Concurrent Robust Checkpointing and Recovery in Distributed Systems†

Pei-Jyun Leu and **Bharat Bhargava**

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

ABSTRACT

We present a checkpoint/rollback algorithm for multiple processes in a distributed system that uses message passing for communication. Each process in the system can initiate the algorithm autonomously. If only one instance of the algorithm is being executed, the algorithm will force the minimal number of additional processes other than the initiator to make new checkpoints (or roll back). The new contributions of this research are as follows: 1) The concurrent execution of the algorithm for different global checkpointing instances and rollback instances initiated by several processes is allowed. Deadlocks or livelocks among different global checkpointing instances and rollback instances will not occur. 2) The algorithm is resilient to multiple process failures, and handles network partitioning in a pessimistic way. 3) The algorithm does not require that messages be received in the order in which they are sent.

1. Introduction

Rollback recovery is a technique to deal with transient errors in a system. To reduce the rollback distance, the system periodically "checkpoints" correct system states. When transient errors are captured, the system restores the last checkpointed state, and restarts.

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* [17, 18] may occur. With distributed checkpointing, the restoration of a previous global state must also be synchronized among the processes. Otherwise, *cyclic restoration* may occur [11]. 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 the cyclic effect can repeat forever. This problem can be solved as follows. A process after rolling back does not send or receive normal messages until the other rollback process also finishes rollback. That is, the process saves outgoing messages in the output queue for later transmission, and discards all incoming messages.

Distributed checkpointing and rollback have been studied in [1, 11, 20]. In contrast to transaction checkpointing [6, 8, 16] that uses undo/redo logs and concurrency controllers, the problem is mainly concerned with message exchange among the processes but not the atomicity of the transactions. We can summarize the past research [1, 11, 20] as follows. Since transient errors may interrupt the execution of one distributed checkpointing instance, 1) the participating processes follow the two-phase commit approach [8] to ensure the atomicity of one checkpointing instance; and 2) each process keeps the last checkpoint and possibly a new checkpoint created by the current instance of distributed checkpointing. In [1, 11], processes that have message exchange since their last checkpoints need to take checkpoints or roll back together. The processes participating in one global checkpointing instance or rollback instance constitute a virtual tree. The two-phase commit approach is performed hierarchically. The root process serves as the coordinator. In [20], all the processes in the system need to take checkpoints or roll back together each time. In [1, 11], different global checkpointing instances and rollback instances can interfere with one another. Interfering instances imply that the corresponding virtual trees overlap. In [1], the interference problem is solved by merging overlapping trees. A new coordinator for the two-phase commit is selected from among the roots of the overlapping trees. In [11], the interference problem is handled by allowing only one instance to complete but rejecting the other instances. That process failures may block the synchronization of one global checkpointing instance or one rollback instance is only partially solved in [1, 11, 20]. In [11], three-phase commit [19] is proposed to handle single process failure. Multiple process failures usually block the algorithm. Also network partitioning is not considered. There are several issues that need further study: non-FIFO channels that do not require the order of message send and message receive to be the same; concurrent execution of several global checkpointing instances and rollback instances; and resiliency against process failures and communication failures.

We design an algorithm that allows unlimited concurrency among interfering checkpointing and rollback instances. Different instances will not block each other, which gives good response time. Resilience against process failure follows a termination approach similar to [19]. A consistent global state is maintained among the currently operational processes. When a failed process is up, a new consistent global state is enforced. Limited cases of network partitioning are also dealt with. Further, our algorithm allows message to be received in non-FIFO order. Last, the algorithm can be modified such that a process is allowed to send messages between the instances that it makes an uncommitted checkpoint and the checkpoint is committed or aborted. In such a case, a process may need to save more than two checkpoints in stable storage. These are the desirable features for the algorithm to solve the problem.

In Section 2, we give the basic notation and definitions. Section 3 describes the checkpoint/rollback algorithm. Two illustrative examples have been given in 3.4. Section 4 deals with the correctness of the algorithm. The proofs are given in [14, 15]. In Section 5, a comparison with related work is made. Section 6 presents mechanisms for the algorithm to tolerate multiple process failures and network partitioning. Section 7 concludes the paper.

† This work was supported by a David Ross fellowship, and partially supported by UNISYS Corporation and NASA.

CH2550-2/88/0000/0154$01.00 © 1988 IEEE

2. Notation and Definitions

We use the following notation:

P_i	A process with index i.
C_i	A checkpoint of P_i.
$e_i^{s(m)}$	The event of sending a message m from P_i.
$e_i^{r(m)}$	The event of receiving a message m in P_i.
e_i^c	The event of making the checkpoint C_i in P_i.

Distributed checkpointing saves a global state at some virtual time. The virtual time clock inherits a *happens before* [12] property defined as follows:

Definition 1. Event x *happens before* event y (denoted $x \rightarrow y$) iff

i) event x occurs before event y in P_i for some i; or

ii) $x = e_i^{s(m)}$, and $y = e_j^{r(m)}$ for some message m sent from P_i to P_j; or

iii) there exists event z such that $x \rightarrow z$ and $z \rightarrow y$.

A global checkpoint is a set of the local checkpoints of processes. The consistency of a global checkpoint is defined as follows.

Definition 2 (consistency constraint C1). A global checkpoint C $= \{ C_1, C_2, \ldots, C_n \}$ of n processes $P_1, P_2, \ldots, P_n$ is *consistent* iff, for any message m from P_i to P_j, $1 \le i, j \le n$,

$$\text{if } e_j^{r(m)} \rightarrow e_j^c, \text{ then } e_i^{s(m)} \rightarrow e_i^c.$$

A consistent global checkpoint saves a consistent global state from which the processes can *safely* restart. An inconsistent checkpoint is shown in Fig. 1. If P_i and P_j both restart from their last checkpoints, then P_i will assume m has not been sent to P_j but $e_j^{r(m)}$ has been already reflected in the global state. Logically this global state does not exist under the *happen before* property in Definition 1.

Definition 3 (consistency constraint C2). A rollback instance of n processes $P_1, P_2, \ldots, P_n$ is *consistent* iff, for any message m from P_i to P_j, $1 \le i, j \le n$,

if $e_i^{s(m)}$ is undone by P_i during the rollback,
then $e_j^{r(m)}$ is also undone by P_j.

An inconsistent rollback instance produces an inconsistent state in which a message is received but logically the message has not been sent yet. We call this phenomenon *dangling receiving*.

Definition 4. A global state of n processes $P_1, P_2, \ldots, P_n$ is *consistent* iff

i) if $e_j^{r(m)} \rightarrow e_j^c$, then $e_i^{s(m)} \rightarrow e_i^c$; and

ii) if $e_j^c \rightarrow e_j^{r(m)}$ in P_j, then $e_i^{s(m)}$ exists, and has notbeen undone in P_i, where m is any message from P_i to P_j, and C_i, C_j are the last committed checkpoints of P_i, P_j, $1 \le i, j \le n$.

i) ensures that rolling back to C_i, $1 \le i \le n$, brings the system to a previous consistent global state. i) and ii) suggest that if $e_j^{r(m)}$ is not undone in P_j, nor is $e_i^{s(m)}$ in P_i. This avoids *dangling receiving* in the system.

3. The Checkpoint and Rollback Recovery Algorithm

The algorithm does not require that messages be received in the order in which they are sent. Messages can get lost during transmission. Retransmission of lost messages is handled by some end-to-end communication protocols. In the algorithm, there are two types of messages: *normal messages* and *control messages*. Messages used in the execution of the algorithm are called *control messages*. All others are called *normal messages*. Local checkpoints and rollback points are numbered sequentially. A rollback point is a time instant on a process from which the process starts rolling back. 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. 2, 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.

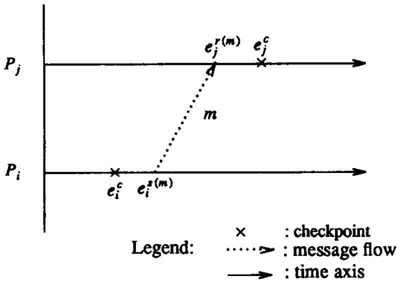

Figure 1. A sample inconsistent global checkpoint.

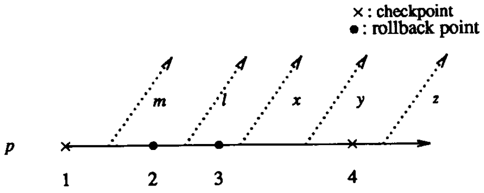

Figure 2. Numbering checkpoints and rollback points.

In the algorithm, each process saves at most two most recent checkpoints (called *oldchkpt* and *newchkpt*) in stable storage. *newchkpt* is an uncommitted checkpoint. *oldchkpt* represents the latest version of the committed checkpoint, which is updated when a new version is generated. A process P is allowed to roll back either to *oldchkpt* or to *newchkpt*. If P rolls back to *oldchkpt*, *newchkpt* (if exists) will be discarded. When *newchkpt* can commit, *oldchkpt* is updated with the value of *newchkpt*. In this algorithm, each process needs to save at most *newchkpt* and *oldchkpt* to restore a previous consistent global state.

To describe the execution of the checkpoint and rollback algorithm among cooperating processes, we derive two virtual trees, checkpoint tree and rollback tree, in the next two subsections.

3.1. Derivation of the Checkpoint Tree

Each process can make a new local checkpoint autonomously. However, it may require other processes to make new checkpoints in order to record a consistent global state. Similarly, even if a process is requested to make a new checkpoint, it may also request some other processes to make new local checkpoints. The processes participating in one global checkpointing instance can be conceptually described as a virtual checkpoint tree. The tree does not necessarily include all the processes in the system. The root represents the initiator process of the checkpoint algorithm. Each non-root node is also a process, which inherits a checkpoint request directly from its parent. The tree rooted at P_i is distinguishable by a pair (i, *initiation time*), which is regarded as a globally unique timestamp of the tree. We use $T(t)$ to denote a tree with timestamp t. Each outgoing control message for this global checkpointing instance is attached the timestamp of the tree by the sender.

The checkpoint algorithm follows a *two-phase commit* [8] approach. Either *every* node in the tree makes a new local checkpoint, or *none* of them makes it. In the first phase, 1) the initiator sends out a checkpoint request to its children; 2) the request is propagated down the virtual tree to all the descendents; each node makes an uncommitted checkpoint upon the request; and then 3) each node collects "ready_to_commit" or "roll_req" responses from all its children, and sends back a "ready_to_commit" or a "roll_req" response to its parent. In the second phase, the root, after receiving the responses from all its children, propagates its "commit" or "abort" decision to all children, which propagate the decision further down the tree. Each uncommitted checkpoint is committed or aborted accordingly.

The checkpoint tree does not exist in priori. First, a process identifies itself as the root (i.e., makes a checkpoint autonomously), and then incorporates other processes incrementally into the tree (i.e., requests other processes to make new checkpoints). The identification of child-parent relationships for a chkpt-tree $T(t)$ proceeds as follows.

Suppose P_j makes an uncommitted checkpoint C_j either autonomously or as requested. Let $\max_{ij}$ be the maximum label of the 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 did not receive any messages from P_i within that interval. P_j then regards P_i for which $\max_{ij}$ is not zero as a *potential* child, and sends the message ("chkpt_req", t, $\max_{ij}$) to P_i. Suppose when P_i receives the checkpoint request, the last committed checkpoint of P_i is C_i. Upon receiving the checkpoint request, if $seqof(C_i) \leq \max_{ij}$, and P_i has not been included in the same tree $T(t)$, and has not undone any outgoing message with the label $\max_{ij}$, then P_i is identified as a *true* child of P_j, and P_j the *true* parent of P_i in the tree $T(t)$. The child-parent relationship is approved by the child P_i, which informs its parent P_j of the result via an acknowledgment message. The child-parent relationships sufficiently determine a complete tree. P_j is a leaf node in the tree $T(t)$ if it has no true children.

The derivation of child-parent relationships follows from Definition 2. Let m be the message from P_i to P_j with the label $\max_{ij}$. $seqof(C_i) \leq \max_{ij}$ suggests the fact $e_i^c \rightarrow e_i^{s(m)}$. That P_j receives m within the interval [$seqof(C_j) - 1$, $seqof(C_j)$] indicates $e_j^{r(m)} \rightarrow e_j^c$. By definition 2, $\{C_j, C_i\}$ is an inconsistent global checkpoint. Thus P_i has to make a new checkpoint upon receiving the checkpoint request from P_j. Conceptually, P_i is described as a child of P_j in the checkpoint tree.

3.2. Derivation of the Rollback Tree

Each process can roll back autonomously, but it may require other processes to roll back in order to avoid *dangling receiving*. Similarly, even if a process is requested to roll back, it may also request some other processes to roll back. The processes participating in one global rollback instance is also a virtual tree. The root represents the initiator process of the rollback algorithm. Each non-root node is also a process, which inherits a rollback request directly from its parent. As described earlier, we use $T(t)$ to denote a tree with timestamp t.

The rollback algorithm follows a *two-phase commit* [8] approach. In the first phase, the initiator rolls back to its last checkpoint, sends "roll_req" to all its children, and waits for "roll_complete" responses from them. The request is propagated down the tree to all the descendents. Once a node has collected "roll_complete" responses from all its children, it sends back a "roll_complete" response to its parent. In the second phase, the initiator, after receiving responses from all its children, propagates its "restart" decision down the tree to all its descendents, which then resume their normal operations.

The identification of child-parent relationships for a roll-tree $T(t)$ proceeds as follows.

Suppose P_i rolls back to its previous checkpoint C_i, and undoes all the previous outgoing messages with labels greater than or equal to *undo_seq*. If P_i has ever sent P_j any message with a label greater than or equal to *undo_seq* since C_i was made, then P_i regards P_j as a *potential* child, and sends P_j the rollback request message ("roll_req", t, *undo_seq*). Upon receiving the rollback request, if P_j has not been included in the same tree $T(t)$, and has received any message m with a label greater than or equal to *undo_do*, then P_j is identified as a *true* child of P_i, and P_i the *true* parent of P_j in the tree $T(t)$. P_j informs P_i via an acknowledgment message. Since P_j may receive from P_i the rollback request before the normal messages, P_i must also inform P_j to discard all subsequent normal messages that are sent before P_i rolls back and have labels greater than or equal to *undo_seq*.

The derivation of child-parent relationships follows from Definition 3. If $e_i^{s(m)}$ is undone, so is $e_j^{r(m)}$. Therefore, the rollback of P_i will cause the rollback of P_j. Conceptually, P_j is described as a child of P_i in the rollback tree.

3.3. Discussion

If a process P_i executes the checkpoint/rollback algorithm for several checkpoint and rollback instances with different timestamps, then we say that the algorithm is concurrently executed in P_i. Two trees may overlap each other partially, or even totally (have the same set of nodes). A node may have more than one parent with respect to different trees. For example, p is a child of q in tree $T(t_1)$ but a child of r in tree $T(t_2)$. Even if $T(t_1)$ and $T(t_2)$ have the same set of nodes, the parent of p can be uniquely identified with respect to different trees.

When the checkpoint algorithm is concurrently executed in P_i, the uncommitted checkpoint made by P_i will be *shared* among different global checkpointing instances. If P_i has different parents with respect to different trees, P_i receives checkpoint requests from the parent, collects responses from all its children, and sends back responses to the parent with respect to each tree. The shared checkpoint can commit if P_i receives a commit message from any of its parents.

3.4. Illustrative Examples

This subsection gives two examples. Example 1 shows a global checkpointing instance without interference. Example 2 shows two global checkpointing instances with interference.

Example 1. (see Fig. 3)

As mentioned earlier, checkpoints in each process are numbered in increasing order. Based on the intervals, the labels of the messages x, l, m are all 1. This global checkpointing instance consists of α_2, α_3 and α_4 as shown in Fig. 3. We use ⊗ to represent a new checkpoint made in this global checkpointing instance. First, P_2 initiates the global checkpointing instance by making α_2 autonomously. After P_2 makes α_2, $\max_{32}$ is equal to 1. P_2 sends ("chkpt_req", t, $\max_{32}$) to P_3 (not shown in Fig. 3) to ask P_3 to make a new checkpoint, where t is the timestamp of the tree. P_3 identifies itself as a true child of P_2, since $seqof(\lambda_3) \leq \max_{32}$. Thus P_3 must make a new checkpoint α_3. Similarly, after P_3 makes α_3,

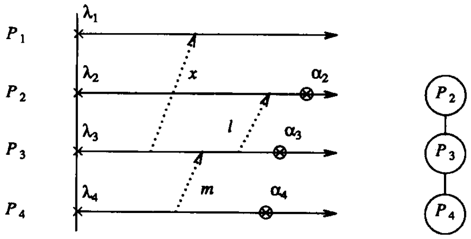

Figure 3. The process timing diagram and the checkpoint tree for example 1.

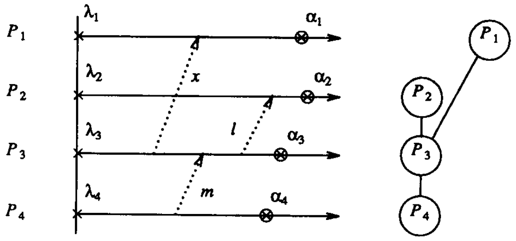

Figure 4. The process timing diagram and the checkpoint trees for example 2.

max$_{43}$ is equal to 1. P_3 sends ("chkpt_req", t, max$_{43}$) to P_4 to ask P_4 to make a new checkpoint. P_4 identifies itself as a true child of P_3, since *seqof* $(\lambda_4) \leq$ max$_{43}$. Thus P_4 must make a new checkpoint α_4. P_4 is a leaf in the checkpoint tree. During the second phase, P_2 propagates the message ("commit", t) to P_3 and P_4. Eventually α_2, α_3, α_4 commit, and $\{ \lambda_1, \alpha_2, \alpha_3, \alpha_4 \}$ is a new global checkpoint satisfying the consistency constraint C1. □

Example 2. (see Fig. 4)

In this example, there are two checkpointing instances running, one consisting of α_2, α_3 and α_4, and the other α_1, α_3 and α_4. As in Example 1, P_2 initiates one global checkpointing instance by making checkpoint α_2 autonomously. Simultaneously, P_1 initiates the other global checkpointing instance by making checkpoint α_1 autonomously. After P_1 makes α_1, max$_{31}$ is equal to 1. P_1 sends ("chkpt_req", t', max$_{31}$) to P_3 (not shown in Fig. 4), where t' represents the timestamp of the tree. P_2 also sends ("chkpt_req", t, max$_{32}$) to P_3. Then P_3 makes α_3 upon the request of P_2 or P_1 depending on which one 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 global checkpointing instances. P_3 is in both trees $T(t)$ and $T(t')$. P_4 is also in both trees if α_4 is not yet committed when P_4 receives the second checkpoint request. The shared uncommitted checkpoint $\alpha_3$can commit, if P_3 receives a commit decision from either P_2 or P_1. Eventually α_1, α_2, α_3, α_4 commit, and compose a new global checkpoint satisfying the consistency constraint C1. Since a process propagates different checkpoint requests, different global checkpointing instances will not block each other. Each root can eventually incorporate all its descendents into the tree, and complete the execution of the algorithm. □

3.5. The Checkpoint/Rollback Algorithm

3.5.1. The conventions for the algorithm

Each process P_i has a daemon process that executes 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 any procedure is exclusive. b1, b2, . . . , b8 represent the invocation conditions for the eight procedures respectively. Procedures roll_initiation() and roll_request_propagation() have the highest priority. That is, if both b5 and b6 are true, the invocation of either procedure is arbitrary. All other procedures have the same lowest priority. If more than one invocation condition are true, the invocation of either procedure is arbitrary. 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 makes 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) Processes in one global checkpointing (or global rollback) instance with timestamp t compose a chkpt-tree (or a roll-tree respectively) $T(t)$. To be precise, we use chkpt-child/chkpt-parent to denote child/parent in a chkpt-tree, and roll-child/roll-parent respectively in a roll-tree. Control messages are attached the timestamp of the tree. For example:

   send (*msg_type*, t, $\cdots$) to chkpt-children;

   or

   receive (*msg_type*, t, $\cdots$) from chkpt-parent;

   indicates the control message is sent to (or received from) its chkpt-children (or chkpt-parent) with respect to the chkpt-tree $T(t)$. Similar notation is used for a roll-tree $T(t)$. To simplify the presentation, we leave out the details in recognizing *potential* or *true* children of a node. Please refer to the previous subsections for the concepts. 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 included in 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 tuple since it is clear from the context. Also note that the sending and receiving of each control message is atomic. That is, operations following the sending of a control message can take place only after the control message has been received.

3) An uncommitted checkpoint may be shared among different global checkpointing instances. *chkpt_commit_set*(i) denotes the set of the timestamps of the global checkpointing instances that currently have a shared local uncommitted checkpoint on P_i. The uncommitted checkpoint of P_i can commit if P_i receives a control message ("commit", t) for *any* $t \in$ *chkpt_commit_set*(i). It is initialized to an empty set.

4) Similarly, concurrent execution of the algorithm for several rollback instances may occur in P_i. *roll_restart_set*(i) denotes the set of the timestamps of the global rollback instances that currently have interleaving operations in P_i. P_i may resume sending and receiving normal messages only after it receives control messages ("restart", t) for *all* $t \in$ *roll_restart_set*(i). It is initialized to an empty set.

3.5.2. 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_restart();
    endcase;
endloop;
```

The following variables are shared among these procedures: n_i, *oldchkpt*(i), *newchkpt*(i), *chkpt_commit_set*(i), *roll_restart_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
              newchkpt$(i)$ = nil
procedure chkpt_initiation();
begin
    $t$ := ($i$, initiation time);
    newchkpt$(i)$.state := current state of $P_i$;
    $n_i$ := $n_i$ + 1;
    newchkpt$(i)$.seq := $n_i$;
    send ("chkpt_req", $t$, $\max_{ki}$) to all its potential chkpt-children
      $P_k$;
    await ("pos_ack", $t$) or ("neg_ack", $t$) from all its potential
      chkpt-children;
    chkpt_commit_set$(i)$ := { $t$ };
    suspend sending normal messages in $P_i$;
end chkpt_initiation;
```

Comments: 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*]. **suspend** only limits the normal message send but not the message receive or the local computation in P_i. Thus, after the procedure is executed, P_i can resume its normal operations except sending normal messages.

```
Condition b2: $P_i$ receives ("chkpt_req", $t$, $\max_{ij}$) from $P_j$
procedure chkpt_request_propagation();
begin
    if $P_i$ is a true chkpt-child of $P_j$ with respect to the tree $T(t)$ then
        send ("pos_ack", $t$) to $P_j$;
    else
        send ("neg_ack", $t$) to $P_j$;
        return;
    endif;
    if newchkpt$(i)$ = nil then
    /*  make an uncommitted checkpoint.  */
        newchkpt$(i)$.state := current state of $P_i$;
        $n_i$ := $n_i$ + 1;
        newchkpt$(i)$.seq := $n_i$;
        chkpt_commit_set$(i)$ := { $t$ };
        suspend sending normal messages in $P_i$;
    else
    /*  reuse newchkpt$(i)$ for this global checkpointing instance.
    */
        chkpt_commit_set$(i)$ := chkpt_commit_set$(i)$ $\bigcup$ { $t$ };
    endif;
    send ("chkpt_req", $t$, $\max_{ki}$) to all its potential chkpt-children
      $P_k$;
    await ("pos_ack", $t$) or ("neg_ack", $t$) from all its potential
      chkpt-children;
    if $P_i$ has no true chkpt-children in the chkpt-tree $T(t)$ then
        send ("ready_to_commit", $t$) to $P_j$;
    endif;
end chkpt_request_propagation;
```

Comments: The idea to check if P_i is a true chkpt-child of P_j is described in 3.1. If P_i is a true chkpt-child of P_j, P_i notifies P_j via a positive acknowledgement ("pos_ack", t). Otherwise, P_i notifies P_j via a negative acknowledgment ("neg_ack", t). Since P_i does not send any normal messages until *newchkpt*(i) can commit, if *newchkpt*(i) is not *nil*, any message from P_i to P_j with the maximum label $\max_{ij}$ must be sent before *newchkpt*(i) is made. Therefore, P_i can reuse the uncommitted checkpoint *newchkpt*(i) upon receiving the checkpoint request from a different checkpoint parent P_j without violating the consistency constraint C1. P_i can recognize all its true chkpt-children based on the acknowledgments from all its potential chkpt-children for the checkpoint request.

```
Condition b3: $P_i$ receives ("ready_to_commit", $t$) from each of its
              true chkpt-children or receives ("neg_ack", $t$) from
              each of its potential chkpt-children with respect to the
              chkpt-tree $T(t)$ in chkpt_initiation();
procedure chkpt_response_collection();
begin
    if $P_i$ has a chkpt-parent in the chkpt-tree $T(t)$ then
        send ("ready_to_commit", $t$) to its chkpt-parent;
    else /*  $P_i$ is the root.  */
        if $t \in$ chkpt_commit_set$(i)$ then
        /*  make the new checkpoint committed.  */
```

```
                send ("commit", $t$) to all its true chkpt-children;
                /* does nothing if $P_i$ a leaf. */
                oldchkpt(i) := newchkpt(i);
                newchkpt(i) := nil;
                chkpt_commit_set(i) := $\varnothing$;
                resume sending normal messages in $P_i$;
            endif;
        endif;
end chkpt_response_collection;
```

Comments: The assignment statement *oldchkpt*(i) := *newchkpt*(i) will copy *newchkpt*(i)*.state* to *oldchkpt*(i)*.state* and *newchkpt*(i)*.seq* to *oldchkpt*(i)*.seq*. The assignment *newchkpt*(i) := *nil* will erase *newchkpt*(i)*.state* and *newchkpt*(j)*.seq* from the stable storage. $t \notin$ *chkpt_commit_set*(i) suggests that all the checkpoints associated with the chkpt-tree $T(t)$ have been already committed or aborted.

```
Condition b4:
    case 1) $P_i$ receives ("commit", $t$) from its chkpt-parent,
            $t \in$ chkpt_commit_set(i) or
    case 2) $P_i$ receives ("abort", $t$) from its chkpt-parent
procedure chkpt_commit/abort();
begin
    if case 1 then
    /*  make the new checkpoint committed.  */
        send ("commit", $t$) to all its true chkpt-children;
        /* does nothing if $P_i$ is a leaf. */
        oldchkpt(i) := newchkpt(i);
        newchkpt(i) := nil;
        chkpt_commit_set(i) := $\varnothing$;
        resume sending normal messages in $P_i$;
    else
    /*  discard the uncommitted checkpoint if it is not shared
        by other global checkpointing instances.  */
        chkpt_commit_set(i) := chkpt_commit_set(i) $-$ { $t$ };
        if chkpt_commit_set = $\varnothing$ then
            newchkpt(i) := nil;
            resume sending normal messages in $P_i$;
        endif;
        send("abort", $t$) to all its true chkpt-children;
        /* does nothing if $P_i$ is a leaf. */
    endif;
end chkpt_commit/abort;
```

Comments: If P_i receives ("commit", t) but $t \notin$ *chkpt_commit_set*(i), then the control message is simply discarded.

```
Condition b5: a transient error is detected in $P_i$
procedure roll_initiation();
begin
    $t$ := (i, initiation time);
    if newchkpt(i) $\neq$ nil then
        rollback to newchkpt(i).state;
    else
        rollback to oldchkpt(i).state;
    endif;
    bad_seq := $n_i$;
    send ("roll_req", $t$, bad_seq) to all its potential roll-children;
    await ("pos_ack", $t$) or ("neg_ack", $t$) from all its potential
      roll-children.
    if $P_i$ has a roll-child in the tree $T(t)$ then
        roll_restart_set(i) := roll_restart_set(i) $\cup$ { $t$ };
        suspend sending and receiving normal messages in $P_i$;
    endif;
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* will undo all the computation, message send and receive of P_i after *newchkpt*(i)*.state* is set. **suspend** only limits the normal message send and receive but not the local computation in P_i. The **suspend** statement causes all subsequent incoming messages to be discarded. After the procedure is executed, P_i can resume its normal operations except sending and receiving normal messages. P_i can recognize all its true roll-children based on the acknowledgments from all its potential roll-children for the rollback request.

```
Condition b6: $P_i$ receives ("roll_req", $t$, undo_seq) from $P_j$
procedure roll_request_propagation();
begin
    if $P_i$ is a true roll-child of $P_j$ with respect to the tree $T(t)$ then
        send ("pos_ack", $t$) to $P_j$;
    else
        send ("neg_ack", $t$) to $P_j$;
        return;
    endif;
    if newchkpt(i) $\neq$ nil and undo_seq $>$ $\max_{ji}$ then
        rollback to newchkpt(i).state;
        bad_seq := $n_i$;
    else if newchkpt(i) $\neq$ nil and undo_seq $\leq$ $\max_{ji}$ then
        rollback to oldchkpt(i).state;
        bad_seq := newchkpt(i).seq $-$ 1;
        newchkpt(i) := nil;
        send ("abort", $t'$) to all its true chkpt-children with respect
          to the chkpt-tree $T(t')$ for all $t' \in$ chkpt_commit_set(i);
        chkpt_commit_set(i) := $\varnothing$;
    else
        rollback to oldchkpt(i).state;
        bad_seq = $n_i$;
    endif;
    roll_restart_set(i) := roll_restart_set(i) $\cup$ { $t$ };
    send ("roll_req", $t$, bad_seq) to all its potential roll-children;
    await ("pos_ack", $t$) or ("neg_ack", $t$) from all its potential
      roll-children;
    suspend sending and receiving normal messages in $P_i$;
    if $P_i$ has no true roll-children in the roll-tree $T(t)$ then
        send ("roll_complete", $t$) to $P_j$;
    endif;
end roll_request_propagation;
```

Comments: The idea to check if P_i is a true roll-child of P_j is described in 3.2. If P_i is a true roll-child of P_j, P_i notifies P_j via a positive acknowledgement ("pos_ack", t). Otherwise, P_i notifies P_j via a negative acknowledgment ("neg_ack", t). *undo_seq* represents the minimum label of the messages that have just been undone by the sender P_j. *bad_seq* represents the minimum label of the messages that are subsequently undone by the receiver P_i. $\max_{ji}$ represents the maximum label of the messages sent from P_j and received within the interval [$newchkpt(i).seq - 1$, $newchkpt(i).seq$]. $undo\_seq > \max_{ji}$ suggests that all messages with labels greater than or equal to *undo_seq* are received after $newchkpt(i)$ is made. Thus the receiver P_i can undo all these messages by rolling back to $newchkpt(i).state$. $undo\_seq \le \max_{ji}$ suggests that some messages with labels greater than or equal to *undo_seq* are received before $newchkpt(i)$ is made. Thus the receiver P_i needs to roll back to $oldchkpt(i).state$. P_i can recognize all its true roll-children based on the acknowledgments from all its potential roll-children for the rollback request.

```
Condition b7: $P_i$ receives ("roll_complete", $t$) from each of its true
              roll-children with respect to the roll-tree $T(t)$
procedure roll_response_collection();
begin
    if $P_i$ is the root of the roll-tree $T(t)$ then
        $roll\_restart\_set(i) := roll\_restart\_set(i) - \{ t \}$;
        send ("restart", $t$) to all its true roll-children;
        /* does nothing if $P_i$ is a leaf. */
        if $roll\_restart\_set = \emptyset$ then
            $n_i := n_i + 1$;
            resume sending and receiving normal messages in $P_i$;
        endif;
    else
        send ("roll_complete", $t$) to its roll-parent;
    endif;
end roll_response_collection;
```

```
Condition b8: $P_i$ receives ("restart", $t$) from its roll-parent $P_j$
procedure roll_restart();
begin
    $roll\_restart\_set(i) := roll\_restart\_set(i) - \{ t \}$;
    send ("restart", $t$) to all its true roll-children;
    /* does nothing if $P_i$ is a leaf. */
    if $roll\_restart\_set = \emptyset$ then
        $n_i := n_i + 1$;
        resume sending and receiving normal messages in $P_i$;
    endif;
end roll_restart;
```

3.5.3. Extension of the algorithm

Now we modify the algorithm so that normal message send in P_j is not suspended after $newchkpt(j)$ is set. Between the instances that $newchkpt(j)$* is set and $newchkpt(j)$ can commit, each outgoing normal message to any receiver P_i must be in addition attached the marker "marker(t')", where t' is the timestamp of the global checkpointing instance in which $newchkpt(j)$ is made. The marker fulfills a similar purpose to that in [4]. Upon receiving the marker attached to a normal message, P_i invokes the procedure**:

chkpt_initiation();

Only after the procedure has been executed, P_i can consume the normal message. All subsequent markers "marker(t')" with the same timestamp t' are ignored. Under this modification, P_i may need to keep more than one uncommitted checkpoint in the stable storage. Suppose P_i currently keeps the uncommitted checkpoints $newchkpt_a(i)$, $newchkpt_{a+1}(i), \ldots, newchkpt_l(i)$ in stable storage for different global checkpointing instances, where subscripts represent the increasing time order in which they are made. Each uncommitted checkpoint can be shared among several checkpointing instances. Upon receiving a checkpoint request ("chkpt_req", t, $\max_{ij}$) from P_j, P_i may have the following cases for any previous outgoing message m to P_j with the label $\max_{ij}$:

Case 1 P_i sends m within the interval $[1, oldchkpt(i).seq]$, i.e., $\max_{ij} < oldchkpt(i).seq$:

P_i rejects the checkpoint request, since P_i is not a true checkpoint child of P_j.

Case 2 P_i sends m within the interval $[newchkpt_h(i).seq - 1, newchkpt_h(i).seq]$, i.e., $\max_{ij} = newchkpt_h(i).seq - 1$:

P_i reuses $newchkpt_h(i)$ for this request. Then P_i sends ("chkpt_req", t, $\max_{ki}$) to all its potential checkpoint children P_k, where $\max_{ki}$ is the maximum label of the messages sent from P_k and received within the interval $[newchkpt_h(i).seq - 1, newchkpt_h(i).seq]$. After receiving a ("commit", t) decision for any $t \in chkpt\_commit\_set_h(i)$, P_i commits $newchkpt_h(i)$.

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

P_i must make a new checkpoint $newchkpt_{l+1}(i)$, where $newchkpt_{l+1}(i).seq$ is set to $n_i + 1$. Then P_i sends ("chkpt_req", t, $\max_{ki}$) to all its potential checkpoint children P_k, where $\max_{ki}$ is the maximum label of the messages sent from P_k and received within the interval $[newchkpt_{l+1}(i).seq - 1, newchkpt_{l+1}(i).seq]$. Then n_i is incremented by one. After receiving a ("commit", t) decision for any $t \in chkpt\_commit\_set_{l+1}(i)$, P_i commits $newchkpt_{l+1}(i)$.

In Cases 2 and 3, when $newchkpt_a(i)$, $newchkpt_{a+1}(i), \ldots, newchkpt_h(i)$ all commit, $oldchkpt(i)$ is updated with the value of $newchkpt_h(i)$, and $newchkpt_a(i)$, $newchkpt_{a+1}(i), \ldots, newchkpt_h(i)$ are discarded.

Operations for rollback also need modifying: The initiator process of a global rollback instance always rolls back to its last checkpoint (either committed or uncommitted), and propagates its rollback request to its descendents in the rollback tree. Upon receiving the rollback request ("roll_req", t, *undo_seq*) from P_j, P_i may have the following cases for incoming normal messages from P_j with labels $\ge undo\_seq$:

Case 1 P_i has not received from P_j any message with a label $\ge undo\_seq$:

P_i rejects the rollback request, since P_i is not a true rollback child of P_j.

* Since P_i may keep more than one uncommitted checkpoint in the stable storage, *newchkpt(j)* denotes the most recent one.

** The execution is delayed if the daemon process of P_i is currently executing a procedure.

Case 2 P_i receives from P_j a message with a label ≥ *undo_seq* within the interval [$newchkpt_h(i).seq - 1$, $newchkpt_h(i).seq$] but not any preceding intervals:

Case 2.1 $h = a$: P_i rolls back to $oldchkpt(i).state$.

Case 2.2 $h > a$: P_i rolls back to $newchkpt_{h-1}(i).state$.

In the above two subcases, the uncommitted checkpoints $newchkpt_h(i)$, $newchkpt_{h+1}(i), \ldots, newchkpt_l(i)$ are discarded, and the associated global checkpointing instances are aborted. Then P_i sets *bad_seq* to $newchkpt_h(i).seq - 1$, and sends ("roll_req", t, *bad_seq*) to all its potential rollback children P_k.

Case 3 P_i receives form P_j a message with a label ≥ *undo_seq* within the interval $[n_i, \infty)$ but not any preceding intervals:

P_i rolls back to $newchkpt_l(i).state$, and sets *bad_seq* to n_i. Then P_i sends ("roll_req", t, *bad_seq*) to all its potential rollback children P_k.

In Cases 2 and 3, after P_i receives the ("restart", t) decision for all $t \in roll\_restart\_set(i)$, P_i increments n_i by 1.

In Section 6, we further design mechanisms to handle multiple process failures and network partitioning. The mechanisms can be implemented as a set of exception handlers, and can be easily incorporated into the algorithm.

4. Correctness Arguments

This section outlines the theorems. We refer the reader to [14, 15] for the detailed proofs.

Definition 5. One global checkpointing instance associated with a checkpoint tree $T(t)$ is said to *terminate with success* (respectively *terminate with failure*) if *every* node in the tree $T(t)$ receives the root's checkpoint request, makes a new committed checkpoint (respectively makes no committed checkpoint), and resumes its normal operations.

Definition 6. One global rollback instance associated with a rollback tree $T(t)$ is said to *terminate* if *every* node in the tree $T(t)$ receives the root's rollback request, rolls back, and resumes its normal operations.

Without loss of generality we assume the root implicitly receives the checkpoint/rollback request from itself.

Theorem 1. Each global checkpointing instance will eventually *terminate* (either *with success* or *with failure*). Also each global rollback instance will eventually *terminate*. □

Theorem 2. Starting with a consistent global state (see Definition 4), the execution of the checkpoint/rollback algorithm ends with a consistent global state after an arbitrary number of global checkpointing instances and rollback instances *terminate*. □

Definition 7. One global checkpointing instance associated with a checkpoint tree $T(t)$ is said to be *isolated* if a) the checkpointing instance is not concurrently executed with other global checkpointing instances or rollback instances on any node of $T(t)$. b) each node of $T(t)$, after making an uncommitted checkpoint, does not send out any normal messages until the checkpoint is committed.

Suppose $P_1, P_2, \ldots, P_k$ make new checkpoints in one isolated global checkpointing instance, and P_1 is the initiator. Let $\mathbf{C} = \{ C_1, C_2, \ldots, C_k \}$ be the set of committed checkpoints of $P_1, P_2, \ldots, P_k$ made in the checkpointing instance. The next theorem states the *minimality* of the checkpoint algorithm.

Theorem 3. Given a consistent global state (see Definition 4), each *isolated* global checkpointing instance requires the minimal number of additional processes other than the initiator to make new checkpoints without violating the consistency constraint **C1**. That is, the global checkpoint $\mathbf{C}' = \mathbf{C} - \{ C_i \} \cup \{ C_i' \}$ is inconsistent for any $2 \le i \le k$, where C_i' is the previous committed checkpoint of P_i before P_i participates in the checkpointing instance. □

Definition 8. One global rollback instance associated with a rollback tree $T(t)$ is said to be *isolated* if the rollback instance is not concurrently executed with other global checkpointing instances or rollback instances on any node of $T(t)$.

Suppose $P_1, P_2, \ldots, P_k$ roll back in one isolated global rollback instance, and P_1 is the initiator. The next theorem states the *minimality* of the rollback algorithm.

Theorem 4. Given a consistent global state (see Definition 4), each *isolated* global rollback instance requires the minimal number of additional processes other than the initiator to roll back without violating the consistency constraint **C2**. That is, for any $2 \le j \le k$, if P_j does not roll back in the rollback instance, then there exists a message m from P_i to P_j such that $e_i^{s(m)}$ is undone by P_i, but $e_j^{r(m)}$ is not undone by P_j, $1 \le i, j \le k$. □

5. Comparison with Related Work

Several distributed checkpointing and recovery mechanisms can be found in [1, 11, 20]. Their distinguishing features are as follows:

Barigazzi-Strigini algorithm [1]:

- 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 after making an uncommitted checkpoint can resume its normal operations only after the checkpoint is committed or aborted.

Tamir-Séquin algorithm [20]:

- All the processes in the system need to take checkpoints or roll back together.
- A process after making an uncommitted checkpoint can resume its normal operations only after the checkpoint is committed or aborted.

Koo-Toueg algorithm [11]:

- Messages are transmitted via FIFO channels.
- Only processes that have message exchange since their last checkpoints need to take checkpoints or roll back together. Concurrent execution of the algorithm among different global checkpointing instances and rollback instances is not allowed.
- A process is not allowed to send normal messages between the instances that it makes an uncommitted checkpoint and the checkpoint is committed or aborted.
- 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:

- Messages can be transmitted via non-FIFO channels.
- Only processes that have message exchange since their last checkpoints need to take checkpoints or roll back together. Higher concurrency in the execution of the algorithm is achieved.

- A process is allowed to send normal messages between the instances that it makes an uncommitted checkpoint and the checkpoint is committed or aborted.
- Resilient to multiple process failures and limited cases of network partitioning. This is discussed in detail in Section 6.

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 [10]. 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.

Concurrency is also an important feature for distributed checkpointing and rollback recovery. The checkpoint/rollback algorithm proposed in [11] does not allow concurrent execution. That is, there can be at most one checkpointing instance or rollback instance taking place on each process. All other instances will be rejected. Thus deadlocks are prevented in the synchronization of several global checkpointing instances and rollback instances. However, a livelock may still occur since two instances may reject each other for infinitely many times, and thus neither of them can terminate. Our algorithm allows concurrent execution for different checkpointing instances and rollback instances. Deadlocks and livelocks will not occur at all. We justify the above features in detail in [14, 15].

6. Resiliency against Process Failures

While processes are executing the checkpoint/rollback algorithm, 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 [13]. 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 [2, 9, 22]. 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) Message spoolers [9] are available for failed processes. Messages addressed to the failed process are redirected to its message spoolers. Messages can be replicated on multiple spoolers to enhance reliability.

We propose the following rules to resolve blocking.

1) If a process fails, and does not respond to a checkpoint request, the requesting process i) propagates the "abort" decision to its other true checkpoint children and their descendents, and then ii) initiates a global rollback instance.

2) If a process fails, and does not respond to a rollback request, the requesting process excludes the failed process as a true rollback child, and continues the execution of the rollback algorithm.

3) A restarting process first determines whether to commit or abort its uncommitted checkpoint if any, and then initiates a global rollback instance. The "commit" or "abort" decision is obtained from its message spoolers. If all its spoolers fail, it inquire all other processes about the decision. The same approach is used in rule 6). If the restarting process was the checkpointing initiator, it always aborts its uncommitted checkpoint, and initiates a global rollback instance.

4) If the initiator of a checkpointing instance fails before sending its "commit" or "abort" decision, this global checkpointing instance is aborted under the control of its true checkpoint children. That is, each child informs its descendents to abort their uncommitted checkpoints.

5) If the initiator of a rollback instance fails before sending its "restart" decision, each of its true rollback children considers itself as a substitute, and continues the execution. That is, each substitute collects the "roll_complete" responses from all its children, and then propagates the "restart" decision to all its descendents.

6) If a process fails, and does not propagate a checkpoint "commit" or "abort" decision (respectively a rollback "restart" decision) of an initiator, the true checkpoint children (respectively the true rollback children) of the failed process inquire all other operational processes in the system about the decision. Based on the algorithm, the uncommitted checkpoint of P_i can commit if P_i receives a control message ("commit", t) for *any* $t \in$ *chkpt_commit_set*(i) from its checkpoint parents. If all checkpoint parents fail without sending the control messages ("commit", t) for all $t \in$ *chkpt_commit_set*(i), P_i broadcasts its inquiry and see if any other process has received a control message ("commit", t) for *any* $t \in$ *chkpt_commit_set*(i). P_i can commit the uncommitted checkpoint and propagate the decision accordingly. Similarly, in the execution of the rollback algorithm, P_i may resume sending and receiving normal messages only after it receives control messages ("restart", t) for *all* $t \in$ *roll_restart_set*(i). If any of P_i's rollback parents fails without sending a control message ("restart", t) for some $t \in$ *roll_restart_set*(i) to P_i, P_i gets the decision by inquiring other processes and see if any of them has received the message ("restart", t). It is possible that all the processes that receive the ("commit", t), ("abort", t), or ("restart", t) message have failed. In this case, P_i must wait.

This mechanism ensures that a consistent global state is maintained among the currently operational processes. After a failed process is up, a new consistent global state is enforced. Each rule can be implemented as an exception handler, which can be easily incorporated into the original algorithm. An exception occurs when a process restarts, or is blocked while waiting for responses from a failed process. Then the corresponding exception handlers are invoked.

We now briefly address the network partitioning problem. Communication link failures or site failures can cause a system of processes to be partitioned into groups of processes. Processes can only communicate with processes within the same partition. To eliminate the blocking in the execution of the algorithm, processes in each partition restore their last checkpoints made before the partitioning. Then any subsequent checkpoint tree or a rollback tree will consist of processes all from a single partition. Thus no synchronization of processes among different partitions is necessary, and blocking can be avoided.

Essentially, we handle process failures and network partitioning differently. However, it is impossible to distinguish a failed process from an operational process in a different partition. In absence of this distinction, we can handle network partitioning in a pessimistic way: Processes in a minor partitions, which has less than one half of the total votes [7, 21], are all regarded to be failed. Operational processes in one major partition, which has more than one half of the total votes [7, 21], are still regarded as operational. Then we apply the rules in 1) to 6) to the network partitioning problem. The regarded-to-be-failed processes in a minor partition follow the rule 3) when the minor partition merges into the major partition. A major partition may split further into smaller partitions, none of which contains a majority of votes. In such a case, a major partition can be determined on a relative basis [3, 5]. That is, a partition that splits from a major partition becomes a new major partition if it contains more than one half of the total votes in the previous major partition.

7. Conclusions

We have formalized and incorporated concurrency and resiliency in the checkpoint/rollback algorithm. The algorithm allows unlimited concurrency among interfering global checkpointing instances and rollback instances. The algorithm is resilient to multiple process failures and communication failures. Further, it does not require messages to be received in the order in which they are sent. This mechanism is more general than all the previous work.

Acknowledgment

The authors wish to thank Dr. R. Koo for valuable comments on this paper.

References

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

[2] 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.

[3] B. Bhargava and P. Ng, ''A dynamic majority determination algorithm for reconfiguration of network partitions,'' *Int. Journal of Information Science* (to appear).

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

[5] D. Dacev and W. Burkhard, ''Consistency and recovery control for replicated files,'' in *Proc. 10th ACM Symp. on Operating Systems Principles*, 1985.

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

[7] D. K. Gifford, ''Weighted voting for replicated data,'' in *Proc. 7th ACM Symp. on Operating System Principles*, Pacific Grove, CA, Dec. 1979.

[8] 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.

[9] 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.

[10] 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.

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

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

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

[14] P. Leu and B. Bhargava, ''Concurrent robust checkpointing and recovery in distributed systems,'' Tech. Report CSD-TR-689, Dept. of Computer Sciences, Purdue University, West Lafayette, June 1987.

[15] P. Leu, ''Consistent state detection and recovery for concurrent processing,'' Ph.D. Thesis, Dept. of Computer Sciences, Purdue University, West Lafayette (in preparation).

[16] 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.

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

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

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

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

[21] R. H. Thomas, ''A majority consensus approach to concurrency control,'' *ACM Trans. Database Syst. 4*, 2(June 1979), 180-209.

[22] 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.