Preventing Useless Checkpoints in Distributed Computations

Jean-Michel HELARY* Achour MOSTEFAOUI* Robert H.B. NETZER† Michel RAYNAL*

Abstract

A useless checkpoint is a local checkpoint that cannot be part of a consistent global checkpoint. This paper addresses the following important problem. Given a set of processes that take (basic) local checkpoints in an independent and unknown way, the problem is to design a communication-induced checkpointing protocol that directs processes to take additional local (forced) checkpoints to ensure that no local checkpoint is useless.

A general and efficient protocol answering this problem is proposed. It is shown that several existing protocols that solve the same problem are particular instances of it. The design of this general protocol is motivated by the use of communication-induced checkpointing protocols in "consistent global checkpoint"-based distributed applications. Detection of stable or unstable properties, rollback-recovery, and determination of distributed breakpoints are examples of such applications.

1 Introduction

A *local checkpoint* is a snapshot of a local state of a process, a *global checkpoint* is a set of local checkpoints, one from each process, and a *consistent global checkpoint* is a global checkpoint such that no message sent by a process after its local checkpoint is received by another process before its local checkpoint. So, the consistency of global checkpoints strongly depends on the flow of messages exchanged by processes. The determination of consistent global checkpoints is a fundamental problem in distributed computing and arises in many applications such as detection of stable properties [5, 11], determination of breakpoints [9, 18], detection of unstable properties [2, 6, 10, 13], rollback recovery upon failure occurences [7, 14, 20], etc.

When processes independently take their local checkpoints there is a risk that no consistent global checkpoint can ever be formed (except the first one composed of their initial states). This is caused by the well-known *unbounded domino effect* [20]. Even if consistent global checkpoints can be formed, it is still possible that some local checkpoints can never be included in a consistent global checkpoint; such local checkpoints are called *useless*.

To prevent useless checkpoints, and thus safely prevent the domino effect, some coordination in the taking of local checkpoints is required. In the family of *coordinated* protocols [5, 15], processes use additional control messages to synchronize their checkpointing activities. This additional synchronization may result in reduced process autonomy and degraded performance of the underlying application. These drawbacks have given rise to the development of a family of *communication-induced* checkpointing protocols. In this family the coordination is achieved by piggybacking control information on application messages: no control messages or synchronization is added to the application [7]. More precisely, processes take local checkpoints independently[1] (called *basic* checkpoints) and the protocol directs them to take additional local checkpoints (called *forced* checkpoints) to ensure that no local checkpoint becomes useless. Taking a forced checkpoint before each message delivery is a safe strategy to prevent useless checkpoints but is very inefficient. Given a set of basic checkpoints, the fewer the forced checkpoints are taken by a communication-induced checkpointing protocol, the better the protocol. A process decides whether to take a forced checkpoint when a message is received by evaluating a predicate. This predicate is based on local control variables of the receiving process and on control values carried by the message. The local control variables managed by a process are a coding of the causal dependencies appearing in its past. Distinct semantics for these control variables and distinct definitions of the predicate give rise to different protocols [1, 3, 4, 14, 17, 21, 22, 24].

In this paper, we present a new communication-induced checkpointing protocol that takes as few forced checkpoints as possible while ensuring no local checkpoint is useless. This protocol is based on the Z-path (and Z-cycle) theory introduced by Netxer and Xu [19] who showed that a useless checkpoint exactly corresponds to the existence of a Z-cycle in the distributed computation. At the model level, our protocol prevents Z-cycles. At the operational level,

*IRISA, Campus de Beaulieu, Université Rennes1, Rennes, France. {helary,mostefaoui,raynal}@irisa.fr.

†Computer Science Department, Brown University, Box 1910, Providence, RI 02921, USA. rn@cs.brown.edu.

[1] For example, in the detection of unstable properties such as conjunctions of local predicates, each process takes a basic checkpoint each time its local predicate becomes true [13].

a sequence number and Lamport timestamp are associated with each local checkpoint. Moreover, each message piggybacks three vectors of size n (one including checkpoint sequence numbers, one including Lamport timestamps, and the last including boolean values; n is the number of processes). This protocol is more efficient[2] than past domino-free communication-induced checkpointing protocols. An interesting feature of the proposed protocol is the following one: for any local checkpoint A, there is a very easy determination of a consistent global checkpoint to which A belongs. Moreover, the proposed protocol enjoys a nice genericity property: if we reduce the size of its control information, or even eliminate some of it altogether, the protocol reduces to already known protocols such as [1, 17, 21]. As a result, our protocol offers a very general and efficient framework for a family of domino-free communication-induced checkpointing protocols.

The paper is divided into five sections. Section 2 presents the model of distributed computations, provides a definition for consistent global checkpoints, and defines Z-paths. Section 3 presents the protocol. Section 4 discusses the protocol and shows that it reduces to existing protocols when reducing its control information. Finally Section 5 concludes the paper.

2 Distributed Computations, Checkpoints and Z-Paths

2.1 Distributed Computations

A distributed computation consists of a finite set P of n processes $\{P_1, P_2, \dots, P_n\}$ that communicate and synchronize only by exchanging messages. We assume that each ordered pair of processes is connected by an asynchronous, reliable, directed logical channel whose transmission delays are unpredictable but finite. (Note that channels are not required to be FIFO.) Each process runs on a different processor, processors do not share a common memory, and there is no bound on their relative speeds. Also, they fail according to the fail-stop model.

A process can execute internal, send[3] and delivery statements. An internal statement does not involve communication. When P_i executes the statement *"send(m) to P_j"* it puts the message m into the channel from P_i to P_j. When P_i executes the statement *"deliver(m)"*, it is blocked until at least one message directed to P_i has arrived, then a message is withdrawn from one of its input channels and delivered to P_i. Executions of internal, send and delivery statements are modeled by internal, sending and delivery events.

[2] When considering the number of forced local checkpoints taken by processes.

[3] We assume a process does not send messages to itself.

Processes of a distributed computation are *sequential*; in other words, each process P_i produces a *sequence* of events $e_{i,1} \dots e_{i,s} \dots$ This sequence can be finite or infinite. Every process P_i has an initial local state denoted $\sigma_{i,0}$. The local state $\sigma_{i,s}$ ($s > 0$) results from the execution of the sequence $e_{i,1} \dots e_{i,s}$ applied to the intial state $\sigma_{i,0}$. More precisely, the event $e_{i,s}$ moves P_i from the local state $\sigma_{i,s-1}$ to the local state $\sigma_{i,s}$. By definition, we say that "$e_{i,x}$ *belongs to* $\sigma_{j,s}$" (sometimes denoted as $e_{i,x} \in \sigma_{j,s}$) if $i = j$ and $x \leq s$.

Let H be the set of all the events produced by a distributed computation. This computation is modeled by the partially ordered set $\widehat{H} = (H, \xrightarrow{hb})$, where $\xrightarrow{hb}$ denotes the well-known Lamport's *happened-before* relation [16].

2.2 Local and Global Checkpoints

Local checkpoints. A *local checkpoint* C is a recorded state (snapshot) of a process. Not every local state is necessarily recorded as a local checkpoint, so the set of local checkpoints is only a subset of the set of local states.

Definition 2.1 *A* communication and checkpoint pattern *is a pair* $(\widehat{H}, \mathcal{C}_{\widehat{H}})$ *where* $\widehat{H}$ *is a distributed computation and* $\mathcal{C}_{\widehat{H}}$ *is a set of local checkpoints defined on* $\widehat{H}$.

$C_{i,x}$ represents the x-th local checkpoint of process P_i; x is called the *index* of this checkpoint. The local checkpoint $C_{i,x}$ corresponds to some local state $\sigma_{i,s}$ with $x \leq s$. Figure 1 shows an example of a checkpoint and communication pattern[4]. We assume that each process P_i takes an initial local checkpoint $C_{i,0}$ (corresponding to $\sigma_{i,0}$), and after each event a checkpoint will eventually be taken.

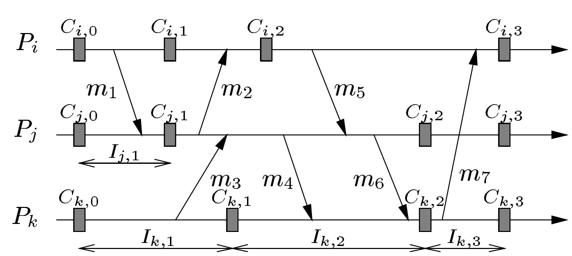

Figure 1. A Checkpoint and Communication Pattern

A message m sent by process P_i to process P_j is called *orphan* with respect to the ordered pair of local checkpoints ($C_{i,x}$,$C_{j,y}$) iff the delivery of m belongs to $C_{j,y}$ ($deliver(m) \in C_{j,y}$) while its sending event does not belong to $C_{i,x}$ (*send*$(m) \notin C_{i,x}$). An ordered pair of local

[4] This figure uses the usual space-time diagram. Local checkpoints are indicated by black rectangular boxes; the other local states are not explicitly indicated.

checkpoints is *consistent* iff there are no orphan messages with respect to this pair. For example, Figure 1 shows that the pair ($C_{k,1}$,$C_{j,1}$) is consistent, while the pair ($C_{i,2}$,$C_{j,2}$) is inconsistent (because of orphan message m_5).

Global checkpoints. A *global checkpoint* is a set of local checkpoints, one from each process. For example, $\{C_{i,1}, C_{j,1}, C_{k,1}\}$ and $\{C_{i,2}, C_{j,2}, C_{k,1}\}$ are two global checkpoints depicted in the Figure 1.

Definition 2.2 *A global checkpoint is* consistent *iff all its pairs of local checkpoints are consistent.*

For example, Figure 1 shows that $\{C_{i,1}, C_{j,1}, C_{k,1}\}$ is a consistent global checkpoint, and due to the inconsistent pair $(C_{i,2}, C_{j,2})$, the global checkpoint $\{C_{i,2}, C_{j,2}, C_{k,1}\}$ is not consistent.

2.3 Z-Paths and Z-Cycles

The sequence of events occurring at P_i between $C_{i,x-1}$ and $C_{i,x}$ $(x > 0)$ is called a *checkpoint interval* and is denoted by $I_{i,x}$ (see Figure 1). The Z-path notion, introduced for the first time by Netzer and Xu [19], generalizes the notion of a causal path of messages defined by Lamport's *happened-before* relation. More precisely :

Definition 2.3 *A* Z-path *exists from local checkpoint A to local checkpoint B if and only if A precedes B in the same process, or a sequence of messages* $[m_1, m_2, \dots, m_q]$ $(q \geq 1)$ *exists such that:*

1. *A precedes $send(m_1)$ in the same process, and*
2. *for each m_i, $i < q$, $delivery(m_i)$ is in the same or erlier interval as $send(m_{i+1})$, and*
3. *$delivery(m_q)$ precedes B in the same process.*

In Figure 1, $[m_3, m_2]$ is a Z-path from $C_{k,0}$ to $C_{i,2}$; $[m_5, m_4]$ and $[m_5, m_6]$ are two Z-paths from $C_{i,2}$ to $C_{k,2}$.

Definition 2.4 *In a Z-path $[m_1, \dots, m_q]$, two consecutive messages m_α and $m_{\alpha+1}$ form a* Z-pattern *iff $send(m_{\alpha+1}) \stackrel{hb}{\to} delivery(m_\alpha)$.*

In Figure 1, we can see that $[m_3, m_2]$ and $[m_5, m_4]$ are two Z-patterns.

Definition 2.5 *A Z-path is* causal *iff it does not include Z-patterns (i.e., the delivery event of each message (except the last) occurs before the send event of the next message in the sequence). A Z-path is* non-causal *iff it is not causal.*

A Z-path with only one message is trivially causal. Every non-causal Z-path is the concatenation of shorter causal Z-paths. In Figure 1, $[m_3, m_2, m_5, m_4, m_7]$ is a non-causal Z-path; it is the concatenation of the causal Z-paths $[m_3]$, $[m_2, m_5]$, and $[m_4, m_7]$.

Definition 2.6 *A Z-path from a local checkpoint $C_{i,x}$ to the same local checkpoint $C_{i,x}$ is called a* Z-cycle*. (We say that it* involves *the local checkpoint $C_{i,x}$.)*

The Z-path $[m_7, m_5, m_6]$ is a Z-cycle that involves the local checkpoint $C_{k,2}$. We can observe that a Z-cycle always includes a Z-pattern.

2.4 Useless Checkpoints

Definition 2.7 *A local checkpoint $C_{i,x}$ is* useless *iff it cannot belong to any consistent global checkpoint.*

The following important characterization of useless checkpoints has been stated in [19]:

Theorem 2.1 (Netzer-Xu 1995) *A local checkpoint $C_{i,x}$ is* useless *iff it is involved in a* Z-cycle.

For example in Figure 1, $C_{k,2}$ is useless. The Z-path $[m_7, m_5, m_6]$ is a Z-cycle including $C_{k,2}$. It includes the Z-pattern $[m_7, m_5]$. The interested reader will find a proof of this theorem in [19] and in [23].

3 The Protocol

The set $\mathcal{C}_{\widehat{H}}$ of checkpoints taken during the execution of a computation $\widehat{H}$ is composed of basic checkpoints and forced checkpoints. As indicated in the Introduction, why and when a basic checkpoint is taken depends only on the application (such as for a property detection protocol or a rollback-recovery protocol). Forced checkpoints are taken by the communication-induced checkpointing protocol to ensure that no checkpoint is useless. The aim of the protocol we wish to design is to keep low the number of forced checkpoints. The protocol works by evaluating a predicate upon every message reception and possibly taking a forced checkpoint (hence the name *communication-induced checkpointing protocol*). The predicate is based on past knowledge of the communication and checkpoint patterns. Forced checkpoints are taken to prevent Z-cycles from forming.

3.1 Basic Idea: A Checkpoint Timestamping Mechanism

With each checkpoint C, let us associate a timestamp denoted $C.t$. We consider in the following that the domain of timestamp values is the set of positive integers. The protocol is based on the following theorem.

Theorem 3.1 *If for any pair of checkpoints $C_{j,y}$ and $C_{k,z}$, such that there is a Z-path from $C_{j,y}$ to $C_{k,z}$, we have $C_{j,y}.t < C_{k,z}.t$, then no checkpoint can be involved in any Z-cycle.*

Proof Suppose that a Z-cycle exists from $C_{i,x}$ to $C_{i,x}$. This Z-cycle is a Z-path from $C_{i,x}$ to $C_{i,x}$. From the assumption of the theorem, this would imply $C_{i,x}.t < C_{i,x}.t$. $\square_{Theorem\ 3.1}$

The idea underlying this theorem is that, if we can design a protocol that manages timestamps and takes forced checkpoints in such a way that timestamps *always* increase along any Z-path, then no Z-cycles can possibly form, and no checkpoints will be useless. We assume each process P_i has a local logical clock lc_i managed in the following classical way [16]:

- Before it takes a (basic or forced) checkpoint, P_i increases by 1 its local clock (and associates the new value with the checkpoint).
- Every message m is timestamped with the value of its sender clock (let $m.t$ be the timestamp associated with m).
- When a process P_i receives a message, m updates its local clock $lc_i := \max(lc_i, m.t)$.

It follows from this classical mechanism that, if there is a *causal* Z-path from $C_{j,y}$ to $C_{k,z}$, then we have $C_{j,y}.t < C_{k,z}.t$. We examine now the case of *non-causal* Z-paths.

3.2 To Checkpoint or Not to Checkpoint?

Given the previous timestamping mechanism, let us consider the situation depicted in Figure 2.a where $C_{j,y}$ is a local checkpoint taken by P_j before sending m_1 and $C_{k,z}$ is the *first* checkpoint of P_k taken after the delivery of m_2. As the sending of m_2 and the delivery of m_1 belong to the same interval of P_i, it follows that $[m_1, m_2]$ constitutes a Z-pattern from $C_{j,y}$ to $C_{k,z}$.

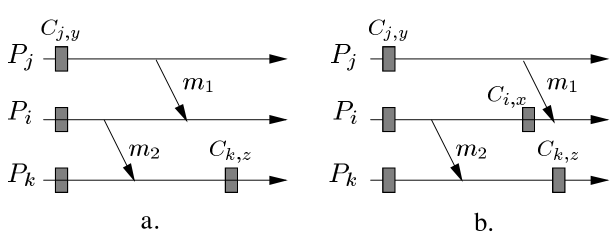

Figure 2. Must P_i Take a Forced Checkpoint?

When P_i receives m_1, two cases can occur:

- $m_1.t \le m_2.t$: In that case, $C_{j,y}.t \le m_1.t \le m_2.t < C_{k,z}$. Hence, the Z-pattern $[m_1, m_2]$ is consistent with the assumption of Theorem 3.1.
- $m_1.t > m_2.t$: In this case, a safe strategy to prevent Z-cycle formation consists of directing P_i to take a forced checkpoint $C_{i,x}$ before delivering m_1 (as shown in Figure 2.b). This "breaks" $[m_1, m_2]$ so it is no longer a Z-pattern. This strategy can be implemented in the following way. Each process P_i manages a boolean array $sent\_to_i[1..n]$ in order to know whether the reception of a message creates a Z-pattern; $sent\_to_i[k]$ has the value $true$ iff P_i has sent a message to P_k since its last checkpoint. Moreover, P_i manages an array of integers $min\_to_i[1..n]$; min_to = FLS
$min\_to_i[k]$ keeps the timestamp of the first message P_i sent to P_k since P_i's last checkpoint. The condition $m_1.t > m_2.t$ is then expressed as:

$$\mathcal{C} \equiv (\exists k : \ sent\_to_i[k] \wedge m_1.t > min\_to_i[k])$$

So, P_i takes a forced checkpoint if $\mathcal{C}$ is true. The next section shows how this safe strategy can be improved by sharpening the predicate which will cause fewer forced checkpoints to be taken.

3.3 Reducing the Number of Forced Checkpoints

The previous strategy does not utilize the information that P_i could have concerning the values of local clocks of the other processes. For each $k (1 \le k \le n)$, let us denote by $cl_i(k)$ the value of P_k's local clock as perceived by P_i (P_i can obtain this knowledge with a classical piggybacking technique as will be shown in Section 3.4). If $k = i$, obviously $cl_i(i) = lc_i$. However, if $k \neq i$, the perception of P_k's local clock by P_i is only an approximation such that $cl_i(k) \le lc_k$. Consider again the situation depicted in Figure 2.a, where message m_1 arrives from P_j, creating a Z-pattern with message m_2 sent to[5] P_k. If the following property holds

$$(m_1.t \le m_2.t) \vee \mathcal{P}, \text{ where}$$
$$\mathcal{P} \equiv (C_{j,y}.t \le m_1.t \le cl_i(k) < C_{k,z}.t)$$

then the Z-pattern $[m_1, m_2]$ is consistent with the assumption of Theorem 3.1. Let us consider the property $\mathcal{P}$ in the case where $m_1.t > m_2.t$. Since $m_1.t$ carries the value of lc_j when m_1 is sent, the first relation $C_{j,y}.t \le m_1.t$ necessarily holds when m_1 is received. So, the property $\mathcal{P}$ can be violated only if, when m_1 is received, $m_1.t > cl_i(k)$ or if $cl_i(k) \ge C_{k,z}.t$. It follows that to prevent the formation of the Z-pattern $[m_1, m_2]$ that would violate property $\mathcal{P}$ (and consequently that could possibly be inconsistent with the assumptions of Theorem 3.1), the protocol requires P_i to take a forced checkpoint before delivering m_1 to the application if $m_1.t > cl_i(k)$ or if $cl_i(k) \ge C_{k,z}.t$.

The question now is to determine to which value of cl_k the approximation $cl_i(k)$ refers. Let us examine the two possible cases.

[5] Recall that $C_{k,z}$ is the first checkpoint taken by P_k after the delivery of m_2.

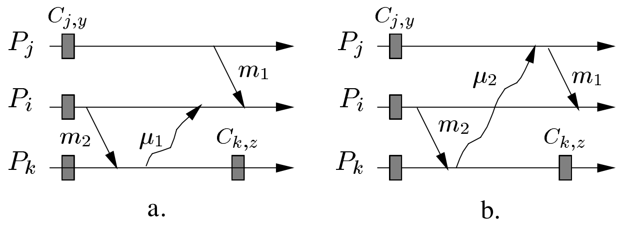

Figure 3. $cl_i(k)$ is a Lower Bound of $C_{k,z}.t$

• The value $cl_i(k)$ has been brought to P_i by a causal Z-path that started from P_k before $C_{k,z}$. This situation is illustrated in Figure 3 (more precisely, $cl_i(k)$ is brought to P_i by μ_1 in Figure 3.a and by $\mu_2 \cdot [m_1]$ in Figure 3.b). In that case we have $cl_i(k) < C_{k,z}.t$ and, consequently, P_i has to take a forced checkpoint only if $m_1.t > cl_i(k)$.

• The value $cl_i(k)$ has been brought to P_i by a causal Z-path that started from P_k after $C_{k,z}$. This situation is illustrated in Figure 4 (more precisely, the relevant causal Z-path is μ_1 in Figure 4.a and is $\mu_2 \cdot [m_1]$ in Figure 4.b; note that both figures can be redrawn as indicated in Figure 5 where μ, the causal Z-path that brings to P_i the last value of P_k's local clock, is μ_1 or $\mu_2 \cdot [m_1]$). In that case we have $cl_i(k) \geq C_{k,z}.t$. This exactly corresponds to the pattern described in Figure 5. So, the problem for P_i is to recognize this pattern and take a forced checkpoint if it occurs. Let $\mathcal{C}_1$ be a predicate describing this pattern occurrence.

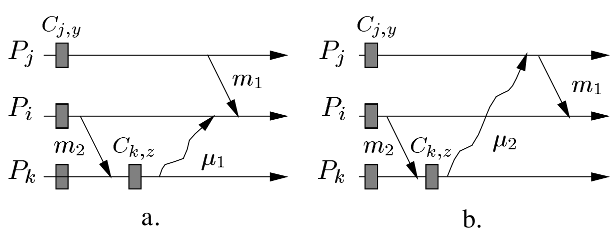

Figure 4. $cl_i(k)$ is not a Lower Bound of $C_{k,z}.t$

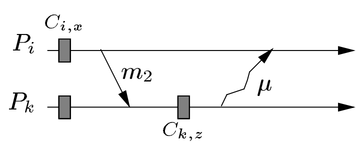

Figure 5. A Causal Z-Path

From this discussion it follows that the previous condition $\mathcal{C}$ (which P_i tests to know if it should take a forced checkpoint when it receives a message m_1) can be refined into $\mathcal{C}'$:

$$\mathcal{C}' \equiv (\exists k : \; sent\_to_i[k] \wedge (m_1.t > min\_to_i[k]) \wedge (m_1.t > cl_i(k) \vee \mathcal{C}_1))$$

The next section shows how to express this predicate with appropriate data structures so it can be evaluated on-line by each process.

3.4 Data Structures

In addition to the arrays $sent\_to_i[1..n]$ and $min\_to_i[1..n]$, every process P_i maintains the following data structures.

Array $clock_i$**.** Each process P_i manages an array $clock_i[1..n]$ with the following meaning: $clock_i[k]$ = highest value of lc_k known by P_i (note that $clock_i[i]$ is lc_i and so we do not require lc_i in the following). This array is initialized to $(0, \ldots, 0)$ and managed as follows[6]:

• When it takes a (basic or forced) checkpoint, P_i increments $clock_i[i]$ by 1 (definition of lc_i).

• When P_i sends a message m, the current value of $clock_i$ is appended to m (let it be $m.clock$).

• When P_i receives m from P_j, it performs the following updates:

- $clock_i[i] := \max(clock_i[i], m.clock[j])$ (since $m.clock[j]$ is P_j's Lamport clock; this statement updates P_i's Lamport clock).
- $\forall k \neq i : clock_i[k] := \max(clock_i[k], m.clock[k])$ (note that $\forall k : \; clock_i[i] \geq clock_i[k]$).

Using this data structure, when P_i receives a message m_1, we have $cl_i(k) = \max(clock_i[k], m_1.clock[k])$. Thus, with these elements, $\mathcal{C}'$ can be rewritten as

$$\begin{aligned}\mathcal{C}' \equiv (\exists k :& sent\_to_i[k] \wedge \\ & (m_1.clock[j] > min\_to_i[k]) \wedge \\ & ((m_1.clock[j] > max(clock_i[k], m_1.clock[k])) \vee \mathcal{C}_1) \\ )&\end{aligned}$$

The next two arrays provide a way to evaluate $\mathcal{C}_1$.

Array $ckpt_i$**.** This array is a vector clock that counts how many checkpoints have been taken by each process. So, $ckpt_i[k]$ = number of checkpoints taken by P_k to P_i's knowledge. This vector clock is managed in the usual way [8]. Let $m.ckpt$ be the value appended to m by its sender P_i (i.e., the value of the array $ckpt_i$ at sending time).

Array $taken_i$**.** This boolean array is used in conjunction with $ckpt_i$ to evaluate $\mathcal{C}_1$. It has the following meaning: $taken_i[k]$ is true iff there is a causal Z-path from the last checkpoint of P_k known by P_i to the next checkpoint of P_i, *and* this causal Z-path includes a checkpoint. It is managed in the following way:

• When P_i takes a checkpoint, it sets to *true* all its entries except the i-th one ($taken_i[i]$ always remains false): $\forall k \neq i : taken_i[k] := true$.

• When it sends a message, P_i appends to it the current value of $taken_i$ (let $m.taken$ be this value).

[6]Note that $clock_i$ is a vector containing Lamport timestamps.

```
procedure take_checkpoint is
    ∀k do sent_to_i[k] := false enddo;
    ∀k do min_to_i[k] := +∞ enddo;
    ∀k ≠ i do taken_i[k] := true enddo;
    clock_i[i] := clock_i[i] + 1;
    save the current local state with a copy of clock_i[i];
    % Let C_{i,x} be this checkpoint. We have C_{i,x}.t = clock_i[i] %
    ckpt_i[i] := ckpt_i[i] + 1;

(S0) initialization
        ∀k do clock_i[k] := 0; ckpt_i[k] := 0 enddo;
        taken_i[i] := false;
        take_checkpoint;

(S1) when P_i sends a message to P_k
        sent_to_i[k] := true; min_to_i[k] := min(min_to_i[k], clock_i[i]);
        send(m, clock_i, ckpt_i, taken_i) to P_k;

(S2) when P_i receives (m, clock, ckpt, taken) from P_j
        % m.clock[j] is the Lamport's timestamp of m (i.e., m.t) %
        if (∃k : sent_to_i[k]  ∧
                (m.clock[j] > min_to_i[k])  ∧
                ((m.clock[j] > max(clock_i[k], m.clock[k]))  ∨  (m.ckpt[i] = ckpt_i[i] ∧ m.taken[i])))
          then take_checkpoint % forced checkpoint %
        endif;
        clock_i[i] := max(clock_i[i], m.clock[j]); % update of the scalar clock lc_i ≡ clock_i[i] %
        ∀k ≠ i do clock_i[k] := max(clock_i[k], m.clock[k]);
                    case m.ckpt[k] < ckpt_i[k] → skip
                          m.ckpt[k] > ckpt_i[k] → ckpt_i[k] := m.ckpt[k]; taken_i[k] := m.taken[k]
                          m.ckpt[k] = ckpt_i[k] → taken_i[k] := taken_i[k] ∨ m.taken[k]
                    end case
                 end do ;
        deliver(m)
```

Figure 6. The Protocol

- When it receives m, P_i updates $taken_i$ in the following way in order to maintain its meaning:

$$\begin{aligned}
\forall k \neq i \ \textbf{do case}\ & m.ckpt[k] < ckpt_i[k] \rightarrow skip \\
& m.ckpt[k] > ckpt_i[k] \rightarrow \\
& \qquad taken_i[k] := m.taken[k] \\
& m.ckpt[k] = ckpt_i[k] \rightarrow \\
& \qquad taken_i[k] := taken_i[k] \vee m.taken[k] \\
\textbf{end docase} &
\end{aligned}$$

With these data structures, the condition $\mathcal{C}_1$ can be expressed in the following way:

$$\mathcal{C}_1 \equiv (m_1.ckpt[i] = ckpt_i[i]) \wedge m_1.taken[i]$$

When considering Figure 5, the first part of condition $\mathcal{C}_1$ states that there is a causal Z-path (namely, $[m_2] \cdot \mu$) starting after $C_{i,x}$ and arriving at P_i before $C_{i,x+1}$, while the second part of $\mathcal{C}_1$ indicates that some process has taken a checkpoint along this causal Z-path.

3.5 The Protocol

The protocol executed by each process P_i is described in Figure 6. S0, S1 and S2 describe the initialization, the statements executed by P_i when it sends a message, and the statements it executes when it receives a message, respectively. The procedure *take_checkpoint* is called each time P_i takes a checkpoint (basic or forced). As indicated previously, why and when a basic checkpoint is taken are not part of the protocol.

3.6 A Property of the Protocol

The following theorem shows how a consistent global checkpoint $\mathcal{C}_a$ can be associated with each Lamport timestamp a. It follows that, given a local checkpoint $C_{i,x}$ timestamped a (i.e., $C_{i,x}.t = a$), $C_{i,x}$ can easily be associated with a consistent global checkpoint to which it belongs.

Theorem 3.2 *Let* a *be a Lamport timestamp* (a > 0) *and let* $\mathcal{C}_a$ *be the global checkpoint* ($C_{1,x_1}, \dots,$ C_{n,x_n}) *defined in the following way:* $\forall k$, C_{k,x_k} *is the last checkpoint of* P_k *such that* $C_{k,x_k}.t \leq$ a*. Then,* $\mathcal{C}_a$ *is a consistent global checkpoint.*

Due to space limitation, the proof (one two-column page long) is omitted. The reader interested in this proof may consult [12].

4 Discussion

This section discusses the protocol. It shows that it provides a general framework from which existing protocols can be obtained.

- Let us suppress all the data structures except the array $sent\_to_i[1..n]$ which is replaced by a single boolean $sent_i$ with the following meaning: $sent_i = (\exists k : sent\_to_i[k])$, i.e., $sent_i$ is true iff a message has been sent by P_i since its last checkpoint. The protocol becomes drastically simplified and reduces to the well-known Russell protocol [21] shown in Figure 7 (this protocol has been adapted to the context of mobile computing in [1]). Russel's protocol is characterized by the following property. When considering only *deliver*, *send* and *checkpoint* events, the behavior of each process corresponds to the following regular language:

$$(deliver^* \quad send^* \quad checkpoint)^*$$

In other words, no *deliver* event can follow immediately a *send* event. Of course, this protocol may take more forced checkpoints (and never less) than the proposed protocol.

procedure take_checkpoint **is**
  do $sent_i := false$ **enddo**;
  save the current local state as a local checkpoint;

(S0) **initialization**
  take_checkpoint;

(S1) **when** P_i **sends a message to** P_k
  $sent_i := true$; send(m) **to** P_k;

(S2) **when** P_i **receives** (m) **from** P_j
  if $sent_i$ **then** take_checkpoint % *forced checkpoint* % **endif**;
  deliver(m)

Figure 7. Russell's Protocol

- Another protocol can be obtained by considering only a subset of the data structures. For example, when we eliminate the arrays $ckpt_i[1..n]$ and $taken_i[1..n]$ and we replace the array $clock_i[1..n]$ by a single scalar $lc_i = clock_i[i]$, we obtain a protocol characterized by the following condition $\mathcal{C}''$:

$$\mathcal{C}'' \equiv \exists k : (sent\_to_i[k] \wedge (m.lc > min\_to_i[k]))$$

which may take more forced checkpoints (and never less) than the proposed protocol but requires messages to piggyback only one integer, namely, $m.lc$ (the value of lc_j at the time P_j sent m).

- We can further simplify the protocol by eliminating the array $sent\_to_i[1..n]$ and by replacing $min\_to_i[1..n]$ by a single variable min_i the value of which is the lowest timestamp used by P_i to timestamp a message since its last checkpoint, i.e., $min_i = min(min\_to_i[x], 1 \leq x \leq n)$. We then get the following condition $\mathcal{C}'''$:

$$\mathcal{C}''' \equiv (m.lc > min_i)$$

These simplifications result in the protocol shown in Figure 8, which is a variant of the protocol described in [4] and of the quasi-synchronous version proposed by Manivannan and Singhal in [17][7].

[7] The original quasi-synchronous protocol proposed in [17] differs from this variant in the following way. Each process takes its basic checkpoints according to some local logical periodicity. Moreover, min_i is reset to the current value of lc_i (instead of $+\infty$) each time a checkpoint is taken; in (S2) lc_i'update is done in the **then** part, just before calling *take_checkpoint*; finally, within the procedure *take_checkpoint*, the variable lc_i in increased only if the checkpoint is basic. This has two consequences. (1) min_i is always equal to lc_i (and therefore can be suppressed). (2) When P_i receives a message, it has to take a forced checkpoint if $m.lc > lc_i$ (whether or not it has sent a message since its last checkpoint).

procedure take_checkpoint **is**
  $lc_i := lc_i + 1$; $min_i := +\infty$;
  save the current local state with a copy of lc_i;

(S0) **initialization**
  $lc_i := 0$; take_checkpoint;

(S1) **when** P_i **sends a message to** P_k
  $min_i := min(min_i, lc_i)$;
  send(m, lc_i) **to** P_k;

(S2) **when** P_i **receives** (m, lc) **from** P_j
  if $(m.lc > min_i)$
    then take_checkpoint % *forced checkpoint* %
  endif;
  $lc_i := max(lc_i, m.lc)$;
  deliver(m)

Figure 8. A Variant of Manivannan-Singhal's Quasi-Synchronous Protocol

New protocols can be designed by considering other simplifications of the basic protocol. This discussion shows that there is a tradeoff between the number of forced checkpoints that are taken and the size of control information piggybacked by application messages. In general, the smaller the control information, the greater the number of forced checkpoints. This raises an interesting question: is the proposed protocol the optimal one, i.e., is it a communication-induced checkpointing protocol that, without *a priori* knowledge of when basic checkpoints are taken, takes the fewest number of forced checkpoints to ensure that no checkpoint is useless? This optimality question remains an open problem.

Some communication-induced checkpointing protocols use a heuristic approach to prevent useless checkpoints. In these protocols the condition tested at message reception is not safe in the sense some basic checkpoints may remain useless. A protocol of this family is described in [24]. Using experimental results, the authors show that their protocol reduces rollback distance to less than one checkpoint interval per process and the number of forced checkpoints is only 4% of the number of basic checkpoints. The proposed protocol encompasses some of these heuristic-based protocols. It is easy to show that the heuristics used in [24] is a weakening of the condition $\mathcal{C}_1 \equiv (m.ckpt[i] = ckpt_i[i] \wedge m.taken[i])$ used in the proposed protocol.

5 Conclusion

A *useless* checkpoint is a local checkpoint that cannot be part of a consistent global checkpoint. This paper has addressed the following important problem. Given a set of processes that take (*basic*) local checkpoints in an independent and unknown way, we have designed a communication-induced checkpointing protocol that directs processes to take (as few as possible) additional local (*forced*) checkpoints to ensure that no local checkpoint is useless.

Our protocol is general and efficient. It has also been shown to take fewer forced checkpoints than existing protocols solving the same problem. These improvements were obtained by using control information of two integer arrays and one boolean array. It has also been shown that the size of this control information can be reduced (or even eliminated) at the price of additional forced checkpoints. So, the protocol can easily be tuned for any desired control-information-overhead / performance tradeoff.

The design of this protocol has been motivated by the wide use of communication-induced checkpointing protocols in applications that require consistent global checkpoints, such as the detection of stable or unstable properties, rollback-recovery, and determination of distributed breakpoints.

References

[1] Acharya, A. and Badrinath, B. R., Checkpointing Distributed Application on Mobile Computers, *Proc. 3rd Int. Conf. on PDIS*, Austin, Sept. 1994, pp.73-80.

[2] Babaoğlu, Ö. Fromentin, E. and Raynal, M., A Unified Framework for the Specification and Run-time Detection of Dynamic Properties in Distributed Computations, *Journal of Systems and Software*, 33:287-298, 1996.

[3] Bhargava, B. and Lian, S.R., Independent Checkpointing and Concurrent Rollback for Recovery - An Optimistic Approach. *Proc. 7th IEEE SRDS*, 1988, pp. 3-12.

[4] Briatico, D., Ciufoletti, A. and Simoncini, L., A Distributed Domino-Effect Free Recovery Algorithm. *Proc. 4th IEEE Symp. on Reliability in Distributed Software and Database Systems*, Maryland, pp. 207-215, October 1984.

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

[6] Cooper, D. and Marzullo, K., Consistent Detection of Global Predicates. *Proc. ACM/ONR Workshop on Par. and Dist. Debugging*, Santa Cruz CA, pp. 167-174, 1991.

[7] Elnozahy, E.N., Johnson, D.B. and Wang, Y.M., A Survey of Rollback-Recovery Protocols in Message-Passing Systems, *Technical Report* CMU-CS-96-181, Carnegie-Mellon University, 1996.

[8] Fidge, C.J., Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):11-76, 1991.

[9] Fowler, J. and Zwaenepoel, W., Distributed Causal Breakpoints. *Proc 10th IEEE Int. Conf. on DCS*, pp. 134-141, May 1990.

[10] Garg, V.K., and Waldecker, B., Detection of Strong Unstable Predicates in Distributed Programs. *IEEE Trans. on PDS*, 7(12), December 1996, pp. 1323-1333.

[11] Helary, J.M., Jard, C., Plouzeau N. and Raynal M., Detection of Stable Properties in Distributed Applications. *Proc. 6th ACM Symp. on PODC*, 1987, pp. 125-136.

[12] Hélary, J.M., Mostefaoui, A., Netzer, R.H.B. and Raynal M., Communication-Based Prevention of Useless Checkpoints in Distributed Computations, *IRISA Research Report 1105*, 1997. www access: ftp://ftp.irisa.fr:/techreports/1997.

[13] Hurfin, M., Mizuno, M., Raynal, M. and Singhal, M., Efficient Distributed Detection of Conjunction of Local Predicates in Asynchronous Computations. *Proc. 8th Int. IEEE SPDP*, New-Orleans, October 1996, pp. 588-594.

[14] Kim, K. H., You, J. H. and Abouelnaga, A., A Scheme for Coordinated Execution of Independently Designed Recoverable Distributed Processes, *Proc. 16th IEEE Symp. on FTCS*, 1986, pp.130-135.

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

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

[17] Manivannan, D. and Singhal, M., A Low Overhead Recovery Technique Using Quasi-Synchronous Checkpointing. *Proc. of the 16th IEEE Int. Conf. on DCS*, pp. 100-107, Hong-Kong, May 1996.

[18] Miller, B., and Choi J., Breakpoint and Halting in Distributed Programs. *Proc. 8th IEEE Int. Conf. on DCS*, San Jose, pp. 316-323, May 1988.

[19] Netzer, R.H.B. and Xu, J., Necessary and Sufficient Conditions for Consistent Global Snapshots, *IEEE Trans. on PDS*, 6(2):165-169, 1995.

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

[21] Russell, D.L., State Restoration in Systems of Communicating Processes, *IEEE Trans. on SE*, SE6(2):183-194, 1980.

[22] Wang, Y.M. and Fuchs, W.F., Lazy Checkpoint Coordination for Bounding Rollback Propagation. *Proc. 12th IEEE SRDS*, pp. 78-85, Oct. 1993.

[23] Wang, Y.M., Consistent Global Checkpoints That Contain a Given Set of Local Checkpoints, *IEEE Trans. on Comp.*, 46(4):456-468, April 1997.

[24] Xu, J. and Netzer, R.H.B., Adaptive Independent Checkpointing for Reducing Rollback propagation. *Proc. 5th IEEE SPDP*, Dallas, pp. 754-761, Dec. 1993.