A Checkpointing-Recovery Scheme For Domino-Free Distributed Systems

Article · October 1998

DOI: 10.1007/978-1-4615-5449-3_5 · Source: CiteSeer

3 authors, including:

Francesco Quaglia
Sapienza University of Rome
201 PUBLICATIONS **1,459** CITATIONS
SEE PROFILE

Bruno Ciciani
Sapienza University of Rome
67 PUBLICATIONS **429** CITATIONS
SEE PROFILE

5 A CHECKPOINTING-RECOVERY SCHEME FOR DOMINO-FREE DISTRIBUTED SYSTEMS

Francesco Quaglia, Bruno Ciciani, Roberto Baldoni

Dipartimento di Informatica e Sistemistica
Universita' di Roma "La Sapienza"
Via Salaria 113, I-00198, Roma, Italy *

quaglia,ciciani,baldoni@dis.uniroma1.it

Abstract: Communication-induced checkpointing algorithms require cooperating processes, which take checkpoints at their own pace, to take some forced checkpoints in order to guarantee domino-freeness. In this paper we present a checkpointing-recovery scheme which reduces the number of forced checkpoints, compared to previous solutions, while piggybacking, on each message, only three integers as control information. This is achieved by using information about the history of a process and an equivalence relation between local checkpoints that we introduce in this paper. A simulation study is also presented which quantifies such a reduction.

INTRODUCTION

In computer systems, the rollback recovery technique allows the restoration of a consistent state in case of failure [12]. Consistent checkpointing is a way to implement this technique in distributed systems [6]. It consists of determining a set of local checkpoints, one for each process (i.e, recovery line), from which the distributed application can be resumed after a failure. A checkpoint is a local state saved on stable storage and a recovery line is a set of checkpoints in which no checkpoint *"happens-before"* another [8].

The identification of a recovery line in distributed applications is not a simple task due to the presence of messages which establish dependencies between local states in different processes. If the local checkpoints are taken without

*Partial funding provided by the *Consiglio Nazionale delle Ricerche* and by the Scientific Cooperation Network of the European Community OLOS (no. ERB4050PL932483).

any coordination (for example by using a local periodic algorithm) a recovery line, close to the end of the computation, might not exist, and a failure could lead to an unbounded rollback that might force the application to its initial state. This phenomenon is known as *domino effect* [2, 12].

Many checkpointing algorithms have been proposed to compute on-line recovery lines. These algorithms can be classified into two categories according to the policy that masters the checkpointing activity.

The first category is classified as *synchronous* approach, and is characterized by an explicit processes coordination, by means of control messages [5, 7]. Following this approach, the last taken checkpoint of each process always belongs to a recovery line because processes take their checkpoints in a mutual consistent way.

In the second category (namely, *communication-induced algorithms*), processes are allowed to take local checkpoints at their own pace (i.e., *basic* checkpoints); the coordination is achieved by piggybacking control information on application messages. This control information directs processes to take *forced* checkpoints in order to ensure the advancement of the recovery line (the interested reader can refer to [6] for a complete survey on rollback-recovery algorithms).

Communication-induced algorithms have been classified in [10] according to the characterization of Netzer and Xu based on the notion of *zigzag path* (*z–path* for short) [11]. A z–path is a generalization of a causal path. In some cases, it allows a message to be sent before the previous one in the path is received. A z–path actually establishes a dependency between a pair of checkpoints. Communication-induced checkpointing algorithms fall in two main classes: z–path–free and z–cycle–free (a z–cycle is a z–path from a checkpoint to itself) algorithms. Members of the first class allow to track on-line all dependencies between local checkpoints [1, 3, 16] by using, at least, a vector of integers as a control information on application messages.

On-line dependency tracking allows, among other properties described in [6, 16], to associate on-the-fly a recovery line to each local checkpoint. The latter property can be achieved, with usually less overhead, by z–cycle–free algorithms [4, 9]. Indeed, they use just an integer (sequence number) as a control information to associate a local checkpoint to a recovery line.

The goal of this paper is to present a checkpointing–recovery scheme for distributed systems. It consists of a z–cycle–free checkpointing algorithm and an asynchronous recovery scheme. As in [9] the scheme requires to piggyback three integers as control information on the application messages. One is due to the checkpointing algorithm and two to the recovery.

The proposed checkpointing algorithm ensures the progression of the recovery line reducing the number of checkpoints compared to previous proposals. We achieve this goal by introducing an equivalence relation between local checkpoints of a process and by exploiting the events' history of a process. The equivalence relation allows, in some cases, to advance the recovery line without increasing its sequence number, thus, it keeps as small as possible the difference between the sequence numbers in different processes that is the major

cause of forced checkpoints. We also show experimental results which quantify the reduction of the number of local checkpoints taken by our algorithm in a distributed execution.

The recovery algorithm is similar to the one in [9]. It is fully asynchronous and requires only two integers as control information. Compared to [9], in some circumstances, the proposed recovery scheme does not force processes to take additional checkpoints before resuming the execution.

The paper is organized as follows. The second section introduces the system model and some definitions and notations. The third section presents the class of z–cycle–free algorithms. The fourth section describes the checkpointing algorithm and a performance evaluation. The fifth section describes the recovery scheme. Some concluding remarks are given in the last section.

MODEL OF THE COMPUTATION

We consider a distributed computation consisting of n processes $(P_1, P_2, \ldots P_n)$ which interact by means of messages sent over reliable point-to-point channels (transmission times are unpredictable but finite). Processes do not share memory, do not share a common clock value and fail following a fail-stop behavior [13]. Moreover, we assume no process can fail during a recovery action.

Execution of a process produces a sequence of events which can be classified as: *send* events, *receive* events and *internal* events. An internal event may change only local variables; send or receive events involve communication. The causal ordering of events in a distributed execution is based on Lamport's *happened-before* relation [8] denoted $\rightarrow$. If a and b are two events then $a \rightarrow b$ iff one of these conditions is true:

(i) a and b are produced on the same process with a first;

(ii) a is the send event of a message M and b is the receive event of the same message;

(iii) there exists an event c such that $a \rightarrow c$ and $c \rightarrow b$.

Such a relation allows to represent a distributed execution as a partial order of events, called $\widehat{E} = (E, \rightarrow)$ where E is the set of all events.

A local checkpoint dumps the current process state on stable storage. The k–th checkpoint in process P_i is denoted as $C_{i,k}$, and we assume that each process P_i takes an initial checkpoint $C_{i,0}$. Each process takes local checkpoints either at its own pace (for example by using a periodic algorithm) or forced by some communication pattern. A checkpoint interval $I_{i,k}$ is the set of events between $C_{i,k}$ and $C_{i,k+1}$ (note that if $C_{i,k+1}$ is not taken, $I_{i,k}$ is unbounded on the right).

A message M sent by P_i to P_j is called *orphan* with respect to a pair (C_{i,x_i}, C_{j,x_j}) iff its receive event occurred before C_{j,x_j} while its send event occurred after C_{i,x_i}.

A global checkpoint $\mathcal{C}$ is a set of local checkpoints $(C_{1,x_1}, C_{2,x_2}, \ldots, C_{n,x_n})$. A global checkpoint $\mathcal{C}$ is consistent if no orphan message exists in any pair of local checkpoints belonging to $\mathcal{C}$. We use the term consistent global checkpoint

and recovery line interchangeably. In the remainder of the paper we use the following definition:

Definition. *Two local checkpoints $C_{i,h}$ and $C_{i,k}$ of process P_i are equivalent with respect to the recovery line $\mathcal{L}$, denoted $C_{i,h} \equiv_{\mathcal{L}} C_{i,k}$, if $C_{i,h}$ belongs to the recovery line $\mathcal{L}$, and the set $\mathcal{L}' = \mathcal{L}$-$\{C_{i,h}\} \cup \{C_{i,k}\}$ is a recovery line.*

Let us now recall the z–path definition introduced by Netzer and Xu [11].

Definition. *A z–path exists from $C_{i,x}$ to $C_{j,y}$ iff there are messages M_1, M_2, ... , M_n such that:*

(1) M_1 is sent by process P_i after $C_{i,x}$;

(2) if M_k $(1 \leq k < n)$ is received by process P_r, then M_{k+1} is sent by P_r in the same or in a later checkpoint interval (although M_{k+1} may be sent before or after M_k is received);

(3) M_n is received by process P_j before $C_{j,y}$.

A checkpoint $C_{i,x}$ is involved in a z–cycle if there is a z-path from $C_{i,x}$ to itself. According to the Netzer and Xu theorem, each local checkpoint not involved in a z–cycle belongs to at least one recovery line.

Z–CYCLE–FREE CHECKPOINTING ALGORITHMS

In this section we give a short overview of z–cycle–free checkpointing algorithms. For the interested readers a complete survey can be found in [10].

In those algorithms [4, 9] each process P_i assignes a sequence number SN_i to each local checkpoint $C_{i,k}$ (we denote this number as $C_{i,k}.SN$). It is assumed to assign SN_i equal to zero to $C_{i,0}$. The sequence number SN_i is attached as a control information $M.SN$ on each outgoing message M. A recovery line $\mathcal{L}_{SN}$ includes local checkpoints with the same sequence number (SN) one for each process (if there is a jump in the sequence number of a process the first checkpoint with greater sequence number must be included). The basic rules, defined by Briatico et al. [4], to update the sequence numbers are:

R1 When a basic checkpoint $C_{i,k}$ is scheduled, the checkpoint $C_{i,k}$ is taken, SN_i is increased by one and $C_{i,k}.SN$ is set to SN_i;

R2 : Upon the receipt of a message M in $I_{i,k-1}$, if $SN_i < M.SN$ a forced checkpoint $C_{i,k}$ is taken, the sequence number $M.SN$ is assigned to $C_{i,k}$ and SN_i is set to $M.SN$; then the message is processed.

The absence of z-cycles is achieved since a message piggybacking a sequence number W is always received by a process after a checkpoint whose sequence number is greater than or equal to W.

Manivannan and Singhal present an algorithm [9] which tries to keep the indices of all the processes close to each other and to push the recovery line

as close as possible to the end of the computation. In this way, it reduces the probability that $SN_i < M.SN$ that, in turn, decreases the number of the local checkpoint forced by a basic one. The Manivannan-Singhal algorithm is based on the following two observations:

1. Let each process be endowed with a counter incremented each x time unit (where x is the smallest period between two basic checkpoints among all processes). If each tx (with $t \geq 1$) time units a process takes a basic checkpoint with sequence number tx, $\mathcal{L}_{tx}$ is the closest recovery line to the end of the computation.

2. If the last checkpoint taken by a process is a forced one and its sequence number is greater than or equal to the next scheduled basic checkpoint, then the basic checkpoint is skipped.

Point 1 would allow a heavy reduction of forced checkpoints, by synchronizing actually the action to take basic checkpoints in distinct processes. However, it is not always possible to get x in a system of independend processes (its access could be precluded by a process), moreover point 1, as explained in [9], requires to put a bound on the local clocks' drift.

Point 2 reduces the number of basic checkpoints compared to [4]. So, let us assume process P_i endows a flag $skip_i$ which indicates if at least one forced checkpoint is taken between two successive scheduled basic checkpoints (this flag is set to FALSE each time a basic checkpoint is scheduled, and set to TRUE each time a forced checkpoint is taken). A version of Manivannan-Singhal algorithm that does not need private information about other processes and to bound clocks' drift can be sketched by the following rules:

R1' : When a basic checkpoint $C_{i,k}$ is scheduled, if $skip_i = TRUE$ then $skip_i = FALSE$, else SN_i is increased by one, the checkpoint $C_{i,k}$ is taken and its sequence number is set to SN_i;

R2' : Upon the receipt of a message M in $I_{i,k-1}$, if $SN_i < M.SN$ then a forced checkpoint $C_{i,k}$ is taken with sequence number $M.SN$, SN_i is set to $M.SN$ and $skip_i = TRUE$; then the message is processed.

A CHECKPOINTING ALGORITHM

The checkpointing algorithm adopts the same mechanism described above to skip some scheduled basic checkpoints but refines both rules R1' and R2'. In particular rules R1' and R2' become:

R1": When a basic checkpoint $C_{i,k}$ is scheduled,
If $skip_i$ **then** $skip_i = FALSE$
else $C_{i,k}$ is taken;
SN_i is increased by one iff $\neg(C_{i,k} \equiv_{\mathcal{L}} C_{i,k-1})$
where $\mathcal{L}$ is the recovery line to which $C_{i,k-1}$ belongs;
the sequence number SN_i is assigned to $C_{i,k}$;

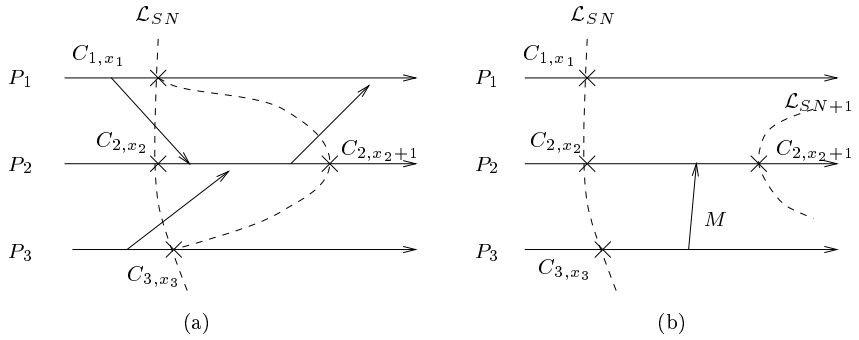

Figure 5.1 (a) execution in which C_{2,x_2} and C_{2,x_2+1} are equivalent wrt $\mathcal{L}_{SN}$; (b) execution in which C_{2,x_2} and C_{2,x_2+1} are not equivalent

R2": Upon the receipt of a message M in $I_{i,k-1}$:

(a) **If** $SN_i < M.SN$ **and** *there has been at least a send event in* $I_{i,k-1}$ **then** a forced checkpoint $C_{i,k}$ is taken with sequence number $M.SN$; SN_i is set to $M.SN$ and $skip_i$ is set to $TRUE$;

(b) **If** $SN_i < M.SN$ **and** *there has been no send event in* $I_{i,k-1}$ **then** SN_i is set to $M.SN$ and $C_{i,k-1}.SN$ is set to $M.SN$;

the message m is processed;

Rule R1" requires to define when a basic checkpoint $C_{i,k}$ is not equivalent to the previous one ($C_{i,k-1}$) whose sequence number is SN. This happens when there exists, in $I_{i,k-1}$, at least one receive event of a message M which piggybacks a sequence number equal to SN. For example, Figure 5.1.b shows a recovery line $\mathcal{L}_{SN}$ and the message M which is orphan with respect to the ordered pair C_{3,x_3} and C_{2,x_2+1}, so checkpoint C_{2,x_2+1} is not equivalent to C_{2,x_2} with respect to $\mathcal{L}_{SN}$ and it will be a part of the recovery line $\mathcal{L}_{SN+1}$. If such a message M does not exist, as in Figure 5.1.a, the checkpoint C_{2,x_2+1} is equivalent to C_{2,x_2} with respect to $\mathcal{L}_{SN}$ so it can belong to the recovery line $\mathcal{L}_{SN}$.

Rule R2" states that there is no reason to take a forced checkpoint if there has been no send event in the current checkpoint interval till the receipt of message M piggybacking the sequence number $M.SN$. Indeed, no non-causal z–path can be formed due to the receipt of M and then the sequence number of the last checkpoint $C_{i,k-1}$ can be updated to belong to the recovery line $\mathcal{L}_{M.SN}$. For example, in Figure 5.2.a, the local checkpoint C_{3,x_3} can belong to the recovery line $\mathcal{L}_{SN+1}$. On the other hand, if a send event has occurred in the checkpoint interval I_{3,x_3}, as shown in Figure 5.2.b, a forced checkpoint C_{3,x_3+1}, belonging to the recovery line $\mathcal{L}_{SN+1}$, has to be taken upon the receipt of message M.

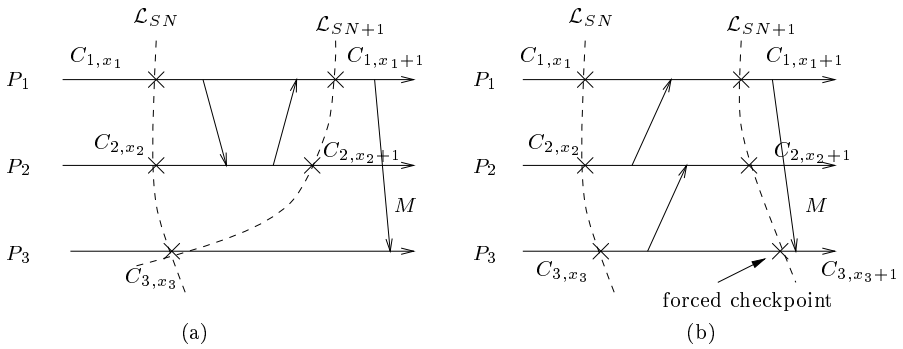

Figure 5.2 (a) upon the receipt of M, C_{3,x_3}, previously tagged SN, can be a part of the recovery line $\mathcal{L}_{SN+1}$; (b) upon the receipt of M, C_{3,x_3} cannot be a part of $\mathcal{L}_{SN+1}$, then the forced checkpoint C_{3,x_3+1} is taken

Rule R2" directly decreases the number of forced checkpoints taken by our algorithm compared to the rule R2'. Rule R1" and the second part of R2" keep the sequence numbers in distinct processes as close as possible reducing so the probability of forced checkpoints.

Data structures and process behavior. We assume each process P_i has the following data structures:

SN_i, RN_i: **integer**;
$send_i$, $recv_i$, $skip_i$: **boolean**.

The variable RN_i represents the value of the maximum sequence number ($M.SN$) associated to received messages (the initial value is RN_i=-1).

The boolean variable $send_i$ (resp. $recv_i$) is set to TRUE if at least one send (resp. receive) event has occurred in the current checkpoint interval. It is set to FALSE each time a checkpoint is taken.

The semantic of the variable SN_i and $skip_i$ have been explained in the third section. In Figure 5.3 the process behavior is shown (the procedures and the message handler are executed in atomic fashion).

Correctness Proof

Lemma 1 *If a message M is sent by process P_i after a local checkpoint $C_{i,k}$ such that $C_{i,k}.SN = W$, it is received, by a process P_j, after a local checkpoint whose sequence number is larger than or equal to W.*

Proof As M has been sent after $C_{i,k}$, then $M.SN \geq W$. When M is received by process P_j in $C_{j,h-1}$, if $SN_j \geq M.SN$, the claim trivially follows. If $SN_j < M.SN$, by rule R2", either a forced checkpoint $C_{j,h}$ is taken and $C_{j,h}.SN = M.SN$ (see R2".a) or, if there is no send event in the current checkpoint interval, $C_{j,h-1}.SN = M.SN$ (see R2".b). In both cases the claim follows. □

```
init SN_i := 0;  RN_i := −1;  send_i := FALSE;  recv_i := FALSE;  skip_i := FALSE;

when a basic checkpoint C_{i,k} is scheduled:
begin
      if skip_i
      then skip_i := FALSE
      else
          begin
          if (recv_i and RN_i = SN_i) then SN_i := SN_i + 1;          % see rule R1" %
          take a basic checkpoint C_{i,k};
          C_{i,k}.SN := SN_i;          % assign the sequence number to the new checkpoint %
          send_i := FALSE;
          recv_i := FALSE;
          end
end.
procedure SEND(M, P_j):                  % M is the message, P_j is the destination %
begin
      M.SN := SN_i;
      send (M) to P_j;
      send_i := TRUE
end.

when (M) arrives at P_i in I_{i,k−1}:
begin
      if (M.SN > SN_i and send_i)                                    % see rule R2".a %
      then
           begin
           take a forced checkpoint C_{i,k};
           SN_i := M.SN;
           RN_i := M.SN;
           C_{i,k}.SN := SN_i;          % assign the sequence number to the new checkpoint %
           send_i := FALSE;
           skip_i := TRUE;
           end
      else
          if M.SN > SN_i                                             % see rule R2".b %
          then
               begin
               SN_i := M.SN;
               RN_i := M.SN;
               C_{i,k−1}.SN := SN_i;   % update the sequence number of the last checkpoint %
               end
          else
              if M.SN > RN_i then RN_i := M.SN;
      recv_i := TRUE;
      process the message
end.
```

Figure 5.3 A z–cycle–free checkpointing algorithm

Theorem 2 *None of the local checkpoints can ever be involved in a z-cycle.*

Proof Let $C_{i,k}$ be a local checkpoint and W be its sequence number. Let us suppose, by the way of contradiction, that $C_{i,k}$ is involved in a z–cycle consisting of messages $M_1, M_2, \ldots, M_h$. From the definition of z-cycle (see the second section) and from Lemma 1, the following inequality holds: $M_h.SN \geq M_{h-1}.SN \ldots \geq M_1.SN \geq W$. By the definition of z–cycle, the receipt of message M_h occurs before $C_{i,k}$ with $C_{i,k}.SN = W$. Due to Lemma 1, this is not possible, so the assumption is contradicted and the claim follows. □

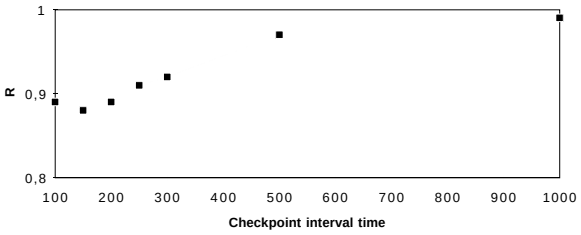

Figure 5.4 Ratio R vs. the checkpoint interval time (in time units)

From lemma 1 trivially follows that a global consistent checkpoint is formed by local checkpoints, one for each process, with the same sequence number (if there is a jump in the sequence number the first checkpoint with greater sequence number has to be included).

Simulation results

We report a quantitative comparison between our algorithm (hereafter QCB) and the one proposed in [9] (hereafter MS). The results have been obtained by simulating a distributed application consisting of 10 identical processes. Each process performs internal, send and receive operations with probability $p_i = 0.8$, $p_s = 0.1$ and $p_r = 0.1$, respectively.

The *time to execute a statement* in a process and *the message propagation time* are exponentially distributed with mean value equal to 1 and 10 time units respectively. Each process selects the destination of a message as a uniformly distributed random variable.

As we are interested in counting how many local states would be selected by one algorithm, the overhead due to checkpoint insertion has not been considered (i.e., a checkpoint is istantaneous). Each run simulates 100000 time units.

Figure 5.4 plots the ratio R between the total number of checkpoints N_{QCB} taken by QCB and the total number of checkpoints N_{MS} taken by MS versus the checkpoint interval time T of the processes [1]. The algorithms perform the same with large checkpoint interval time T. On the other hand for small values of T, QCB performs better. Two reasons lead to such a behavior. For small checkpoint interval times, a few receive events occur in each checkpoint interval increasing so the probability of equivalent local checkpoints which leads to a reduction of forced checkpoints. Moreover, the smaller the checkpoint interval time is, the higher is the probability that rule R2".b applies. i.e., no send event has occurred in a checkpoint interval before the receive of a message piggybacking a sequence number grater than the local one.

A RECOVERY SCHEME

Our recovery scheme is similar to the one proposed in [9]. The major difference is that, in some circumstances, our scheme does not force processes to take a forced checkpoint to be incuded in the recovery line. The scheme is fully

asynchronous: the failed process informs the other processes about its failure, and resumes its computation from its last checkpoint without waiting for any acknowledgment.

Data structures and process behavior. The recovery scheme requires, as well as the variables defined in the fourth section, other two local varables (for the sake of clarity we adopt the same notation as in [9]):

INC_i, REC_LINE_i: **integer**.

INC_i represents the number of recoveries experienced by process P_i (incarnations) since the beginning of its execution. REC_LINE_i indicates, upon rolling back, the sequence number of the local checkpoint from which the computation must be resumed. INC_i and REC_LINE_i are initialized to zero.

The values of variables SN_i, INC_i and REC_LINE_i are recorded on stable storage so that they are not lost in case of process failure. Each application message M piggybacks three integers: a copy of SN_i ($M.SN$), INC_i ($M.INC$) and REC_LINE_i ($M.REC\_LINE$).

In Figures 5.4 and 5.5 we report the procedures that explain the behavior of a process. The procedure *WHEN a basic checkpoint $C_{i,k}$ is scheduled* is the same that the one in the fourth section, therefore, it has been omitted.
As in [9], if P_i fails it restores its latest checkpoint (i.e., the one with sequence number SN_i), increases INC_i by one and sets $REC\_LINE_i = SN_i$. Then, it broadcasts the $rollback(INC_i, REC\_LINE_i)$ message to all the other processes.

Upon receiving the $rollback(INC_i, REC\_LINE_i)$ message, P_j (with $j \neq i$) behaves as follows: if $INC_i > INC_j$ then a rollback procedure is executed (P_j is not aware of the recovery with incarnation number INC_i). After the rollback procedure, P_j restarts the computation without waiting the rollback phase termination of the other processes. For this reason a process P_k, before receiving the $rollback(INC_i, REC\_LINE_i)$ message, can receive a message M sent by P_j after its rollback procedure. This message carries a value $M.INC > INC_k$ which forces P_k to start the recovery action at the same way as when the $rollback(INC_i, REC\_LINE_i)$ message arrives. When P_k receives the rollback message with the same incarnation number as $M.INC$ it skips the message.

We focus our attention on the explanation of the *rollback* procedure which slightly differs from the one presented in [9]. We introduced some modifications with the aim of reducing the number of forced checkpoints in the rollback phase.

Suppose P_i fails, upon executing the rollback procedure P_j compares the value of either REC_LINE_i, received with the rollback message from P_i, or $M.REC\_LINE$, received with some application message, with its own checkpoint sequence number SN_j. If $REC\_LINE_i > SN_j$ (or $M.REC\_LINE > SN_j$), then P_j has not to rollback because it has not taken the checkpoint belonging to the recovery line with number REC_LINE_i (or $M.REC\_LINE$). In such a case a forced checkpoint, with sequence number REC_LINE_i (or

```
init SN_i := 0; RN_i := -1; send_i := FALSE; recv_i := FALSE; skip_i := FALSE;
     INC_i := 0; REC_LINE_i := 0;

procedure SEND(M, P_j):                    % M is the message, P_j is the destination %
begin
      M.SN := SN_i;
      M.INC := INC_i;
      M.REC_LINE := REC_LINE_i;
      send (M) to P_j;
      send_i := TRUE
end.

procedure STARTING-RECOVERY-AFTER-FAILURE:                    % P_i starts the recovery %
begin
      restore the last checkpoint C_{i,k};
      SN_i := C_{i,k}.SN;
      INC_i := INC_i + 1;                                % update the incarnation number %
      REC_LINE_i := SN_i;                                  % set the recovery line number %
      send rollback(INC_i, REC_LINE_i) to all the other processes
end.

when (M) arrives at P_i in I_{i,k-1}:
begin
      if M.INC > INC_i
      then
           begin
           REC_LINE_i := M.REC_LINE;                     % set the recovery line number %
           INC_i := M.INC;                                 % set the incarnation number %
           ROLL_BACK(P_i)                                % execute the roll back procedure %
           end;
      if (M.SN > SN_i and send_i)                                          % see rule R2".a %
      then
           begin
           take a forced checkpoint C_{i,k};
           SN_i := M.SN;                                   % update the sequence number %
           RN_i := M.SN;
           C_{i,k}.SN := SN_i;
           send_i := FALSE;
           skip_i = TRUE;
           end
      else
          if M.SN > SN_i                                                   % see rule R2".b %
          then
               begin
               SN_i := M.SN;                               % update the sequence number %
               RN_i := M.SN;
               C_{i,k-1}.SN := SN_i;   % update the sequence number of the last checkpoint %
               end
          else
              if M.SN > RN_i then RN_i := M.SN;
      recv_i := TRUE;
      process the message
end.
```

Figure 5.5 The checkpointing-recovery scheme (part i)

```
when rollback(INC_j, REC_LINE_j) arrives at P_i in I_{i,k-1}:
begin
      if INC_j > INC_i
      then
           begin
           INC_i := INC_j;                          % set the incarnation number %
           REC_LINE_i := REC_LINE_j;                % set the recovery line number %
           ROLL_BACK(P_i);                          % execute the roll back procedure %
           continue as normal
           end
      else skip the rollback message
end.

procedure ROLL_BACK(P_i):
begin
      if (REC_LINE_i > SN_i)                                  % no need to rollback %
      then
           begin
                SN_i := REC_LINE_i;
                if send_i
                then
                     begin
                     take a forced checkpoint C_{i,k};
                     C_{i,k}.SN := REC_LINE_i;
                     send_i := FALSE;                                  % reset flags %
                     recv_i := FALSE;
                     end
                else C_{i,k-1}.SN := REC_LINE_i;          % assign the sequence number %
           end
      else
          begin
          find the erliest checkpoint C_{i,h} with C_{i,h}.SN ≥ REC_LINE_i;
          SN_i := C_{i,h}.SN;                                % set the sequence number %
          restore checkpoint C_{i,h} and delete all checkponts C_{i,x} with x > h;
          send_i := FALSE;                                                % reset flags %
          recv_i := FALSE;
          end
end.
```

Figure 5.6 The checkpointing-recovery scheme (part ii)

$M.REC\_LINE$), is taken only if a send event occurred in the current checkpoint interval.

Correctness proof

Observation 1
Suppose process P_i fails and restores to checkpoint C_{i,x_i} with sequence number equal to REC_LINE_i.
The possible behaviors of process P_j (with $j \neq i$), upon receiving either the $rollback(INC_i, REC\_LINE_i)$ message with $INC_i > INC_j$ or an application message with $M.INC > INC_j$, are the following:
(a) $SN_j \geq REC\_LINE_i$: in this case P_j rolls back to its earliest checkpoint C_{j,x_j} such that $C_{j,x_j}.SN \geq REC\_LINE_i$ and sets $SN_j = C_{j,x_j}.SN$
(b) $SN_j < REC\_LINE_i$ and $send_j = TRUE$: in this case P_j takes a checkpoint C_{j,x_j}, then sets $SN_j = REC\_LINE_i$ and $C_{j,x_j}.SN = REC\_LINE_i$

(c) $SN_j < REC\_LINE_i$ and $send_j = FALSE$: in this case P_j sets the sequence number of its last checkpoint C_{j,x_j} to the value $C_{j,x_j}.SN = REC\_LINE_i$ and sets $SN_j = REC\_LINE_i$.

In any case, after the rollback phase, P_j has a checkpoint with sequence number $C_{j,x_j}.SN \geq REC\_LINE_i$. We say that P_j rolls back to C_{j,x_j} either if it behaves as in (a), or in (b), or in (c).

Observation 2
All checkpoints taken by P_j before C_{j,x_j} have sequence numbers less than or equal to $C_{j,x_j}.SN$ (they have sequence numbers equal to $C_{j,x_j}.SN$ only if they are equivalent to C_{j,x_j}).

Observation 3
For any message M sent by P_j: if $send(M) \in I_{j,x}$ (with $x < x_j$) then $M.SN \leq C_{j,x_j}.SN$ and vice versa.

Observation 4
For any message M received by P_j: if $receive(M) \in I_{j,x}$ (with $x < x_j$) then $M.SN < C_{j,x_j}.SN$

Observation 5
For any j, P_j receives and processes a message M only after a checkpoint C_{j,x_j} such that $C_{j,x_j}.SN \geq M.SN$.

Theorem 3 *Suppose process P_i broadcasts the rollback(INC_i, REC_LINE_i) message and for all $j \neq i$ process P_j rolls back to checkpoint C_{j,x_j}, then the set*

$$S = (C_{1,x_1}, C_{2,x_2}, \ldots, C_{n,x_n})$$

where $C_{i,x_i}.SN = REC\_LINE_i$, is a recovery line.

Proof

Suppose set S is not a recovery line. Then, there exists a message M, sent by some process P_j to a process P_k, that is orphan with respect to the pair (C_{j,x_j}, C_{k,x_k}). From observations 4, 3 and 1

$$C_{k,x_k}.SN > M.SN \geq C_{j,x_j}.SN \geq REC\_LINE_i$$

Since $M.SN \geq REC\_LINE_i$, P_k receives and processes M only after a checkpoint with sequence number larger than or equal to REC_LINE_i (observation 5).

Since $receive(M)$ happens before C_{k,x_k}, there exists a checkpoint C_{k,x_k-x} such that $C_{k,x_k-x}.SN \geq REC\_LINE_i$. Inequality $C_{k,x_k-x}.SN > C_{k,x_k}.SN$ never holds (observation 2), on the other hand, if $C_{k,x_k-x}.SN = C_{k,x_k}.SN$, $C_{k,x_k-x} \equiv_{\mathcal{L}_{SN}} C_{k,x_k}$ (observation 2) and so M cannot be received before C_{k,x_k}, i.e., it cannot be orphan with respect to the pair (C_{j,x_j}, C_{k,x_k}). In any case the assumption is contradicted.

□

CONCLUSION

In this paper we presented a communication-induced checkpointing–recovery scheme for distributed applications with asynchronous cooperating processes. It

consists of a z–cycle–free checkpointing algorithm and an asynchronous recovery scheme.

The proposed checkpointing algorithm ensures the progression of the recovery line while reducing the number of checkpoints compared to previous proposals. We achieved this goal by introducing an equivalence relation between local checkpoints (and a rule to track on line this relation) and by exploiting information about the events' history of a process.

The recovery algorithm is fully asynchronous and does not require a vector of timestamps for tracking dependencies between checkpoints. We have also shown experimental results which quantify, the reduction of the number of local checkpoints in a distributed execution.

Notes

1. To avoid fully synchronous checkpointing activities we shift the action to take the first basic checkpoint of each process of a value uniformly ditributed between 0 and T.

References

[1] A. Acharya and B.R. Badrinath. *Checkpointing Distributed Application on Mobile Computers.* In *3-th International Conference on Parallel and Distributed Information Systems Proc.*, pages 73-80, Austin, Texas, 1994.

[2] R. Baldoni, J.M. Helary, A. Mostefaoui and M. Raynal. On Modeling Consistent Checkpoints and the Domino Effect in Distributed Systems. *Technical Report No.2569, INRIA, France*, 1995.

[3] R. Baldoni, J.M. Helary, A. Mostefaoui and M. Raynal. A Communication-Induced Checkpointing Protocol that Ensures Rollback-Dependency Trackability. In *IEEE Int. Symposium on Fault Tolerant Computing Proc.*, pages 68-77, 1997.

[4] D. Briatico, A. Ciuffoletti and L.Simoncini. *A Distributed Domino-Effect Free Recovery Algorithm.* In *4-th IEEE symp. on Reliability Distr. Software and Database Proc.*, pages 207-215, 1984.

[5] K.M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Computer Systems*, 3(1):63–75, 1985.

[6] E.N. Elnozahy, D.B. Johnson and Y.M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *Technical Report No. CMU-CS-96-181, School of Computer Science, Carnegie Mellon University*, 1996.

[7] R. Koo and S. Toueg. Checkpointing and Rollback- Recovery for Distributed Systems. *IEEE Trans. on Software Engineering*, 13(1):23–31, 1987.

[8] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558–565, 1978.

[9] D. Manivannan and M. Singhal. A Low-overhead Recovery Technique Using Quasi-Synchronous Checkpointing, *in Proc. IEEE INT. Conf. Distributed Comput. Syst.*, pages 100-107, 1996.

[10] D. Manivannan and M. Singhal. Quasi-Synchronous Checkpointing: Models, Characterization, and Classification. *Technical Report No.OSU-CISRC-5/96-TR33, Dept. of Computer and Information Science, The Ohio State University*, 1996.

[11] R.H.B. Netzer and J. Xu. Necessary and Sufficient Conditions for Consistent Global Snapshots. *IEEE Trans. on Parallel and Distributed Systems*, 6(2):165–169, 1995.

[12] B. Randell. System Structure for Software Fault Tolerance. *IEEE Trans. on Software Engineering*, SE1(2):220–232, 1975.

[13] R.D. Schlichting and F.B. Schneider. Fail-Stop Processors: an Approach to Designing Fault-Tolerant Computing Systems. *ACM Trans. on Computer Systems*, 1(3):222–238, 1983.

[14] R.E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Trans. on Computer Systems*, 3(2):204–226, 1985.

[15] K. Venkatesh, T. Darhakrishnan and F. Li. Optimal Checkpointing and Local Encoding for Domino–Free Rollback Recovery. *Information Processing Letters*, 25:295–303, 1987.

[16] Y.M. Wang. Consistent Global Checkpoints that Contains a Set of Local Checkpoints. *IEEE Trans. on Computers*, 46(4):456–468, 1997.