminated by the transient error, there is no need to roll forward after rolling back.

In a distributed system, where processes do not share memory, and message passing is the only way to communicate, a global state must be checkpointed distributively over all processes. If the processes make checkpoints without synchronization, *domino effect* [RAND75, RAND78] may occur. Further, the restoration of a previous global state must also be synchronized among the processes. Otherwise, *cyclic restoration* may occur [KOO87]. This represents a problem that a process after rolling back receives messages subsequently undone by the sender, and thus it has to roll back again. In such a case, the rollback of one process will cause the rollback of the other, and a cyclic effect can repeat forever. This problem can be solved as follows. A process after rolling back holds all subsequent incoming normal messages in a buffer until the other rollback process also finishes its rollback. The receiver then determines which messages in the buffer have been undone by the sender, and thus must be discarded. The receiver extracts the remaining messages for its local computation. We call one instance of the checkpoint algorithm executed on multiple processes a *checkpoint instance*. Similarly, we call one instance of the rollback algorithm executed on multiple processes a *rollback instance*.

Distributed checkpointing and rollback recovery have been studied in [BARI83, KOO87, TAMI84]. In contrast to transaction checkpointing [FISC82, GRAY79, MOSS83] that deals with the consistency of database, the problem is mainly concerned with the consistency of the process states. We can summarize the past research [BARI83, KOO87, TAMI84] as follows.

Since transient errors may interrupt the execution of one checkpoint instance, 1) participating processes follow two-phase commit [GRAY79] to ensure the atomicity of one checkpoint instance; and 2) each participating process keeps both its last checkpoint and the newly made checkpoint until the checkpoint instance can commit. In [BARI83, KOO87], processes that have exchanged message their last checkpoints need to take checkpoints or roll back together. The processes participating in a checkpoint instance or a rollback instance constitute a virtual tree. Two-phase commit is performed hierarchically. The root process serves as the coordinator. In [TAMI84], all the processes in the system need to take checkpoints or roll back together each time. In [BARI83, KOO87], different checkpoint instances and rollback instances can interfere with one another. Interfering instances imply that the corresponding virtual trees overlap. In [BARI83], the interference problem among multiple checkpoint instances is solved by merging overlapping trees. A new coordinator is selected from among the roots of the overlapping trees to conduct the execution of the algorithm. In [KOO87], the interference problem is handled by allowing only one instance to complete but rejecting all other instances. There are several issues that need further study: concurrent execution of multiple checkpoint instances and rollback instances; non-FIFO channels that do not require the order of message send and message receive to be the same; and resiliency against process failures and communication failures.

We model concurrent checkpointing and recovery as a concurrent transaction processing problem. Checkpoint/rollback operations of multiple processes are organized as a transaction. We design an algorithm that executes checkpoint transactions or rollback transactions concurrently. A rollback transaction can always commit without being aborted. A checkpoint transaction is aborted only when it interferes with a rollback transaction. Blocking due to process failures or network partitioning is resolved using a termination protocol. Blocking of a rollback transaction can always be resolved. Blocking possibility of checkpoint transactions has been reduced. Further, our algorithm allows normal messages to be received in any order.

In section 2, we describe the concurrent checkpointing and recovery problem. We model this as a concurrent transaction processing problem. We show that the concurrent checkpointing and recovery problem can be solved by enforcing serializability on the corresponding transactions. A locking protocol is designed to enforce serializability on concurrent transactions. Some optimizations in synchronization of the concurrent transactions are discussed. Section 3 describes the checkpoint/rollback algorithm that uses the locking protocol to synchronize concurrent transactions. The optimizations are also incorporated. Two illustrative examples are given. Section 4 shows the correctness of the algorithm. In section 5, a comparison with related work is made. Section 6 evaluates the performance experimentally. Section 7 presents solutions to reduce blocking of transactions due to multiple process failures and network partitioning. Section 8 generalizes the algorithm when processes keep multiple checkpoints in the stable storage. A process may need to roll back to any previous checkpoint. The last section concludes the paper.

2. Concurrent Checkpointing and Recovery

2.1. The Problem

In a distributed system, processes communicate by exchanging messages. Messages generated by the sender may trigger some actions at the receiver. The distributed checkpointing and recovery problem deals with the synchronization of checkpoint operations, message passing operations, and rollback operations to ensure consistency. In Fig. 1, checkpoints C_i and C_j compose a *recovery line* from which the processes restart after rolling back. Rollback points U_i and U_j compose a *rollback line* from which the processes start rolling back to their last checkpoints. As part of rollback, process P_i undoes all actions in the period from C_i to U_i. There is no need to redo these actions after restart because they may have been caused by transient errors. If P_i has sent some messages in that period, then the receivers of the messages must also roll back to undo the actions triggered by the messages. In general, we cannot assume the sender, after it restarts, will regenerate those outgoing messages sent after the recovery line, because the sending of the outgoing messages may have been caused by a transient error. After a process restarts, it replays all incoming messages except those sent by the processes that have rolled back. Incoming messages are recorded in a message log for replaying purposes. An inconsistent state is caused when a sender rolls back, undoing some message send action, but the receiver does not undo the actions triggered by the message. We call such a message *dangling receive.* A message is undone if both the message send and its triggered actions are undone.

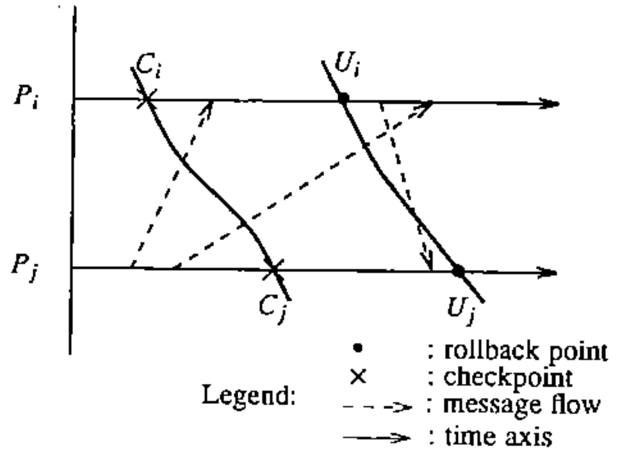

Figure 1. Consistent recovery line and rollback line.

Fig. 2 shows an inconsistent recovery line and an inconsistent rollback line. Processes P_i and P_j roll back to C_i and C_j respectively. During the rollback, P_i undoes the sending of m. P_j is supposed to undo all actions triggered by m. Since P_j only rolls back to C_j, actions triggered by m in the period T are not undone. Therefore, C_i and C_j compose an inconsistent recovery line. Similarly, rollback points U_i and U_j compose an inconsistent rollback line, because P_j rolls back, undoing the sending of l, but P_i rolls back before receiving l. Therefore, actions triggered by l at P_i are not undone. Fig. 1 shows a consistent recovery line and a consistent rollback line, where neither the sending of a message nor the receiving of the message is undone by the processes.

In summary, the consistency constraint of a recovery line can be described as follows. If a message is received before a checkpoint, then it must also be sent before a checkpoint. The consistency constraint of a rollback line can be described as follows. If the sender rolls back, undoing the sending of a message, the receiver of the message must also roll back to undo all

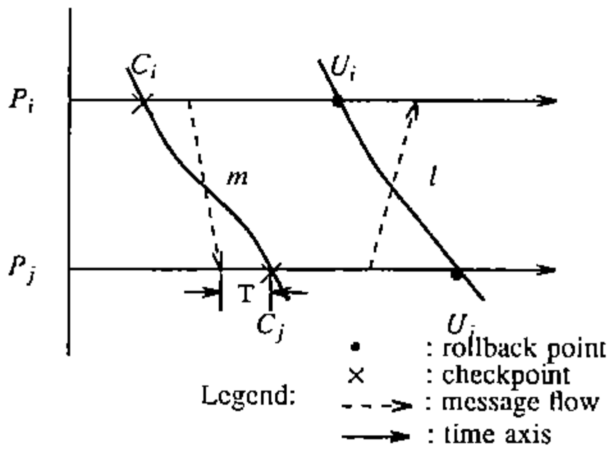

Figure 2. Inconsistent recovery line and rollback line.

actions triggered by the message.

2.2. The Approach

In the synchronous approach [BARI83, KOO87, TAMI84], processes synchronize their checkpoint operations, message operations, and rollback operations in order to maintain consistency. We model this problem as a concurrent transaction processing problem. We will show that the concurrent checkpointing and recovery problem can be solved by enforcing serializability on concurrent transactions. In this model, we couple each message send with a message receive as a message transaction. We group all checkpoint operations in an instance as a checkpoint transaction, and all rollback operations in an instance as a rollback transaction. Based on the Lamport clock [LAMPO78], there exists a partial order among all operations taking place in a distributed system. This order represents the "happen before" relationships among the distributed operations. This order is acyclic and transitive. Therefore, we can map distributed operations to points on a global time axis where all the partial order relationships among the distributed operations are preserved.

For example, in Fig. 3, we map distributed operations from processes P_i, P_j, P_k to operations on a global time axis. C represents a checkpoint operation. S stands for a message send. R represents a message receive. The sequence of operations on the global time axis can

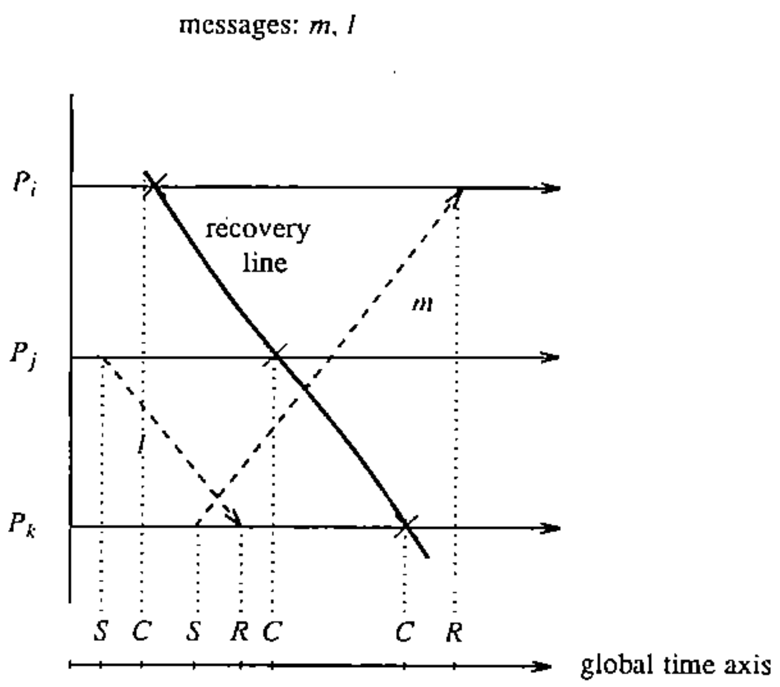

Figure 3. Concurrent checkpoint operations and message operations.

be represented by a log:

$$L = S_2[j]C_1[i]S_3[k]R_2[k]C_1[j]C_1[k]R_3[i].$$

Subscripts represent transaction indices. The variable in the brackets represents the index of the process that executes the operation. This log represents a possible concurrent execution of the three transactions:

$$T_1 = C_1[i]C_1[j]C_1[k]$$
$$T_2 = S_2[j]R_2[k]$$
$$T_3 = S_3[k]R_3[i]$$

The checkpoint transaction T_1 is executed by all processes in the system. These checkpoints made by the processes compose a recovery line. The message transaction T_2 sends a message from process P_j to process P_k. The message transaction T_3 sends a message from process P_k to process P_i. We use $\sigma(T_a)$ to denote the set of indices of the processes that execute transaction T_a. σ_0 represents the set of indices of all processes in the system. The three types of transactions are described as follows:

- A checkpoint transaction T_a is a sequence of operations $\{ C_a[i] \mid i \in \sigma(T_a) = \sigma_0 \}$. $C_a[i]$ is a checkpoint operation executed by process P_i.
- A rollback transaction T_a is a sequence of operations $\{ U_a[i] \mid i \in \sigma(T_a) = \sigma_0 \}$. $U_a[i]$ is a rollback operation executed by process P_i.
- A message transaction T_a is a sequence of the two operations $\{ S_a[i], R_a[j] \}$, $i \neq j$, $i, j \in \sigma_0$. This means the message is sent from process P_i to process P_j.

For simplicity, we may omit the transaction indices of operations of different types in the discussion. Each checkpoint transaction establishes a recovery line. Each rollback transaction establishes a rollback line. The consistency of recovery lines and rollback lines can be assured by enforcing serializability on the dependency order among concurrent transactions. The dependency order among the concurrent transactions is determined by the order of their conflicting operations. Two operations conflict if they are not commutable in a log. The log may produce a different result if we switch two conflicting operations. For example, in the log

$$L = \{x := x + 1\}\{x := x * 2\},$$

the two atomic operations {increment x by 1} and {multiply x by 2} are not commutable.

Definition 1. Let $O_a[i]$ and $O_b[j]$ be operations of two transactions T_a and T_b respectively. O_a precedes O_b in a log. The two operations conflict iff

i) they are executed by the same process, i.e., $i = j$, and

ii) $f(O_a, O_b) = \times$, where f is a binary function defined in Table 1.

We next define the dependency order among concurrent transactions. We design a locking protocol to enforce serializability. Several optimizations are discussed.

Definition 2. Transactions T_a and T_b are dependent (denoted by $T_a \rightarrow T_b$) in a log if

i) some operation O_a of T_a precedes and conflicts with some operation O_b of T_b, $a \neq b$; or

Table 1. Operation dependency table.

f	C	S	R	U
C	○	×	○	×
S	×	○	○	×
R	×	○	○	○
U	×	×	*	○

C: checkpoint operation
S: message send operation
R: message receive operation
U: rollback operation
*: f(U, R) = × if the sending of the message is undone, and ○ otherwise.

ii) there exists T_l such that $T_a \rightarrow T_l$ and $T_l \rightarrow T_b$.

This dependency relation is transitive, but may not be acyclic in concurrent transaction processing.

Definition 3. A log is D-serializable (dependency serializable) iff its dependency relation ($\rightarrow$) is acyclic [BERN79].

Theorem 1. Let L be a log of checkpoint transactions, message transactions, and rollback transactions. If L is D-serializable, then the recovery line established by each checkpoint transaction is consistent, and the rollback line established by each rollback transaction is consistent.

Proof: The proof is by contradiction. Since the dependency between any checkpoint transaction and any rollback transaction is acyclic, the recovery line and rollback line established by the two transactions do not intersect. We show that

i) If the recovery line established by a checkpoint transaction T is inconsistent, then L is not D-serializable.

This occurs when there exists a message transaction $M = S[i]R[j]$ such that $C[i]$

precedes $S[i]$, and $R[j]$ precedes $C[j]$ in the log from Fig. 2. Then we have $T \rightarrow M$ and $M \rightarrow T$ from Definitions 1 and 2. Therefore, L is not D-serializable.

ii) If the rollback line established by a rollback transaction T is inconsistent, then L is not D-serializable.

This occurs when there exists a message transaction $M = S[j]R[i]$ such that $U[i]$ precedes $R[i]$, and $S[j]$ precedes $U[j]$ in the log from Fig. 2. Then we have $T \rightarrow M$ and $M \rightarrow T$ from Definitions 1 and 2. Therefore, L is not D-serializable.

From i) and ii), the theorem follows. $\square$

We design a locking protocol to ensure the serializability of transactions. Each checkpoint transaction and rollback transaction follows a two-phase locking protocol [ESWA76]. Each message transaction follows a simple locking protocol, where a lock is obtained immediately before an operation is executed, and is released immediately after the operation is executed. A transaction issues operations to multiple processes. These processes execute the operations on behalf of the transaction. In order to synchronize checkpoint operations, rollback operations, message sends, and message receives, a process must also obtain a lock on behalf of the transaction before executing its operation. If a lock cannot be granted, the operation is delayed or aborted. In Fig. 4, we show how transactions follow the locking protocol. T_1 is a checkpoint transaction. M is a message transaction. T_2 is a rollback transaction. A geometric description of their locking sequences is shown in the lower part of the picture. Each rising edge represents a lock action. Each falling edge represents an unlock action. The corresponding recovery line and rollback line established by the transactions are shown in the upper part of the picture.

We define four types of locks: c-lock, s-lock, r-lock, and u-lock. c-locks are for checkpoint operations. s-locks are for message send operations. r-locks are for message receive operations. u-locks are for rollback operations. Two locks are compatible iff $g(l_1, l_2) = \bigcirc$, where g is a binary function defined in Table 2. If a process already holds a lock l_1, it cannot obtain any lock l_2, where $g(l_1, l_2) = \times$. We next formalize the locking protocol.

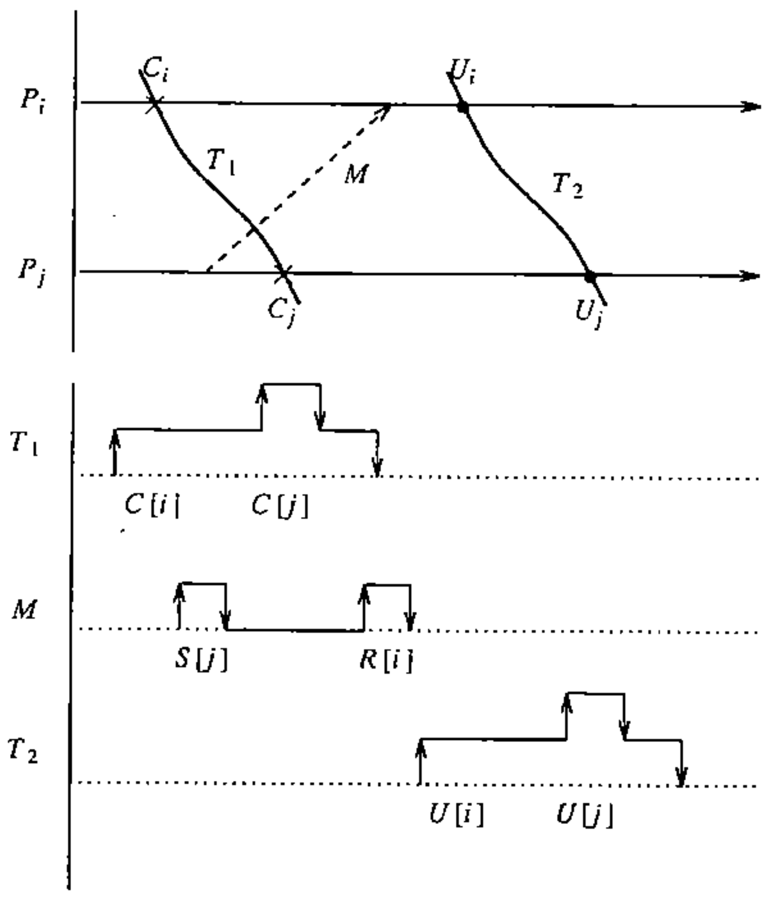

Figure 4. Concurrent transactions and their locking sequences.

Table 2. Lock compatibility table.

g	c	s	r	u
c	○	×	○	×
s	×	×	×	×
r	×	×	×	×
u	×	○	×	○

The Locking Protocol

For a checkpoint transaction $T = \cdots C[i] \cdots$, where $C[i]$ is any checkpoint operation of T:

- T issues $C[i]$ to cause process P_i to take an uncommitted checkpoint.
- Before P_i takes a checkpoint, P_i must obtain a c-lock.
- T starts to commit checkpoint operations only after all the checkpoint operations have been executed.
- P_i does not release the c-lock until it commits the checkpoint, and discards its previously committed checkpoint.
- For T to complete successfully, all processes must have committed their checkpoints, and released all c-locks.

For a message transaction $M = S[i]R[j]$:

- Process P_i executes $S[i]$ by sending a message.
- Before P_i sends a message, P_i must obtain an s-lock. P_i releases the lock immediately after sending the message.
- Process P_j executes $R[j]$ by receiving a message.
- Before P_j receives a message, P_j must obtain a r-lock. P_j releases the lock immediately after receiving the message

For a rollback transaction $T = \cdots U[i] \cdots$, where $U[i]$ is any rollback operation of T:

- T issues $U[i]$ to cause process P_i to roll back to its last checkpoint. If P_i has sent some message to P_j since its last checkpoint, P_i informs P_j to ignore the message when P_j receives the message.
- Before P_i rolls back, P_i must obtain a u-lock.
- T starts to commit rollback operations only after all the rollback operations have been executed.
- P_i does not release the u-lock until it commits the rollback operation.
- For T to complete successfully, all processes must have rolled back to their last committed checkpoints, and released all u-locks.

Theorem 2. The locking protocol assures the serializability of concurrent checkpoint transactions, and message transactions, and rollback transactions.

Proof:

i) The dependency order between any checkpoint transaction and any rollback transaction is acyclic, because they follow a two-phase locking protocol.

ii) Suppose the dependency between a checkpoint transaction T and a message transaction $M = S[i]R[j]$ is cyclic. Then we have $C[i]$ precedes $S[i]$, and $R[j]$ precedes $C[j]$ in the log from Fig. 2. Since $S[i]$ precedes $R[j]$, $S[i]$ must appear between $C[i]$ and $C[j]$ in the log. Since T follows a two-phase locking protocol, P_i holds a c-lock for the transaction T in the period from $C[i]$ to $C[j]$. From Table 2, g(c, s) = ×. Therefore, P_i cannot obtain a s-lock, and cannot execute $S[i]$ in the period from $C[i]$ to $C[j]$, a contradiction.

iii) Suppose the dependency between a rollback transaction T and a message transaction $M = S[j]R[i]$ is cyclic. Then we have $U[i]$ precedes $R[i]$, and $S[j]$ precedes $U[j]$ in the log from Fig. 2. Since $U[i]$ precedes $R[i]$ in the log, P_i cannot execute $R[i]$ while P_i holds a u-lock for the transaction T. Suppose P_i executes $R[i]$ after P_i releases the lock. All rollback operations of T must have been executed. Therefore, process P_j must have executed $U[j]$, and informed process P_i not to receive the message, a contradiction. □

We use four types of operations and five types of conflicts to model the concurrent checkpointing and recovery problem. We have shown that the concurrent checkpointing and recovery problem can be solved by enforcing serializability on the corresponding transactions.

Deadlock Resolution

Deadlocks may occur between a checkpoint transaction and a rollback transaction. There are two approaches to resolve deadlocks:

1) Deadlock detection:

Once a deadlock is detected, we choose a victim transaction, and abort that transaction.

2) Deadlock avoidance:

   Once a C-lock of a checkpoint transaction or a U-lock of a rollback transaction cannot be granted, we abort the checkpoint transaction or the rollback transaction respectively.

2.3. Optimizations

Optimization 1: Share Checkpoints and Rollback Points

In the previous locking protocol, there can be multiple checkpoint operations and rollback operations issued to process P_i. P_i may have to keep more than one uncommitted checkpoint in the stable storage. Also P_i needs to roll back once for each rollback operation. In this section, we study an optimization in which P_i takes one uncommitted checkpoint for all concurrent checkpoint transactions, and rolls back once for all concurrent rollback transactions. Let

$$L = \cdots C_a[i] \cdots C_b[i] \cdots .$$

Suppose the log L follows the locking protocol. $C_a[i]$ and $C_b[i]$ are checkpoint operations of transactions T_a and T_b respectively.

Theorem 3. If P_i holds a C-lock during the period from $C_a[i]$ to $C_b[i]$, then the recovery line established by the operations $\{ C_a[i] \} \cup \{ C_b[j] \mid j \in \sigma(T_b), j \neq i \}$ is also consistent. That is, the dependency order between the virtual transaction $T_0 = \{ C_a[i] \} \cup \{ C_b[j] \mid j \in \sigma(T_b), j \neq i \}$, and any message transaction and any rollback transaction in the log is acyclic.

Proof: The proof is by contradiction. From Theorem 2, the dependency between the checkpoint transaction T_b and any message transaction $M = S[w]R[v]$ is acyclic, when $w \neq i$ and $v \neq i$. Therefore, the dependency between T_0 and a message transaction M is cyclic only when $M = S[i]R[j]$, or $M = S[j]R[i]$.

i) Suppose the dependency between T_0 and a message transaction $M = S[i]R[j]$ is cyclic, then

$$L = \cdots C_a[i] \cdots S[i] \cdots R[j] \cdots C_b[j] \cdots .$$

Since P_i holds a c-lock during the period from $C_a[i]$ to $C_b[i]$, and T_b follows a two-phase locking protocol, P_i holds a c-lock also during the period from $C_a[i]$ to $C_b[j]$. Therefore, P_i cannot obtain an s-lock, and cannot execute $S[i]$ during the period from $C_a[i]$ to $C_b[j]$, a contradiction.

ii) Suppose the dependency between T_0 and a message transaction $M = S[j]R[i]$ is cyclic, then

$$L = \cdots C_b[j] \cdots S[j] \cdots R[i] \cdots C_a[i] \cdots C_b[i] \cdots .$$

Then the dependency between T_b and M is cyclic, a contradiction.

The dependency between the checkpoint transaction T_b and any rollback transaction T_e is acyclic. Therefore, the order between $C_b[w]$ and $U_e[w]$ must be the same as the order between $C_b[v]$ and $U_e[v]$ in the log for any w, $v \neq i$. The dependency between T_0 and a rollback transaction T_e is cyclic, only when there exists j such that the order between $C_a[i]$ and $U_e[i]$ is not the same as the order between $C_b[j]$ and $U_e[j]$. We discuss the possible cases from iii) to vi).

iii) $U_e[i]$ and $U_e[j]$ appear between $C_a[i]$ and $C_b[j]$:

$$L = \cdots C_a[i] \cdots U_e[i] \cdots U_e[j] \cdots C_b[j] \cdots ,$$

or

$$L = \cdots C_a[i] \cdots U_e[j] \cdots U_e[i] \cdots C_b[j] \cdots .$$

Since P_i holds a c-lock during the period from $C_a[i]$ to $C_b[i]$, and T_b follows a two-phase locking protocol, P_i holds a c-lock also during the period from $C_a[i]$ to $C_b[j]$. Therefore, P_i cannot obtain a u-lock for T_e, and cannot execute $U_e[i]$ during the period from $C_a[i]$ to $C_b[j]$, a contradiction.

iv) $U_e[i]$and $U_e[j]$ appear between $C_b[j]$ and $C_a[i]$:

$$L = \cdots C_b[j] \cdots U_e[i] \cdots U_e[j] \cdots C_a[i] \cdots C_b[i] \cdots ,$$

or

$$L = \cdots C_b[j] \cdots U_e[j] \cdots U_e[i] \cdots C_a[i] \cdots C_b[i] \cdots .$$

Then the dependency between T_b and T_e is cyclic, a contradiction.

v) $C_a[i]$ and $C_b[j]$ appear between $U_e[i]$ and $U_e[j]$:

Since P_i holds a u-lock for T_e during the period from $U_e[i]$ to $U_e[j]$, P_i cannot execute $C_a[i]$ during that period, a contradiction.

vi) $C_a[i]$ and $C_b[j]$ appear between $U_e[j]$ and $U_e[i]$:

Since P_j holds a u-lock for T_e during the period from $U_e[j]$ to $U_e[i]$, P_j cannot execute $C_b[j]$ during that period, a contradiction.

□

Let

$$L = \cdots U_a[i] \cdots U_b[i] \cdots .$$

Suppose the log L follows the locking protocol. $U_a[i]$ and $U_b[i]$ are rollback operations of transactions T_a and T_b respectively.

Theorem 4. If P_i holds a u-lock during the period from $U_a[i]$ to $U_b[i]$, then the rollback line established by the operations $\{ U_a[i] \} \cup \{ U_b[j] \mid j \in \sigma(T_b), j \neq i \}$ is also consistent. That is, the dependency order between the virtual transaction $T_0 = \{ U_a[i] \} \cup \{ U_b[j] \mid j \in \sigma(T_b), j \neq i \}$, and any message transaction and any rollback transaction in the log is acyclic

Proof. The proof is by contradiction. The dependency between the rollback transaction T_b and any message transaction $M = S[w]R[v]$ is acyclic, when $w \neq i$ and $v \neq i$. Therefore, the dependency between T_0 and a message transaction M is cyclic only when $M = S[i]R[j]$, or $M = S[j]R[i]$.

i) Suppose the dependency between T_0 and a message transaction $M = S[i]R[j]$ is cyclic, then

$$L = \cdots S[i] \cdots U_a[i] \cdots U_b[j] \cdots R[j] \cdots ,$$

or

$$L = \cdots S[i] \cdots U_b[j] \cdots U_a[i] \cdots R[j] \cdots .$$

$S[i]$ is undone by $U_a[i]$. We show that $R[j]$ is not executed. $R[j]$ cannot be executed while P_j holds a u-lock for T_a. Suppose $R[j]$ is executed after P_j releases the lock. All rollback operations of T_a must have been executed. Process P_i must have executed

$U_a[i]$, and informed P_j not to receive the message, a contradiction.

ii) Suppose the dependency between T_0 and a message transaction $M = S[j]R[i]$ is cyclic, then

$$L = \cdots S[j] \cdots U_a[i] \cdots U_b[j] \cdots R[i] \cdots U_b[i] \cdots ,$$

or

$$L = \cdots S[j] \cdots U_b[j] \cdots U_a[i] \cdots R[i] \cdots U_b[i] \cdots .$$

Since P_i holds a u-lock during the period from $U_a[i]$ to $U_b[i]$, P_i cannot obtain a r-lock, and cannot execute $R[i]$ during the period from $U_a[i]$ to $U_b[i]$, a contradiction. Note that $R[i]$ cannot appear after $U_b[i]$. Otherwise, the dependency between T_b and M is cyclic.

The dependency between the rollback transaction T_b and any checkpoint transaction T_e is acyclic. Therefore, the order between $U_b[w]$ and $C_e[w]$ must be the same as the order between $U_b[v]$ and $C_e[v]$ in the log for any w, $v \neq i$. The dependency between T_0 and a checkpoint transaction T_e is cyclic, only when there exists j such that the order between $U_b[i]$ and $C_e[i]$ is not the same order between $U_b[j]$ and $C_e[j]$. We discuss the possible cases from iii) to vi).

iii) $C_e[i]$ and $C_e[j]$ appear between $U_a[i]$ and $U_b[j]$:

$$L = \cdots U_a[i] \cdots C_e[i] \cdots C_e[j] \cdots U_b[j] \cdots ,$$

or

$$L = \cdots U_a[i] \cdots C_e[j] \cdots C_e[i] \cdots U_b[j] \cdots .$$

Since P_i holds a u-lock during the period from $U_a[i]$ to $U_b[i]$, and T_b follows a two-phase locking protocol, P_i holds a u-lock also during the period from $U_a[i]$ to $U_b[j]$. Therefore, P_i cannot obtain a c-lock for T_e, and cannot execute $C_e[i]$ during the period from $U_a[i]$ to $U_b[j]$, a contradiction.

iv) $C_e[i]$ and $C_e[j]$ appear between $U_b[j]$ and $U_a[i]$:

$$L = \cdots U_b[j] \cdots C_e[i] \cdots C_e[j] \cdots U_a[i] \cdots U_b[i] \cdots ,$$

or

$$L = \cdots U_b[j] \cdots C_e[j] \cdots C_e[i] \cdots U_a[i] \cdots U_b[i] \cdots .$$

Then the dependency between T_b and T_e is cyclic, a contradiction.

v) $U_a[i]$ and $U_b[j]$ appear between $C_e[i]$ and $C_e[j]$:

Since P_i holds a c-lock for T_e during the period from $C_e[i]$ to $C_e[j]$, P_i cannot execute $U_a[i]$ during that period, a contradiction.

vi) $U_a[i]$ and $U_b[j]$ appear between $C_e[j]$ and $C_e[i]$:

Since P_j holds a c-lock for T_e during the period from $C_e[j]$ to $C_e[i]$, P_j cannot execute $U_b[j]$ during that period, a contradiction. □

The virtual transaction T_0 is a mixture of transactions T_a and T_b. However, we can view T_0 as an ordinary transaction following a two-phase locking protocol. The locking sequence of T_0 is the same as that of T_b except for $C_a[i]$ (or $U_a[i]$). T_0 obtains a c-lock (or a u-lock) for $C_a[i]$ (or for $U_a[i]$) when T_a obtains that lock. T_0 releases that lock when T_b releases a c-lock (or a u-lock) for $C_b[i]$ (or for $U_b[i]$). Therefore, theorems 3 and 4 can be recursively applied. Transaction T_a or T_b in the log L can be either an ordinary transaction or a mixed transaction containing operations drawn from other transactions. Based on the theorems, a checkpoint transaction $T = \cdots C[i] \cdots$ can be implemented more efficiently. When T issues $C[i]$ to process P_i, P_i ignores $C[i]$ if P_i already holds a c-lock on behalf of another checkpoint transaction. Similarly when a rollback transaction issues $U[i]$ to process P_i, P_i ignores $U[i]$ if P_i already holds a u-lock on behalf of another rollback transaction. However, P_i still needs to obtain a lock on behalf of T. Through this optimization, P_i keeps at most one uncommitted checkpoint in the stable storage. Also P_i rolls back once when there are concurrent rollback transactions issuing rollback operations to P_i.

Optimization 2: Establish Partial Recovery Lines and Rollback Lines

A checkpoint transaction or a rollback transaction does not have to issue operations to all processes in the system. Then the recovery line established by the checkpoint transaction $\{ C[i] \mid i \in Q, Q \subseteq \sigma_0 \}$ is called a partial recovery line. The rollback line established by the rollback transaction $\{ U[i] \mid i \in Q, Q \subseteq \sigma_0 \}$ is called a partial rollback line.

Definition 4. Let L be a log of checkpoint transactions, message transactions, and rollback transactions. Each partial recovery line and partial rollback line established by these transactions is consistent if L is D-serializable, and there is no *dangling receive* message in the augmented log $L' = L\ T_f$, for any rollback transaction $T_f = \{\ U_f[i] \mid i \in \sigma(T_f) \subseteq \sigma_0\ \}$.

This means execution of an arbitrary rollback transaction after all other transactions have terminated will not produce dangling receive messages.

Optimization 3: Never Abort a Rollback Transaction

If rollback transactions have a low abort rate, then the system can recover from failure faster. In the earlier discussion, if a rollback transaction is locked out by a checkpoint transaction, the rollback transaction is aborted or blocked. Instead, we may want to abort the checkpoint transaction if we are sure the checkpoint transaction has not committed any checkpoint operation. We can enforce some conditions in concurrent transaction processing such that a rollback transaction can always commit without being aborted.

In the next section, we have incorporated these optimizations into the algorithm. A checkpoint transaction or a rollback transaction is initiated by a coordinator. The transaction is dynamically expanded on a tree of processes. The partial recovery lines and partial rollback lines established by these transactions are consistent. A process can be a participant of several concurrent transactions. In such a case, the process makes only one checkpoint upon the first checkpoint request, or rolls back only once upon the first rollback request. Whenever a rollback transaction is locked out by a checkpoint transaction, the checkpoint transaction has not committed any checkpoint, and can always be aborted. Transactions follow the locking protocol specified earlier. We use a more restrictive lock compatibility table shown in Table 3 where g(c, r) and g(u, s) have been changed to "x". This ensures that a checkpoint transaction can always be aborted when it interferes with a rollback transaction.

Table 3. New Lock compatibility table.

g	c	s	r	u
c	○	×	×	×
s	×	×	×	×
r	×	×	×	×
u	×	×	×	○

3. The Checkpoint and Rollback Recovery Algorithm

In the algorithm, there are two types of messages: *normal messages* and *control messages*. Messages used in the execution of checkpoint and rollback transactions are called *control messages*. All others are called *normal messages*. We do not require that messages be received in the order in which they are sent. Either type of messages can get lost during transmission. Retransmission of lost messages is handled by some end-to-end communication protocols. Local checkpoints and rollback points are numbered sequentially. Suppose $[n, n+1]$ is the interval bounded by two adjacent checkpoints and/or rollback points. Then outgoing normal messages sent within the interval $[n, n+1]$ are attached the label n. For example, in Fig. 5, the labels of the messages m, l, x, y, z are 1, 2, 3, 3 and 4 respectively. We use $seqof(C_i)$ to denote the sequence number of a checkpoint C_i of P_i.

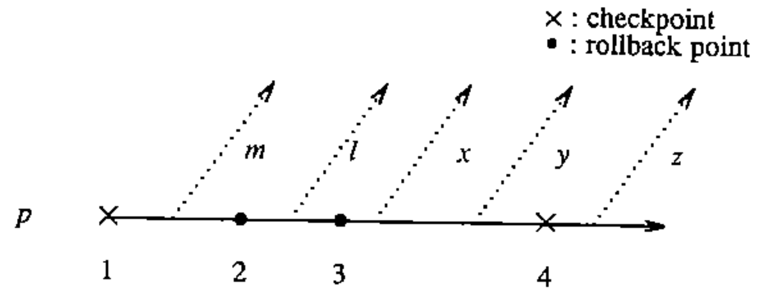

Figure 5. Numbering checkpoints and rollback points.

In the algorithm, each process saves at most two youngest checkpoints (called *oldchkpt* and *newchkpt*) in stable storage. *newchkpt* is an uncommitted checkpoint. *oldchkpt* represents the latest version of the committed checkpoint. Each participating process of a checkpoint transaction makes a new uncommitted checkpoint *newchkpt*. If the checkpoint transaction can commit, *oldchkpt* is updated with the content in *newchkpt*, and *newchkpt* is discarded. If P rolls back to *oldchkpt*, *newchkpt* (if exists) will be discarded.

Koo [KOO87] presents a checkpoint/rollback algorithm, in which each process can initiate an instance of the algorithm. The number of participants in the synchronization instance is minimal. We extend the idea such that multiple instances can be run concurrently. Each instance is modeled as a transaction. Checkpoint transactions and rollback transactions are dynamically expanded from process to process. A transaction does not necessarily include all processes in the system. We describe how checkpoint transactions and rollback transactions are expanded concurrently. The expansion order is determined by some dependency relationships among the processes. We define the dependencies of processes as follows.

Definition 5. $P_i \rightarrow P_j$ iff there exists a normal message m from P_i to P_j, and m is sent after the latest committed checkpoint of P_i, and is received after the latest committed checkpoint of P_j, and m is not undone by any rollback transactions.

We can describe the dependency relationships by a digraph where nodes represent processes, and edges represent dependencies among processes. Edges can be dynamically added or deleted due to message passing, checkpoint operations, or rollback operations. In general, the graph may be cyclic, and may not be connected.

If $P_i \rightarrow P_j$, and P_j is a participating process of a checkpoint transaction, P_i must also be a participating process of that transaction. On the other hand, if $P_i \rightarrow P_j$, and P_i is a participating process of a rollback transaction, P_j must also be a participating process of that transaction. According to this rule, a process can participate in more than one transaction. Initially, a transaction is initiated by a coordinator, and then expanded to other processes.

Edges can be added to a node dynamically because processes exchange messages, which creates new dependencies. Checkpoint transactions and rollback transactions follow the two-phase locking protocol specified earlier. Once a node holds a lock for a transaction, no edges can be added to the node until the transaction terminates. This is because the node process does not send or receive normal messages until the transaction terminates. A node will be traversed at most once when a transaction is being expanded. Therefore, the expansion of a transaction will eventually terminate. The dependency relationship is transitive. We define the transitive relation as follows.

Definition 6. $P_i \rightarrow^+ P_j$ iff $P_i \rightarrow P_j$ or there exists P_k such that $P_i \rightarrow P_k$, and $P_k \rightarrow^+ P_j$.

A checkpoint transaction or a rollback transaction initiated by the coordinator P_i is uniquely identified by the timestamp t, $t = (i,\ initiation\_time)$. We use $T(t)$ to denote the transaction with timestamp t. Each control message sent for this transaction is attached the timestamp t by the sender.

In the next three subsections, we describe the algorithm that expands checkpoint transactions and rollback transactions on a tree of processes.

3.1. Expansion of a Checkpoint Transaction

A checkpoint transaction $T(t)$ does not exist in priori. First, a process identifies itself as the coordinator of $T(t)$ (i.e., makes a checkpoint autonomously), and then expands the transaction to some other processes (i.e., issues checkpoint requests to them). The checkpoint transaction is expanded on a tree of processes. The expansion precedes as follows.

When P_j becomes a participant or the coordinator of transaction $T(t)$, P_j makes an uncommitted checkpoint C_j. Let $\max_{ij}$ be the maximum label of the normal messages sent from P_i and received within the interval $[seqof(C_j) - 1, seqof(C_j)]$. $\max_{ij}$ is set to zero if P_j receives no normal messages from P_i within that interval. P_j then regards P_i for which $\max_{ij}$ is not zero as a *potential* participant of the transaction, and sends P_i the checkpoint request message (''chkpt_req'', t, s, $\max_{ij}$). s is the index of the coordinator. Suppose when P_i receives the checkpoint request, the latest committed checkpoint of P_i is C_i. Upon the checkpoint request, if $seqof(C_i) \leq \max_{ij}$, and P_i has not been a participant or the coordinator of $T(t)$, and has not undone the the sending of the message with the label $\max_{ij}$, then P_i becomes a new participant of $T(t)$, and executes a checkpoint operation. Otherwise, P_i rejects P_j's request.

The checkpoint transaction can commit only after all participants have executed checkpoint operations. Each participant makes an uncommitted checkpoint, and then sends the coordinator a "chkpt_yes" response. The participant may expand the transaction to some other processes. The coordinator, after receiving the responses from all participants, sends a "commit" message to every participant. Upon the message, each participant commits its uncommitted checkpoint accordingly. A participant may inform the coordinator to abort the checkpoint transaction. In such a case, the coordinator sends an "abort" message to every participant.

3.2. Expansion of a Rollback Transaction

A rollback transaction $T(t)$ is also dynamically expanded. First, a process identifies itself as the coordinator of $T(t)$ (i.e., rolls back to its last committed checkpoint autonomously), and then expands the transaction to some other processes (i.e., issues rollback requests to them). The rollback transaction is expanded on a tree of processes. The expansion precedes as follows.

When P_i becomes a participant or the coordinator of transaction $T(t)$, P_i rolls back to its last committed checkpoint C_i. If P_i has ever sent any normal message to P_j since C_i was made, then P_i regards P_j as a *potential* participant of the transaction, and sends P_j the rollback request message ("roll_req", t, s, *undo_seq*). s is the index of the coordinator. *undo_seq* represents the label of the normal messages to be undone. Upon the rollback request, if P_j has received from P_i any message m with the label *undo_do*, and has not been a participant or the coordinator of $T(t)$, then P_j becomes a new participant of $T(t)$, and executes a rollback operation. Otherwise, P_j rejects P_i's request. Since the rollback request may reach P_j faster than some normal messages to P_j with the label *undo_seq*, P_i must also inform P_j to ignore such incoming normal messages later.

The rollback transaction terminates only after all participants have executed rollback operations. Each participant sends the coordinator a "roll_yes" response, and rolls back to its last committed checkpoint. The participant may expand the transaction to some other processes. The coordinator, after receiving the responses from all participants, sends a "roll_finish" message to every participant. Upon this message, each participant resumes sending and receiving normal messages. A rollback transaction will never be aborted. Whenever a checkpoint transaction interferes with a rollback transaction, the checkpoint transaction has not committed any checkpoint, and will always be aborted.

3.3. The Algorithm

The Conventions for the Algorithm

Each process P_i has a daemon process to execute the algorithm. The algorithm on each daemon process contains eight major procedures, four for checkpoint and four for rollback. A procedure is invoked by the daemon process if its corresponding invocation condition is true. The execution of each procedure is exclusive. After a procedure is executed, P_i can resume all its local computation. b1, b2, . . . , b8 represent the invocation conditions for the eight procedures respectively. For efficiency purposes, procedures roll_initiation() and roll_request_propagation() have higher priority over the other six procedures. All the other six procedures have the same priority. If more than one invocation condition are true, procedures with higher priority must be invoked first. Procedures with the same priority can be invoked in an arbitrary order. The following conventions are used in the algorithm:

1) n_i keeps track of the sequence numbers of the checkpoints and/or rollback points in P_i. Each time P_i commits a new checkpoint or rolls back, n_i is incremented by one. n_i is initialized to zero. Each outgoing normal message of P_i is attached the current value of n_i as a label.

2) Control messages sent for transaction $T(t)$ are attached the timestamp t. For example:

   send (*msg_type*, t, $\cdots$) to P_k;

   or

   receive (*msg_type*, t, $\cdots$) from P_k;

   indicates the control message is sent to (or received from) P_k for the transaction $T(t)$. It should be noted that messages are passed out by value. Thus **send**(*msg_type*, t, $\cdots$) means the content of a local variable t is copied into the second field of the outgoing message. **receive**(*msg_type*, t, $\cdots$) means the second field of the incoming message is copied into a local variable t. For simplicity, we leave out the identity of the sender from the message since it is clear from the context.

3) *chkpt_child*(i) records the indices of processes from which P_i has received normal messages during the latest period when both *chkpt_lock_set*(i) and *roll_lock_set*(i) are empty.

4) *roll_child*(i) records the indices of processes to which P_i has sent normal messages during the latest period when both *chkpt_lock_set*(i) and *roll_lock_set*(i) are empty.

5) *chkpt_participant*(i) records the indices of participants of the checkpoint transaction initiated by the coordinator P_i.

6) *roll_participant*(i) records the indices of participants of the rollback transaction initiated by the coordinator P_i.

7) A process may execute checkpoint operations for several concurrent checkpoint transactions. *chkpt_lock_set*(i) records the the timestamps of the checkpoint transactions for which P_i holds c-locks. P_i commits *newchkpt*(i) if at least one of the transactions can commit. While P_i holds c-locks, P_i does not send or receive normal messages. The set is initialized to an empty set.

8) P_i may execute rollback operations for several concurrent rollback transactions. *roll_lock_set*(i) records the timestamps of the rollback transactions for which P_i holds u-locks. While P_i holds u-locks, P_i does not send or receive normal messages. The set

is initialized to an empty set.

A process can initiate a checkpoint transaction by calling chkpt_initiation(), or initiate a rollback transaction by calling roll_initiation(). The transaction will be expanded on a tree of processes. The transactions follow the locking protocol specified earlier. Processes obtain c-locks and u-locks on behalf of transactions. P_i does not send or receive normal messages if either *chkpt_lock_set*(i) or *roll_lock_set*(i) is not empty. This causes subsequent incoming messages to be held in the channels. If an incoming message is undon by a sender, P_i must discard that message held in the channel.

Checkpoints and rollback points are shared among concurrent transactions. Therefore, P_i makes one checkpoint upon the first checkpoint request. Also P_i rolls back once upon the first rollback request.

The Algorithm

Now we outline the algorithm. P_i runs in foreground while its daemon sleeps in background. Control is switched to the daemon when some invocation condition becomes true.

```
Daemon process for checkpoint and rollback in $P_i$:
**loop**
    **sleep until** boolean condition b1 **or** b2 **or** ··· **or** b8 is true;
    **case**
        b1 : chkpt_initiation();
        b2 : chkpt_request_propagation();
        b3 : chkpt_response_collection();
        b4 : chkpt_commit/abort();
        b5 : roll_initiation();
        b6 : roll_request_propagation();
        b7 : roll_response_collection();
        b8 : roll_finish();
    **endcase**;
**endloop**;
```

The following variables are shared among these procedures: n_i, *oldchkpt*(i), *newchkpt*(i), *chkpt_child*(i), *roll_child*(i), *chkpt_participant*(i), and *roll_participant*(i), *chkpt_lock_set*(i), *roll_lock_set*(i). They are initialized to 0, *nil*, *nil*, ∅, ∅, ∅, ∅, ∅, and ∅ respectively. *nil* and ∅ are system reserved symbols. They represent a null value and an empty set respectively. *newchkpt*(i)*.state* represents a local state of P_i, and *newchkpt*(i)*.seq* is the sequence number of the checkpoint. b1, or b5 is true when some guarding variables contain certain values. After P_i makes a new checkpoint, its checkpoint timer is reset to its initial value. b2, b3, b4, b6, b7, or b8 is true when some control messages have been received. After the corresponding procedure is invoked, the received control messages are consumed, which nullifies the associated invocation condition. Next, we detail each procedure and its corresponding invocation condition.

```
Condition b1: the checkpoint timer of $P_i$ timeout and $chkpt\_participant(i) = \varnothing$ and
              $roll\_lock\_set(i) = \varnothing$
procedure chkpt_initiation();
begin
    $t := (i,\ initiation\_time)$;
    $newchkpt(i).state$ := current state of $P_i$;
    $newchkpt(i).seq := n_i + 1$;
    if $chkpt\_child(i) \neq \varnothing$ then
        $chkpt\_lock\_set(i) := \{\ t\ \}$;
        $chkpt\_participant(i) := chkpt\_child(i)$;
        send (''chkpt_req'', $t$, $i$, $\max_{ki}$) to $P_k$, for all $k \in chkpt\_child(i)$;
    else
        $oldchkpt(i) := newchkpt(i)$;
        $newchkpt(i) := nil$;
        $n_i := n_i + 1$;
    endif;
end chkpt_initiation;
```

Comments: *initiation_time* represents the real time or the logical time when the procedure starts. P_i and its daemon process for checkpoint and rollback have separate state information. $newchkpt(i).state$ saves only the state of P_i. $\max_{ki}$ is the maximum label of the messages sent from P_k and received within the interval $[newchkpt(i).seq - 1,\ newchkpt(i).seq]$. $k \in chkpt\_child(i)$ if $\max_{ki} \neq 0$. The transaction is expanded to processes P_k for all $k \in chkpt\_child(i)$. The assignment statement $oldchkpt(i) := newchkpt(i)$ copies $newchkpt(i).state$ to $oldchkpt(i).state$ and $newchkpt(i).seq$ to $oldchkpt(i).seq$. The assignment $newchkpt(i) := nil$ removes $newchkpt(i).state$ and $newchkpt(i).seq$ from the stable storage. For efficient implementation, we can copy the address pointer of $newchkpt(i)$ to that of $oldchkpt(i)$ without block transfer.

```
Condition b2: $P_i$ receives (''chkpt_req'', $t$, $s$, $\max_{ij}$) from $P_j$
procedure chkpt_request_propagation();
begin
    if $P_i$ has rolled back and undone the sending of the message with the label $\max_{ij}$ then
        send (''chkpt_abort'', $t$) to $P_s$;
        return;
    endif;
    if $t \in chkpt\_lock\_set(i)$ or $\max_{ij} < oldchkpt.seq$ then
        send (''chkpt_no'', $t$) to $P_s$;
        return;
    endif;
    atomic_send (''chkpt_yes'', $t$, $chkpt\_child(i)$) to $P_s$;
    if $chkpt\_lock\_set(i) = \emptyset$ then
    /*   make an uncommitted checkpoint.   */
        $newchkpt(i).state$ := current state of $P_i$;
        $newchkpt(i).seq := n_i + 1$;
    endif;
    send (''chkpt_req'', $t$, $s$, $\max_{ki}$) to $P_k$, for all $k \in chkpt\_child(i)$;
    /*   obtain a c-lock for $T(t)$.   */
    $chkpt\_lock\_set(i) := chkpt\_lock\_set(i) \cup \{ t \}$;
end chkpt_request_propagation;
```

Comments: P_s is the coordinator of the checkpoint transaction $T(t)$. Upon the checkpoint request, if P_i has rolled back and undone the message with the label $\max_{ij}$, the checkpoint transaction must be aborted. Thus P_i replys with the message (''chkpt_abort'', t). If P_i has already held a c-lock for $T(t)$, or the last committed checkpoint of P_i and the uncommitted checkpoint of P_j already compose a consistent recovery line, P_i replys to the coordinator with the message (''chkpt_no'', t). In all other cases, P_i replys with the message (''chkpt_yes'', t, $chkpt\_child(i)$). We require the sending and receiving of this message to be atomic. This ensures the coordinator recognizes potential participants before they reply. P_i may be a participant of several concurrent transactions. In such a case, P_i makes an uncommitted checkpoint only upon the first checkpoint request, but not upon subsequent checkpoint requests.

```
Condition b3:
    case 1) $P_i$ receives (''chkpt_yes'', $t$, $chkpt\_child(k)$) from $P_k$ or
    case 2) $P_i$ receives (''chkpt_no'', $t$) from $P_k$ or
    case 3) $P_i$ receives (''chkpt_abort'', $t$) from $P_k$
procedure chkpt_response_collection();
begin
    if case 1 then
        $chkpt\_participant(i) := chkpt\_participant(i) + chkpt\_child(k)$;
        mark a single $k$ in $chkpt\_participant(i)$;
    else /*  case 2 or case 3  */
        delete a $k$ which is unmarked from $chkpt\_participant(i)$;
    endif;
    if case 3 then
        $ABORT := true$;
    endif;
    if all members in $chkpt\_participant(i)$ have been marked then
        if $ABORT$ then
            send (''abort'', $t$) to $P_i$ and $P_k$ for all $k$ in $chkpt\_participant(i)$;
        else
            send (''commit'', $t$) to $P_i$ and $P_k$ for all $k$ in $chkpt\_participant(i)$;
        endif;
        $chkpt\_participant(i) := \emptyset$;
        $ABORT := false$;
    endif;
end chkpt_response_collection;
```

Comments: P_i will execute this procedure only if P_i is the coordinator of the checkpoint transaction $T(t)$. *chkpt_participant*(i) records all potential participants of the transaction. An element k is marked in *chkpt_participant*(i) if P_k is a true participant. An element k is deleted from *chkpt_participant*(i) if P_k rejects the rollback request. In case 1, P_k agrees to be a participant of the checkpoint transaction, and informs the coordinator of more potential participants recorded in the set *chkpt_child*(k). The ''+'' operator unions the two sets without eliminating duplicate elements. In case 2, P_k has already been a participant or the coordinator of the checkpoint transaction, and thus rejects the checkpoint request. In case 3, P_k requests the coordinator to abort the whole checkpoint transaction. Elements in *chkpt_participant*(i) will all be marked after every process that has received a checkpoint request has replied to the coordinator. P_i sends (''abort'', t) to every participant (including itself) if any of the participants cannot make a new checkpoint. Otherwise, P_i sends (''commit'', t) to every participant (including itself). The flag *ABORT* is initialized to *false*.

```
Condition b4:
     case 1) $P_i$ receives (''commit'', $t$) from $P_s$
     case 2) $P_i$ receives (''abort'', $t$) from $P_s$ and $t \in chkpt\_lock\_set(i)$
procedure chkpt_commit/abort();
begin
     $chkpt\_lock\_set(i) := chkpt\_lock\_set(i) - \{ t \}$;
     if case 1 then
     /*   commit the checkpoint.   */
          $oldchkpt(i) := newchkpt(i)$;
     endif;
     if $chkpt\_lock\_set(i) = \emptyset$ then
          $newchkpt(i) := nil$;
          $n_i := n_i + 1$;
     endif;
end chkpt_commit/abort;
```

Comments: P_s is the coordinator of the checkpoint transaction with timestamp t. P_i may be a participant of several concurrent checkpoint transactions. In such a case, *newchkpt*(i) is shared among these checkpoint transactions. *chkpt_lock_set*(i) records the timestamps of these transactions. P_i releases a c-lock upon a commit or an abort decision from the coordinator. P_i commits the new checkpoint if at least one of the coordinators can commit. Otherwise, P_i aborts the new checkpoint.

Condition b5: a transient error is detected in P_i

```
procedure roll_initiation();
begin
    $t := (i, initiation\_time)$;
    if $chkpt\_lock\_set(i) \neq \emptyset$ then
        rollback to $newchkpt(i).state$;
        return;
    endif
    if $roll\_child(i) \neq \emptyset$ and $roll\_lock\_set(i) = \emptyset$ then
        $roll\_lock\_set(i) := \{ t \}$;
        $roll\_participant(i) := roll\_child(i)$;
        send ("roll_req", $t$, $i$, $n_i$) to $P_k$, for all $k \in roll\_child(i)$;
    endif;
    rollback to $oldchkpt(i).state$;
end roll_initiation;
```

Comments: Transient errors are detected in time before P_i intends to make a new checkpoint. Thus a checkpoint never saves a state contaminated by hidden transient errors. **rollback to** *newchkpt*(i).*state* restores the state *newchkpt*(i).*state*. The transaction is expanded to process P_k for all $k \in roll\_child(i)$.

```
Condition b6: $P_i$ receives ("roll_req", $t$, $s$, $undo\_seq$) from $P_j$
procedure roll_request_propagation();
begin
    if $t \in roll\_lock\_set(i)$ or $P_i$ has not received any messages with the label $undo\_seq$
    then
        send ("roll_no", $t$) to $P_s$;
        return;
    endif;
    $newchkpt(i) := nil$;
    $chkpt\_lock\_set(i) := \emptyset$;
    atomic_send ("roll_yes", $t$, $roll\_child(i)$) to $P_s$;
    send ("roll_req", $t$, $s$, $n_i$) to $P_k$, for all $k \in roll\_child(i)$;
    if $roll\_lock\_set(i) = \emptyset$ then
        rollback to $oldchkpt(i).state$;
    endif;
    /*  obtain a u-lock for $T(t)$.  */
    $roll\_lock\_set(i) := roll\_lock\_set(i) \cup \{ t \}$;
end roll_request_propagation;
```

Comments: P_s is the coordinator of the rollback transaction $T(t)$. If P_i has already held a u-lock for $T(t)$ or P_i has not received any messages undone by the sender P_j, P_i replys to the coordinator with the message ("roll_no", t). Otherwise, P_i replys with the message ("roll_yes", t, *roll_child*(i)). We require the sending and receiving of this message to be atomic. This ensures the coordinator recognizes potential participants before they reply. P_i may be a participant of several concurrent rollback transactions. In such a case, P_i rolls back only once upon the first rollback request, but not upon subsequent rollback requests. *undo_seq* represents the label of the messages that have just been undone by the sender P_j. P_i keeps a message log for all incoming messages since last committed checkpoint. Therefore, P_i can decide if P_i has received any messages with the label *undo_seq*. Note that P_i replys to the coordinator before rolling back. This achieves a higher degree of parallelism. Even if P_i fails after replying to the coordinator, we can eventually restore its checkpointed state in the stable storage.

```
Condition b7:
    case 1) $P_i$ receives (``roll_yes'', $t$, $roll\_child(k)$) from $P_k$ or
    case 2) $P_i$ receives (``roll_no'', $t$) from $P_k$
procedure roll_response_collection();
begin
    if case 1 then
        $roll\_participant(i) := roll\_participant(i) + roll\_child(k)$;
        mark a single $k$ in $roll\_participant(i)$;
    else /* case 2 */
        delete a $k$ which is unmarked from $roll\_participant(i)$;
    endif;
    if all members in $roll\_participant(i)$ have been marked then
        send (``roll_finish'', $t$) to $P_i$ and $P_k$ for all $k$ in $roll\_participant(i)$;
        $roll\_participant(i) := \varnothing$;
    endif;
end roll_response_collection;
```

Comments: P_i will execute this procedure only if P_i is the coordinator of the rollback transaction $T(t)$. An element k is marked in *chkpt_participant*(i) if P_k is a true participant. An element k is deleted from *chkpt_participant*(i) if P_k rejects the rollback request. In case 1, P_k agrees to be a participant of the rollback transaction, and informs the coordinator of more potential participants recorded in the set *roll_child*(k). The ``+'' operator unions the two sets without eliminating duplicate elements. In case 2, P_k has already been a participant or the coordinator of the rollback transaction, and thus rejects the rollback request. Elements in *roll_participant*(i) will all be marked after every process that has received a rollback request has replied to the coordinator. After receiving responses from all participants of the rollback transaction, P_i sends (``roll_finish'', t) to every participant (including itself).

```
Condition b8: $P_i$ receives (''roll_finish'', $t$) from $P_s$
procedure roll_finish();
begin
    $roll\_lock\_set(i) := roll\_lock\_set(i) - \{ t \}$;
    if $roll\_lock\_set = \emptyset$ then
        $n_i := n_i + 1$;
    endif;
end roll_finish;
```

Comments: P_s is the coordinator of the rollback transaction with $T(t)$. P_i may be a participant of several concurrent rollback transactions. $roll\_lock\_set(i)$ records the timestamps of these transaction's. P_i releases a u-lock upon a ''roll_finish'' message from the coordinator.

3.4. Illustrative Examples

This subsection gives two examples. Example 1 shows the execution of a single checkpoint transaction. Example 2 shows concurrent execution of two checkpoint transactions. Each figure shows only normal message passing but not control message passing.

Example 1 (see Fig. 6).

As mentioned earlier, checkpoints in each process are numbered in increasing order. Based on the intervals, the labels of the normal messages x, l, m are all 1. The checkpoints α_2, α_3 and α_4 in Fig. 6. are created by the transaction. First, P_2 initiates the checkpoint transaction $T(t)$ by making α_2 autonomously, and sending P_3 the message (''chkpt_req'', t, 2, $\max_{32}$), where $\max_{32}$ has the value 1. Upon the message, P_3 makes a new checkpoint α_3. Then P_3 replys with (''chkpt_yes'', t, $\{ 4 \}$), where 4 is the index of the potential checkpoint participant P_4, and sends P_4 the message (''chkpt_req'', t, 2, $\max_{43}$), where $\max_{43}$ has the value 1. Upon the message, P_4 makes a new checkpoint α_4, and replys with (''chkpt_yes'', t, $\{ \}$). After all checkpoint operations are executed, P_2 sends (''commit'', t) to P_2, P_3 and P_4. α_2, α_3, α_4 commit, and compose a new consistent recovery line. λ_2, λ_3, λ_4 are discarded. □

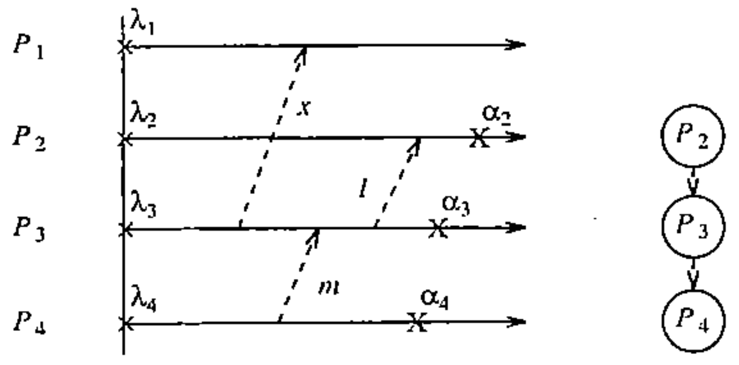

Figure 6. The process timing diagram and the checkpoint spanning tree for example 1.

Example 2 (see Fig. 7).

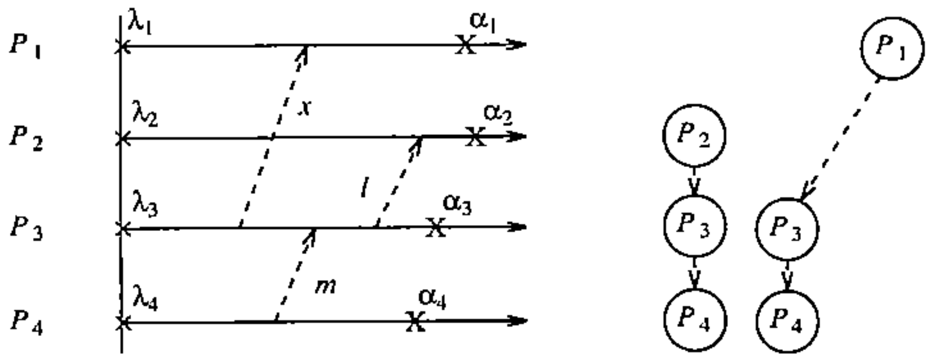

Figure 7. The process timing diagram and the checkpoint spanning trees for example 2.

In this example, two checkpoint transactions run concurrently. As in Example 1, P_2 initiates one checkpoint transaction $T(t)$ by making checkpoint α_2 autonomously. P_1 initiates the other checkpoint transaction $T(t')$ by making checkpoint α_1 autonomously, and sends P_3 the message (''chkpt_req'', t', 1, $\max_{31}$), where $\max_{31}$ has the value 1. P_2 sends P_3 (''chkpt_req'', t, 2, $\max_{32}$). Then P_3 makes α_3 upon the request of P_2 or of P_1 whichever comes first. Both checkpoint requests are propagated from P_3 to P_4. Therefore, P_4 receives two checkpoint requests originating from P_2 and P_1. In this case, the uncommitted checkpoint α_3 is shared between the two checkpoint transactions. P_3 can commit the shared checkpoint α_3 if at least one of the two tranmsactions can commit. Eventually $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ commit, and compose a new consistent recovery line. The previous checkpoints $\lambda_1, \lambda_2, \lambda_3, \lambda_4$ are discarded. Since processes propagates checkpoint requests for each transaction, the two checkpoint transactions will not block each other. Each transaction can eventually be expanded. □

4. Correctness Proof

This section shows the correctness of the algorithm, and derives some properties of the algorithm.

Theorem 5 Each checkpoint transaction will eventually terminate (either committed or aborted). Also each rollback transaction will eventually terminate.

Proof: Each transactions is expanded hierarchically. The coordinator recognizes each potential participant before it replys. Upon a checkpoint request, a process will always reply to the coordinator with either a "chkpt_yes" or a "chkpt_no" or a "chkpt_abort" message. Similarly, upon a rollback request, a process will always respond with either a "roll_yes" or a "roll_no" message. Thus, once a process identifies itself as the coordinator, it will eventually receive responses from all the participants of the transaction. Since the number of processes in the system is finite, each transaction has a finite number of participants. Each participant expands the transaction at most once. Each transaction will eventually terminate. □

Theorem 6. Each partial recovery line and each partial rollback line established by a checkpoint transaction or a rollback transaction is consistent.

Proof:

i) When a single checkpoint transaction or a single rollback transaction is executed without interference:

Since transactions follow the locking protocol specified earlier, the dependency between any message transaction and the transaction is acyclic. Also according to the algorithm, when $P_i \rightarrow P_j$, then if P_j is a process of a checkpoint transaction, then P_i is also a process of the transaction. On the other hand, if P_i is a process of a rollback transaction, then P_j is also a process of the transaction. Therefore, there is no *dangling receive* message.

ii) When checkpoint transactions T_a and T_b are executed concurrently:

Suppose P_i receives checkpoint requests from T_a and T_b, and $chkpt\_lock\_set(i) \neq \emptyset$ upon a checkpoint request from T_b. Then P_i makes a checkpoint for T_a, but not for T_b. This does not affect the consistency from Theorem 3, because P_i does not send or receive normal messages when $chkpt\_lock\_set(i)$ is not empty.

iii) When rollback transactions T_a and T_b are executed concurrently:

Suppose P_i receives rollback requests from T_a and T_b, and $roll\_lock\_set(i) \neq \emptyset$ upon a rollback request from T_b. Then P_i rolls back for T_a, but not for T_b. This does not affect the consistency from Theorem 4, because P_i does not send or receive normal messages when $roll\_lock\_set(i)$ is not empty.

iv) When a checkpoint transaction and a rollback transaction are executed concurrently:

iv)-A When a checkpoint request is issued to P_i from P_j, and P_i has undone the sending of a message to P_j:

P_i will inform the checkpoint coordinator to abort the checkpoint transaction.

iv)-B When a rollback request is issued to P_i from P_j, and $chkpt\_lock\_set(i)$ is not empty:

Case 1 P_i is the coordinator of the rollback transaction, i.e., $i = j$:

P_i rolls back to the last uncommitted checkpoint, and the rollback transaction is completed. In such a case, the rollback transaction has only one rollback operation.

Case 2 P_i has not receive any messages to be undone by P_j:

P_i ignores this rollback request.

Case 3 Otherwise:

P_i becomes a participant of the rollback transaction. P_i rolls back to the last committed checkpoint, and discards the uncommitted checkpoint. We next show that the checkpoint transaction for which P_i is a participant has not committed any checkpoint, and will be aborted.

For any P_i, P_i does not receive normal messages when *chkpt_lock_set*(i) is not empty. Therefore, if P_i becomes a rollback participant, and discards its uncommitted checkpoint, there must exist a normal message received before that checkpoint is made. Suppose P_c is the coordinator of the checkpoint transaction, and P_r the coordinator of the rollback transaction. P_i participates in both the checkpoint transaction and the rollback transaction. According to the expansion rule of checkpoint transactions and rollback transactions, we have $P_r \rightarrow^+ P_i$ and $P_i \rightarrow^+ P_c$, where $\rightarrow^+$ is defined in Definition 6. Hence, $P_r \rightarrow^+ P_c$. There must exist P_k and P_h on the path from P_r to P_c such that i) $P_k \rightarrow Ph$; ii) P_k participates in the rollback transaction before the checkpoint transaction; and iii) P_h participates in the checkpoint transaction before the rollback transaction. Then P_k will inform P_c to abort the checkpoint transaction upon a checkpoint request from P_h. □

Suppose P_1 is the coordinator of a checkpoint transaction, and $P_2, \ldots, P_k$ are the participants. Let $\mathrm{C} = \{ C_1, C_2, \ldots, C_k \}$ be the recovery line established by the checkpoint transaction. The next theorem states the *minimality* of the checkpoint transaction.

Theorem 7. Each checkpoint transaction has the minimal number of participants. That is, the recovery line $\overline{\mathrm{C}} = \mathrm{C} - \{ C_i \} \cup \{ \overline{C_i} \}$ is inconsistent for any $2 \le i \le k$, where $\overline{C_i}$ is the last committed checkpoint of P_i made before C_i

Proof: By Theorem 6, C is a consistent recovery line. Since P_i, $2 \le i \le k$, is not the root of the checkpoint spanning tree, P_i must have a parent P_j in the tree, $1 \le j \le k$. From definition 5, $P_i \rightarrow P_j$. That is, there exists a normal message m sent by P_i after $\overline{C_i}$ and received by P_j before C_j. Then the order between the message transaction of m and the checkpoint transaction that establishes the recovery line $\overline{\mathrm{C}}$ is cyclic. □

Suppose P_1 is the coordinator of a rollback transaction, and $P_2, \ldots, P_k$ are the participants. Let $\mathrm{U} = \{ U_1, U_2, \cdots, U_k \}$ be the rollback line established by the rollback transaction. The next theorem states the *minimality* of the rollback transaction.

Theorem 8. Each rollback transaction has the minimal number of participants. That is, the rollback line $\overline{\mathrm{U}} = \mathrm{U} - \{ U_i \}$ is inconsistent for any $2 \le i \le k$.

Proof: Since $P_j, 2 \le j \le k$, is not the root of the rollback spanning tree, P_j must have a parent P_i in the tree, $1 \le i \le k$. From Definition 5, $P_i \rightarrow P_j$. P_j must have received a message m from P_i upon the rollback request from P_i. If P_j does not roll back, the actions triggered by the message at P_j are not undone. Therefore, this recover line $\overline{\mathrm{U}}$ produces a dangling receive message. □

5. Comparison with Related Work

Several distributed checkpointing and recovery mechanisms can be found in [BARI83, KOO87, TAMI84]. Their distinguishing features are as follows:

Barigazzi-Strigini algorithm [BARI83]:

- The sending and receiving of a message is atomic, which is more restrictive than FIFO channels. Under this constraint, sending a message will block the operations of the sender until the message is received.
- A process suspends all normal operations while the process is a participating process of a checkpoint instance or a rollback instance.

Tamir-Séquin algorithm [TAMI84]:

- All the processes in the system need to take checkpoints or roll back together.
- A process suspends all normal operations while the process is a participating process of a checkpoint instance or a rollback instance.

Koo-Toueg algorithm [KOO87]:

- Messages are assumed to be transmitted in First-in-First-out order.
- Only processes that have exchanged messages since their last checkpoints need to take checkpoints or roll back together. Concurrent execution of multiple checkpoint instances

or rollback instances is not allowed. Each rollback process rolls back only after every other process in the instance agrees to roll back.

- A process cannot send normal messages while the process is a participating process of a checkpoint instance or a rollback instance.
- Multiple process failures usually block the algorithm. The algorithm needs to wait until the failed processes recover. Also Network partitioning is not considered.

Leu-Bhargava algorithm:

- Normal messages and most of the control messages can be transmitted in any order.
- Only processes that have exchanged messages since their last checkpoints need to take checkpoints or roll back together. Concurrent execution of checkpoint transactions or rollback transactions is allowed. Each rollback transaction always commits without being aborted or blocked. Each rollback process rolls back without waiting until other participants agree to roll back. Once it rolls back, it can immediately continue all its operations except sending and receiving normal messages.
- A process cannot send or receive normal messages while the process is a participating process of a checkpoint transaction or a rollback transaction.
- Blocking possibility of checkpoint transactions due to multiple process failures and network partitioning is reduced. Blocking of a rollback transactions can always be resolved.

5.1. Advantages of Non-FIFO Channels

Our algorithm allows normal messages to be transmitted in any order, not necessarily in First-in-First-out order. Due to the message delay or loss of messages, it is more expensive to implement FIFO channels than non-FIFO channels. Some applications prefer non-FIFO semantics to FIFO semantics. One example is distributed discrete event simulation [JEFF82]. Second example is that a sender may set up a "mailbox" storing all the outgoing messages, which are subsequently "pulled out" by the receivers based on some priority, not necessarily in the order in which they are produced by the sender. Some messages may not be inspected at all. Third example is that two processes may be connected by more than one logical FIFO channel for different purposes. Then the overall effect will make these FIFO channels look like one single non-FIFO channel.

6. Performance Evaluation

Distributed checkpointing and recovery has been studied in various literature. Not much research has been done on performance analysis. We have implemented the algorithm on a network of SUN–3[1] workstations, and have collected performance data. This section analyzes the performance of the algorithm.

6.1. Experimental Design

6.1.1. Experimental Procedures

This experiment is done in the RAID system [BHAR88c], which runs on SUN-3 workstations connected by Ethernet. Each experimental scenario performs the following steps:

a) Execute normal processes which send normal messages to one another.

b) Invoke a checkpoint starter or a rollback starter which sends a special message to designated processes. A process that receives this message initiates a checkpoint transaction or a rollback transaction respectively. Before the transaction terminates, the process does not send or receive normal messages.

c) Run a special command to stop the experiment.

1. SUN-3 is a trademark of Sun Microsystems, Inc.

6.1.2. Measured Data

In the experiments we measure the performance of the coordinator and participants separately during the execution of a checkpoint transaction or a rollback transaction. In the following discussion, "transaction" means either a checkpoint transaction or a rollback transaction. Each transaction is executed by two to eight processes. We have measured *elapsed time* and *cpu usage* during the execution of transactions. Elapsed time is the total time a process spends during the execution of a transaction. This period starts from the time when the process receives a checkpoint request or a rollback request until it receives a commit or an abort decision of the coordinator. Elapsed time contains three components: a) time to take a single checkpoint or roll back to the last checkpoint, b) cpu time, and c) idle time waiting for messages.

Single checkpoint delay is the time to write the image of a process into the disk, while single rollback delay is the time to read the image of a process from the disk. For processes ranging from 4K bytes to 48K bytes, the checkpoint and rollback were measured to take time ranging from 89 ms (milliseconds) to 496 ms (milliseconds). Reading memory images or restoring images can be done by a back-end processor, which does not consume cpu resource. So we simulate this approach by requiring a process to sleep for a period of time while taking a single checkpoint or rolling back.

Processes coordinate with one another to synchronize their checkpoint operations and rollback operations. Each process spends cpu time sending, receiving, and processing synchronization messages. CPU time contains two components: a) communication cost, and b) computation cost for the execution of the algorithm.

A process can be a participant of two or three concurrent transactions. We have measured elapsed times of processes during the execution of a single transaction and that of concurrent transactions. The following notation is used in the analysis.

Notation.

$elap_c$	elapsed time of the coordinator of a single transaction.
$elap_p$	elapsed time of a participant of a single transaction.
$elap_{cp}$	elapsed time of a process that is the coordinator of one transaction and also a participant of the other transaction. In this case, two transactions are executed concurrently.
$elap_{pp}$	elapsed time of a process that is a common participant of two transactions.
$elap_{cpp}$	elapsed time of a process that is the coordinator of one transaction and also a common participant of the other two transactions. In this case, three transactions are executed concurrently.
$elap_{ppp}$	elapsed time of a process that is a common participant of three transactions.

Similarly, CPU_c, CPU_p, CPU_{cp}, CPU_{pp}, CPU_{cpp}, and CPU_{ppp} denote the corresponding cpu times of processes during the execution of a single transaction and that of concurrent transactions.

6.2. Performance in Concurrent Execution

In the synchronous checkpointing approach, it is likely to have more than one coordinator at a time. Each coordinator initiates a transaction. Different transactions may interfere. If concurrent execution is not allowed, one transaction would have to wait for the other to finish. The delay may be accumulated as there are more transactions running. For example, suppose transactions $T_1, T_2, \cdots, T_{k+1}$ are initiated at the same time, and they have the same number of participants. T_1 takes time x to finish. Each process takes Δy to make a checkpoint, and propagate the checkpoint request. Process P_k is a common participant of transaction T_k and T_{k+1}. P_k can execute a checkpoint operation for T_{k+1} only after T_k has terminated. Therefore, transaction T_{k+1} will finish Δy time later than T_k. The finish time of transaction T_k is $x + (k - 1)\Delta y$.

If each common participant can execute checkpoint operations concurrently for two transactions, the two transactions can precede simultaneously. The common participant spends $2\Delta y$ executing two checkpoint operations. The finish time of each transaction will be the same.

Table 4. Elapsed times (in milliseconds).

number of processes of each transaction: 5
Each process is on a different site.
single checkpoint/rollback delay: 251 ms

		elapsed time of checkpoint process	elapsed time of rollback process
single transaction	$elap_c$	559	301
	$elap_p$	322	303
two concurrent transactions	$elap_{cp}$	588	354
	$elap_{pp}$	360	359
three concurrent transactions	$elap_{cpp}$	617	400
	$elap_{ppp}$	389	405

We have done some optimization in concurrent execution. Concurrent transactions can share checkpoints or rollback points. Therefore, each common participant spends less than $2\Delta y$ executing checkpoint operations for the two transactions. We have studied experimentally the effect of sharing checkpoints and rollback points on the elapsed time. Table 4 shows the elapsed time of the coordinator and that of a participant during the execution of a single transaction and that of concurrent transactions. Each process is of 20K bytes, and spends 251 ms making a single checkpoint or rolling back. In the experiment of concurrent transaction processing, each transaction is executed by the same five processes. A process can be a coordinator of at most one transaction, but can be a common participant of two or three transactions.

Due to the sharing of the checkpoints and rollback points, the elapsed time of a process that executes operations for concurrent transactions will be smaller. From Tables 4, we found the following relationships:

$$elap_{cp} \approx elap_c + elap_p - d$$

$$elap_{pp} \approx elap_p + elap_p - d$$

$$elap_{cpp} \approx elap_{cp} + elap_p - d$$

$$elap_{ppp} \approx elap_{pp} + elap_p - d$$

$d = 251$ ms, which is the time to take a single checkpoint or to roll back.

Data on the left side of $\approx$ are measured experimentally. The expressions on the right side represent expected values. When a process executes checkpoint operations for two concurrent checkpoint transactions, the process makes a single checkpoint instead of two. Therefore, $elap_{cp}$ and $elap_{pp}$ can be expected to be d milliseconds shorter than if the process makes two checkpoints. When a process executes operations for two concurrent rollback transactions, the process rolls back once instead of twice. Therefore, $elap_{cp}$ and $elap_{pp}$ can be expected to be d milliseconds shorter than if the process rolls back twice. For checkpoint processes, the measured data are even smaller than expected, because processes tend to utilize CPU idle time more efficiently in concurrent transaction processing.

6.3. Performance in Rollback-Recovery

In our algorithm, rollback transactions can always commit without being aborted. A process rolls back without waiting until other participants agree to roll back. Therefore, the process can recover from transient errors faster. The period of time during which normal operations are suspended is about the same as the time for the process to roll back.

If processes roll back only after other participants agree to do so, their normal operations will be suspended for a longer period of time. This period will include the time to synchronize with other processes, and the time to await decisions from the coordinator. Table

5 shows the elapsed time of a process during the execution of a rollback transaction with respect to three different rollback delays in a single site environment. This period of time is about 1.4 to 4 times the single rollback delay.

Table 5. Elapsed times (in milliseconds).

number of processes of the rollback transaction: 8
All the eight processes are on the same site.
single rollback delays: 89 ms, 251 ms, 496 ms

	rollback delay		
	89	251	496
coordinator	363	472	719
participant	316	438	684

Table 6. Maximum number of synchronization messages.

number of processes of the transaction: 5
There are one coordinator and four participants.

	number of messages sent	number of messages received	total
coordinator	13	26	39
participant	8	5	13
total	$13 + 4 \times 8 = 45$	$26 + 4 \times 5 = 46$	

6.4. Overhead of the Algorithm

Processes synchronize their checkpoint operations and rollback operations by sending messages. The total message overhead is no worse than other synchronous checkpointing algorithms presented in [BARI83, KOO87]. In our algorithm, the message overhead is not uniformly distributed. The total number of messages sent and received by the coordinator is in the order $O(n^2)$. The total number of messages sent and received by a participant is $O(n)$, where n is the number of the processes of the transaction. It costs CPU time to process the messages. We study the worst case when the maximum number of synchronization messages are sent among the processes. Table 6 shows the maximum number of synchronization messages a process sends and receives in executing a single transaction. This number only depends on the number of participants of the transaction. The maximum number of messages used in the execution of a checkpoint transaction is the same as that of a rollback transaction if they contain the same number of participants.

We have measured the CPU time a process spends during the execution of a single transaction and that of concurrent transactions. Each transaction is executed by the same five processes. Table 7 shows the CPU costs when all five processes are on a single site. Table 8 shows the CPU costs when each process is on a different site. We have the following observations:

- The CPU cost of a checkpoint transaction is about the same as that of a rollback transaction with the same number of participants.
- When each process is on a different site, the CPU cost is about 2.2 times that when all five processes are on a single site. This is because remote communication is more expensive than local communication.
- CPU cost contains two components: a) *communication cost*, and b) *computation cost* for the execution of the algorithm. Communication cost is about 45% of the total CPU cost in a single site environment, and increases to 75% when each process is on a different site.

Table 7. CPU costs (in milliseconds).

number of processes of each transaction: 5
All the five processes are on the same site.

		CPU cost of checkpoint process	CPU cost of rollback process
single transaction	CPU_c	47.0	48.8
	CPU_p	18.3	18.6
two concurrent transactions	CPU_{cp}	67.1	63.1
	CPU_{pp}	36.3	38.6
three concurrent transactions	CPU_{cpp}	80.9	79.9
	CPU_{ppp}	55.5	57.9

Table 8. CPU costs (in milliseconds).

number of processes of each transaction: 5
Each process is on a different site.

		CPU cost of checkpoint process	CPU cost of rollback process
single transaction	CPU_c	102.0	99.1
	CPU_p	41.6	41.0
two concurrent transactions	CPU_{cp}	135.1	136.7
	CPU_{pp}	87.7	86.3
three concurrent transactions	CPU_{cpp}	173.0	174.9
	CPU_{ppp}	138.8	135.6

6.5. Effect of Multiprogramming Level on Elapsed Time

We study how multiprogramming level may affect the elapsed time of a process during the execution of a transaction. When all processes of a transaction are on the same site, the elapsed time will be the longest. When there is one process per site, the elapsed time will be the shortest. Table 9 shows the elapsed time of processes at different multiprogramming levels. There are four processes in a transaction. The checkpoint delay and rollback delay are 251 ms. We choose 1, 2, 4 as the multiprogramming levels. That means, each site has one process, each site has two processes, and one site has all the four processes respectively.

multiprogramming levels: 1, 2, 4 (# of processes per site)
number of processes of the transaction: 4
single checkpoint/rollback delay: 251 ms

Table 9. Elapsed times at different multiprogramming levels.

		multiprogramming level		
		1	2	4
checkpoint transaction	coordinator	542	587	645
	participant	308	344	395
rollback transaction	coordinator	299	327	374
	participant	291	302	328

From Tables 9, we have the observations:

- Multiprogramming level affects the coordinator more than a participant. This is because the coordinator has higher CPU cost, which incurs more time sharing delay. Therefore, as the multiprogramming level increases, the elapsed time of the coordinator increases faster than that of a participant.

- Multiprogramming level affects processes of a checkpoint transaction more than those of a rollback transaction. This is because the two-phase commit protocol used in the execution of a rollback transaction has a higher degree of parallelism than that of a checkpoint transaction. Upon a rollback request, a process, replys to the coordinator, and propagates the rollback request before it rolls back to its last checkpoint. The coordinator can process some messages while other participants are rolling back, which does not consume the cpu resource. Therefore, rollback processes can utilize CPU idle time more efficiently than checkpoint processes. On the other hand, upon a checkpoint request, a process replys to the coordinator, and propagates the checkpoint request only after it has made a checkpoint. From Table 9, the elapsed time of a checkpoint process increases faster than that of a rollback process.

6.6. Comparison with the Independent Checkpointing Algorithm and Concluding Remarks

We compare the performance with that of the independent checkpointing algorithm [BHAR88b]. In this algorithm, processes take checkpoints independently without synchronization. Since the last checkpoints of processes may not compose a consistent recovery line, a process may not discard old checkpoints when a new checkpoint is generated. When a coordinator process decides to roll back, it initiates a rollback instance. The coordinator collects checkpoint information from all other processes in the system, and determines a consistent recovery line to which the other processes need to roll back.

6.6.1. The Independent Checkpointing Algorithm

- This algorithm is more efficient in checkpointing because no synchronization is needed. However, it is less efficient in rollback-recovery. During rollback-recovery, a coordinator needs to collect information from all other processes in the system. It may determine a recovery line composed by very early checkpoints. Then processes have to roll back to the very early checkpoints. Based on the experimental results [BHAR88a], the rollback distance depends on the message exchange pattern among the processes. When processes exchange messages very frequently, a rollback recovery is likely to cause all processes to roll back to very early checkpoints. To cope with this kind of problem, the checkpoint intervals of processes should be made adaptable. Checkpoint intervals should be made smaller when failure rate is high or when message exchange rate is high. Therefore, this algorithm will have a high overhead when message exchange rate is high, because each process either needs to take checkpoints more frequently or tends to roll back to a very earlier checkpoint. To discard old checkpoints, processes also need to take checkpoints more frequently. In such a way, new checkpoints more likely compose a consistent recovery line.

- The size of synchronization messages is in a quadratic order of all the processes in the system. The total number of messages is in a linear order of all the processes in the system. In the experiment [BHAR88a], each process keeps 4 to 10 checkpoints in the stable storage. Message size also depends on the number of checkpoints kept by each process.
- This algorithm does not allow concurrent execution. Rollback-recovery is slower compared to the synchronous algorithm.

6.6.2. The Synchronous Algorithm

- Only processes that have exchanged messages since their last checkpoints need to coordinate with one another. The algorithm will perform better when the processes have a bigger image size. For smaller processes, we can group them together as a checkpoint unit or a rollback unit. In such a way, we can reduce the message overhead at the expense of increasing single checkpoint cost and single rollback cost.
- The last committed checkpoints of all processes always compose a consistent recovery line. The rollback distance is independent of the message exchange rate. We can determine an optimal checkpoint interval based on the failure rate.

- The size of synchronization messages is in a linear order of the number of processes of the transaction. The total number of messages is in a quadratic order of the number of processes in the transaction. Processes spend less time processing each message. In our experiment, the message size is only 22 bytes.
- The synchronous algorithm allows concurrent execution. We have shown that concurrent execution reduces the response time of checkpoint transactions and rollback transactions, and improves the performance of rollback recovery.

7. Resolving Blocking due to Process Failures and Network Partitioning

While checkpoint transactions or rollback transactions are run on some processes, some process may fail and block other processes. We adopt the following assumptions about failures. a) Process failures are clean; that is, a process fails and stops without sending any forged control messages. b) Process failures do not affect the stable storage [LAMPS79]. Thus a recovering process can always restore its last checkpointed state. c) Operational processes are informed of process failures in finite time. The mechanisms monitoring the process status information have been studied in [BHAR86, HAMM80, WALT82]. d) After a process notices a process failure, it discards all subsequent normal messages from the failed process. These messages are in transit when the process fails. e) A recovering process can always collect all its lost incoming control messages either from its message spoolers [HAMM80] or from some other processes. These messages addressed to the failed process were redirected to its message spoolers. Messages can be replicated on multiple spoolers to enhance reliability. If all message spoolers fail, the recovering process must inquire its cooperating processes of the same transaction. So, the recovering process can catch up with its cooperating processes.

In our algorithm, a rollback operation can always commit. Under the assumptions about failures, blocking of rollback transactions can always be resolved. However, blocking of a checkpoint transaction may not be resolved. If the checkpoint transaction has not committed any checkpoint operation, the checkpoint transaction should be completely aborted. Otherwise, operational processes must commit or abort their uncommitted checkpoints based on the decision of the coordinator. However, if the operational processes cannot tell if the coordinator has made a decision in case of cascading failures, the transaction is blocked until failed processes recover. However, operational processes are still allowed to continue their own operations except sending and receiving normal messages.

8. An Algorithm that Allows Multiple Checkpoints in Stable Storage

In the earlier algorithm, each process keeps in stable storage exactly one committed checkpoint and at most one uncommitted checkpoint. In some applications, it is necessary to allow a process to make new checkpoints before its previous checkpoints commit. Also, a process may have to keep previously committed checkpoints for rollback purposes.

The Applications

We study the following applications where a process may keep multiple checkpoints in stable storage:

1) *Application 1*: In the original algorithm, when a checkpoint transaction is blocked due to process failures or network partitioning, participating processes cannot commit their checkpoints or initiate new checkpoint transactions. An alternative approach is to allow participating processes to initiate new checkpoint transactions. In such a case, the checkpoints made by the old checkpoint transaction are kept uncommitted.

2) *Application 2*: In this case, the system may not detect transient errors immediately. It is possible that a checkpoint may save a state containing transient errors. Then, rolling back to the youngest committed checkpoint is not sufficient to eliminate the transient errors. Thus, each process may keep in stable storage previously committed checkpoints. From time to time, a process starts verifying its committed checkpoints in stable storage. If they contain no transient errors, some of them can be discarded.

In applications 1, P_i keeps exactly one committed checkpoint and multiple uncommitted checkpoints in stable storage. In application 2, P_i keeps multiple committed checkpoints and at most one uncommitted checkpoint in stable storage. To deal with a combination of the two, P_i needs to keep multiple committed checkpoints and multiple uncommitted checkpoints in stable storage. We next modify the original algorithm. We describe how checkpoint transactions and rollback transactions are expanded on a tree of processes, when each process may keep multiple checkpoints in the stable storage.

The Modified Algorithm

Suppose P_i keeps in stable storage the checkpoints $oldchkpt_a(i)$, $oldchkpt_{a+1}(i)$, . . . , $oldchkpt_\alpha(i)$, $newchkpt_b(i)$, $newchkpt_{b+1}(i)$, . . . , $newchkpt_\beta(i)$. The checkpoints record the states of P_i in an increasing order indicated by the subscripts. Each $oldchkpt(i)$ is a committed checkpoint, and each $newchkpt(i)$ is an uncommitted checkpoint. Each uncommitted checkpoint can be shared among multiple checkpoint transactions. We use $chkpt\_lock\_set(i)$ to record the timestamps of the checkpoint transactions for which P_i holds c-locks. $roll\_lock\_set(i)$ records the timestamps of the rollback transactions for which P_i holds u-locks. P_i does not send or receive normal messages if either $chkpt\_lock\_set(i)$ or $roll\_lock\_set(i)$ is not empty. However, if checkpoint transactions for which P_i holds c-locks are all blocked due to process failures, P_i is still allowed to send and receive normal messages. Let $\max_{ijg}$ denote the maximum label of the messages sent from P_i and received within the interval $[newchkpt_g(j).seq - 1,\ newchkpt_g(j).seq]$. Each checkpoint transaction and each rollback transaction is dynamically expanded on a tree of processes. The coordinator commits the transaction after every participant has executed the checkpoint or rollback operation. We classify uncommitted checkpoints into *marked* ones and *unmarked* ones. Initially, any uncommitted checkpoint is unmarked. Suppose P_j issues a checkpoint request to P_i for transaction $T(t)$. P_s is the coordinator. Upon the request (''chkpt_req'', t, s, $\max_{ijg}$) from P_j, P_i may have the following cases for any previous outgoing message m to P_j with the label $\max_{ijg}$:

Case 1 P_i has undone the sending of the message with the label $\max_{ijg}$:

P_i requests the coordinator to abort $T(t)$.

Case 2 P_i sends m within the interval $[newchkpt_h(i).seq - 1,\ newchkpt_h(i).seq]$, i.e., $\max_{ijg} = newchkpt_h(i).seq - 1$, and $newchkpt_h(i)$ is not marked:

P_i does not make another checkpoint upon the request. Let $\max_{kih}$ be the maximum label of the messages sent from P_k and received within the interval $[newchkpt_h(i).seq - 1,\ newchkpt_h(i).seq]$. P_i then regards P_k for which $\max_{kih}$ is not zero as a potential participant of the transaction $T(t)$, and sends out the checkpoint request (''chkpt_req'', t, s, $\max_{kih}$) to P_k.

Case 3 P_i sends m within the interval $[n_i, \infty)$, i.e., $\max_{ijg} = n_i$:

P_i must make a new checkpoint $newchkpt_{b+1}(i)$, where $newchkpt_{b+1}(i).seq$ is set to $n_i + 1$. Let $\max_{ki(b+1)}$ be the maximum label of the messages sent from P_k and received within the interval $[newchkpt_{b+1}(i).seq - 1,\ newchkpt_{b+1}(i).seq]$. P_i then regards P_k for which $\max_{ki(b+1)}$ is not zero as a potential participant of $T(t)$, and sends out the checkpoint request. (''chkpt_req'', t, s, $\max_{ki(b+1)}$) to P_k.

Case 4 Otherwise:

P_i rejects the request from P_j.

P_i can be a participant of several checkpoint transactions, which share the checkpoint $newchkpt_h(i)$. If at least one of the checkpoint transactions can commit, P_i marks $newchkpt_h(i)$. P_i releases a c-lock upon a final decision from the coordinator. When P_i releases all c-locks and commits its checkpoint, P_i increments n_i by 1.

When $newchkpt_b(i),\ newchkpt_{b+1}(i), \ldots,\ newchkpt_h(i)$ are all marked, $newchkpt_h(i)$ can now commit. That is, a new committed checkpoint $oldchkpt_{\alpha+1}(i)$ is created with the content in $newchkpt_h(i)$, and $newchkpt_b(i),\ newchkpt_{b+1}(i), \ldots,\ newchkpt_h(i)$ are discarded.

Operations for rollback also need modification: When a transient error is detected in P_j, P_j initiates a rollback transaction by rolling back to the last committed checkpoint. The transaction $T(t)$ is dynamically expanded on a tree of processes. Suppose P_j issues a checkpoint request to P_i for transaction $T(t)$. P_s is the coordinator. Upon the rollback request ("roll_req", t, s, *undo_seq*) from P_j, P_i may have the following cases for all incoming normal messages that are sent from P_j and have the label *undo_seq*:

Case 1 P_i has not received from P_j any message with the label *undo_seq*, or has already held a u-lock for $T(t)$:

P_i rejects the rollback request.

Case 2 All the incoming messages are received after the state $oldchkpt_h(i).state$, $a \le h \le \alpha$, $oldchkpt_h(i)$ is the latest checkpoint such that this condition holds:

P_i sends ("roll_req", t, s, n_i) to potential participants of $T(t)$. Then P_i rolls back to $oldchkpt_h(i).state$, and discards $oldchkpt_{h+1}(i), \ldots, newchkpt_\beta(i)$.

P_i can be a participant of several rollback transactions. P_i releases a u-lock upon a final decision from the coordinator. When P_i releases all u-locks, P_i increments n_i by 1.

9. Conclusions

We have modeled concurrent checkpointing and recovery as a concurrent transaction processing problem. This model unifies the concepts of concurrent checkpointing and concurrent transaction processing. We have shown that the concurrent checkpointing and recovery problem can be solved by enforcing serializability on the corresponding transactions. A locking protocol has been designed to synchronize the transactions. Several optimizations are discussed. We incorporated these optimizations in the checkpoint/rollback algorithm. The algorithm executes checkpoint transactions or rollback transactions concurrently. Rollback transactions will never be aborted or blocked. A checkpoint transaction may be aborted only when it interferes with a rollback transaction. Blocking of a rollback transaction due to process failures can always be resolved. Blocking possibility of a checkpoint transaction has been reduced. Also, it allows normal messages to be transmitted in any order. We further generalize this algorithm when processes keep multiple checkpoints in the stable storage. A process may need to roll back to any previous checkpoint. The algorithm is more general than all the previous work.

Acknowledgment

The authors wish to thank Dr. R. Koo and Dr. S. P. Rana for valuable comments on an earlier version of this paper.

References

[BARI83] G. Barigazzi and L. Strigini, ''Application-transparent setting of recovery points,'' in *Proc. 13th IEEE Symp. Fault-Tolerant Computing*, Milano, Italy, June 1983.

[BERN79] P. A. Bernstein, D. W. Shipman, and W. S. Wong, ''Formal aspects of serializability in database concurrency control,'' *IEEE Trans. Softw. Eng. SE-5*, 3(May 1979), 203-216.

[BHAR86] B. Bhargava and Z. Ruan, ''Site recovery in distributed database systems with replicated data,'' in *Proc. 6th IEEE Int. Conf. on Distributed Comput. Syst.*, Cambridge, MA, May 1986.

[BHAR88a] B. Bhargava, P. Leu, and S. Lian, ''Experimental evaluation of concurrent checkpointing and rollback-recovery algorithms,'' *CSD-TR-790*, Dept. of Computer Sciences, Purdue University, West Lafayette, IN, July 1988.

[BHAR88b] B. Bhargava and S. Lian, ''Independent checkpointing and concurrent rollback recovery for distributed systems - An optimistic approach,'' in *Proc. 7th IEEE Symp. on Reliability in Distributed Systems*, Columbus, OH, Oct. 1988.

[BHAR88c] B, Bhargava and J. Riedl, ''Implementation of RAID,'' in *Proc. 7th IEEE Symp. on Reliability in Distributed Systems*, Columbus, OH, Oct. 1988.

[CHAN85] K. M. Chandy and L. Lamport, ''Distributed snapshots: Determining global states of distributed systems,'' *ACM Trans. Comput. Syst. 3*, 1(Feb. 1985), 63-75.

[FISC82] M. Fischer, N. Griffeth, and N. Lynch, ''Global states of a distributed system,'' *IEEE Trans. Software Eng. SE-85*, (May 1982), 198-202.

[GRAY79] J. N. Gray, ''Notes on data base operating systems.'' in *Operating systems: An advanced course*, R. Bayer, R. M. Graham, G. Seegmuller, Eds., Springer-Verlag, New York, 1979, 393-481.

[HAMM80] M. Hammer and D. Shipman, ''Reliability mechanisms for SDD-1: A system for distributed databases,'' *ACM Trans. Database Syst. 5*, 4(Dec. 1980), 431-466.

[JEFF82] D. R. Jefferson and H. A. Sowziral, ''Fast concurrent simulation using the time warp mechanism, Part I: Local control,'' Tech. Report N-1906-AF, Rand Corporation, Santa Monica, CA, Dec. 1982.

[KOO87] R. Koo and S. Toueg, ''Checkpointing and rollback-recovery for distributed systems,'' *IEEE Trans. Software Eng. SE-13*, 1(Jan. 1987), 23-31.

[LAMPO78] L. Lamport, ''Time, clocks and the ordering of events in a distributed system,'' *Commun. ACM 21*, 7(July 1978), 54-70.

[LAMPS79] B. Lampson and H. Sturgis, ''Crash recovery in a distributed storage system,'' Xerox Palo Alto research Center, Tech. Report, April 1979.

[LEU88] P. Leu and B. Bhargava, ''Concurrent robust checkpointing and recovery in distributed systems,'' in *Proc. 4th IEEE Int. Conf. Data Engineering*, Los Angeles, CA, Feb. 1988.

[MOSS83] J. E. Moss, 'Checkpoint and restart in distributed transaction systems,'' in *Proc. 3rd IEEE Symp. on Reliability in Distributed Software and Database Syst.*, July 1983.

[RAND75] B. Randell, ''System structure for software fault tolerance,'' *IEEE Trans. Software Eng. SE-1*, (June 1975), 226-232.

[RAND78] B. Randell, P. A. Lee, and P. C. Treleaven, ''Reliability issues in computing system design,'' *Computing Surveys 10*, 2(June 1978), 123-165.

[SKEE82] D. M. Skeen, ''Crash recovery in a distributed database management system,'' Ph.D. Thesis, EECS Department, University of California, Berkeley, 1982.

[TAMI84] Y. Tamir and C. H. Séquin, ''Error recovery in multicomputers using global checkpoints,'' in *Proc. 13th IEEE Int. Conf. Parallel Processing*, Aug. 1984.

[WALT82] B. Walter, ''A robust and efficient protocol for checking the availability of remote sites,'' in *Proc. 6th Int. Workshop on Distributed Data Management and Computer Networks*, 1982.