Direct Dependency-Based Determination of Consistent Global Checkpoints*

Roberto BALDONI, Giacomo CIOFFI
DIS
Università di Roma "La Sapienza"
Via Salaria 113, 00198, Roma, ITALY
E.mail: {baldoni,cioffi}@dis.uniroma1.it

Jean-Michel HELARY, Michel RAYNAL
IRISA
Campus de Beaulieu
35042 Rennes Cedex, FRANCE
E.mail: {helary,raynal}@irisa.fr

Abstract

Building consistent global checkpoints that contain a given set of local checkpoints has been usually handled by using transitive dependency tracking. This imply the usage of a vector of integers piggybacked on each message of the computation (the vector size being given by the number of processes). In this paper we address the problem to get consistent global checkpoints including a given subset of local checkpoints tracking just direct dependencies. In that case application messages are required to piggyback one integer. An algorithm is proposed that takes a set of local checkpoints as an input and returns the minimum consistent global checkpoint, if any, including that set. Otherwise it returns the first consistent global checkpoint that follows this subset. Among the applications of the algorithm there are rollback-recovery and global predicate detection.

Index Terms: Asynchronous Systems, Checkpoint Consistency, Communication-Induced Checkpointing, Direct Dependency, Predicate Detection, Transitive Dependency, Uncoordinated Checkpointing.

1 Introduction

A local checkpoint is a local state of a process and a global checkpoint is a set of local checkpoints, one from each process. A process defines some of its local states as local checkpoints for specific application or system-oriented reasons. This is the case, for example, in the conjunction of local

*This work has been partially supported by a grant in the context of the joint projects Consiglio Nazionale delle Ricerche/Centre National de la Recherche Scientifique (CNR/CNRS) - Project n. 5273/1998.
predicates detection problem and in the crash/recovery problem. In the first case, the local checkpoints of a process are its local states that satisfy a given local predicate (a predicate that is only on its local variables) [12]. In the latter case, a local checkpoint is a local state that a process has saved on stable storage for recovery purposes [7]. When a process defines a local state as a local checkpoint, it is usually said that the process takes a local checkpoint.

Among the whole set of global checkpoints, the set of consistent global checkpoints is of fundamental importance. This is because a consistent global checkpoint is a global state through which the application has passed or could have passed. At the operational level, this means that a consistent global checkpoint can not exhibit messages received and not sent. More precisely, let \( GC = (c_1, \cdots, c_n) \), be a global checkpoint (\( n \) being the number of processes and \( c_i \) being a local checkpoint of process \( P_i \)). \( GC \) is consistent if, for any ordered pair \((c_i, c_j)\), there is no message received by \( P_i \) before \( c_i \) and sent by \( P_j \) after \( c_j \).

Three families of checkpointing protocols have been investigated (1).

- The family of coordinated checkpointing protocols. In this family, processes coordinate their actions to take local checkpoints [14]. This coordination requires additional control messages, but ensures that piecing together the \( x \)-th local checkpoints of each process always provides a consistent global checkpoint. This kind of protocols have been mainly investigated to solve recovery problems [7]. The previous property allows an efficient recovery after a failure.

- The family of communication-induced checkpointing protocols. In that case, processes takes local checkpoints (namely, basic checkpoints) independently from each other, but they also cooperate to take additional local checkpoints (namely, forced checkpoints) [3, 4, 5, 10, 16, 22]. This cooperation ensures that any local checkpoint will belong to at least one consistent global checkpoint. This is done by piggybacking control information on application messages. No additional message is required. When a process receives a message, it uses its local control information and the information carried by the message to decide if it has to take a forced checkpoint.

According to the control information managed by processes and carried by application messages, several communication-induced checkpointing protocols can be designed. Some of them are able to associate with each local checkpoint \( c \) the identity of a consistent global checkpoint to which \( c \) belongs [3, 5, 10, 22].

\(^1\)A nice survey on checkpointing-related problems in the context of rollback-recovery can be found in [7].
The family of uncoordinated checkpointing protocols. In that family, each process takes its local checkpoints independently from the other processes. Consequently, it is possible that some local checkpoints can not be a member of a consistent global checkpoint; this can lead to the well-known domino effect [20]. Moreover, when a process takes a local checkpoint, it is not possible to on-the-fly associate it with one of its consistent global checkpoints (if any).

A distributed computation on which a set of local states have been defined as local checkpoints is usually called a communication and checkpoint pattern (CCP). It is important to note that a CCP constitutes an abstraction of the computation: only the local states that are local checkpoints are visible, and messages create a dependency relation among them. This abstraction is sometimes represented by a checkpoint graph [22].

In this paper we are interested in local checkpoints defined by checkpointing protocols that do not associate on-the-fly a consistent global checkpoint with each local checkpoint. This concerns all the uncoordinated checkpointing protocols and a subset of the communication-induced checkpointing protocols (e.g., [4]). So, we assume that dependency information of all the local checkpoints they define are sent to a checker process. Then, given an arbitrary set $M_x$ containing $x$ local checkpoints from distinct processes, the problem we want to solve is:

To find an algorithm for the checker process which computes a consistent global checkpoint that includes $M_x$, or if there is no consistent global checkpoint including $M_x$, which computes the first consistent global checkpoint that is after $M_x$.

To attain this goal, control information has to be associated with local checkpoints when they are taken. This information is on the dependencies between local checkpoints. It is then used by the checker process to determine the relevant consistent global checkpoint that can be associated with $M_x$. It has been shown that the tracking of transitive dependencies (vector timestamps [18]) between local checkpoints provides control information that is sufficient for the checker process to solve the problem [9]. More precisely, during the computation, processes manage vector timestamps and messages carry them. This allows to associate a vector timestamp with each local checkpoint, these timestamps can then be used by the the checker process to compute a consistent global checkpoint. This approach solves the problem but, as noted before, requires that during the computation, application messages carry vector timestamps.

It is important to remark that tracking transitive dependencies means detecting the happen before relation [15] on local checkpoints. But, it has been shown that "direct dependency tracking suffices to enable the determination of consistent global checkpoints ... the missing dependencies
(with respect to the "happen before" relation) are redundant with respect to consistent global checkpoints and \( \ldots \) direct dependency tracking automatically filters out this redundant information" [23]. The authors proved their claim by exhibiting a formal transformation that can be applied to any communication and checkpoint pattern.

This paper investigates this idea and complements the previous result by providing an operational answer (i.e., an algorithm) to the problem. It requires that, during a computation, processes track only direct dependencies between local checkpoints. More precisely, during the computation each process manages a direct dependency vector and each message has to carry a single integer. When a process takes a local checkpoint \( c \), it associates with \( c \) a timestamp whose value comes from its direct dependency vector.

In both cases (transitive dependency tracking and direct dependency tracking) the checker process receives all local checkpoints with their vector timestamps (but the meaning of vectors is particular to each case). From them the checker can build the abstraction defined by the corresponding CCP, and solve the problem. But it is important to note that, during the computation, the tracking of direct dependency is less expensive than the tracking of transitive dependency: application messages do not carry an integer vector but a single integer. Moreover, integer values in direct dependency vectors never increase faster than the ones used by transitive dependency vectors. Let us finally note that finding the minimal information that has to be tracked to solve the problem is an open question.

The paper is structured as follows. Section 2 describes the model of the computation. Section 3 introduces the problem and the structure of global checkpoints of a distributed computation. Section 4 presents the algorithm and gives its proof of correctness. Section 5 discusses related work. Finally Section 6 concludes the paper.

2 Model of the Computation

2.1 Distributed Computation

A distributed computation consists of a finite set of \( n \) processes \( \{P_1, P_2, \ldots, P_n\} \) that communicate and synchronize only by exchanging messages. They do not share a memory. Each ordered pair of processes is connected by a reliable directed logical channel. The system is asynchronous. More precisely, transmission delays are finite but unpredictable and there is no bound for the relative speed of processes.

A process can execute internal, send and receive statements. An internal statement is any
statement that does not involve communication. A send statement allows a process to send a message. When $P_i$ executes $send(m)$ to $P_j$, it actually puts the message $m$ into the channel from $P_i$ to $P_j$. A receive statement allows a process to receive a message. When $P_i$ executes $receive(m)$, it is blocked until at least one message (directed to $P_i$) has arrived; then a message is withdrawn from one of its input channel and delivered to $P_i$. Executions of send and receive statements are modeled by send and receive events.

Processes of a distributed computation are sequential, in other words, each process produces a sequence of events; $e_{i,t}$ denotes the $t$-th event executed by $P_i$ ($e_{i,0}$ is a fictitious event that initializes $P_i$'s local state, namely $\sigma_{i,0}$).

As an example, Figure 1 depicts a distributed computation in the usual space-time diagram: events are represented by black circles and messages by arrow.

### 2.2 Local Checkpoints

The event $e_{i,t}$ moves the local context of $P_i$ from the local state $\sigma_{i,t-1}$ to the local state $\sigma_{i,t}$. By definition we say that "$e_{i,t}$ belongs to $\sigma_{j,t'}$" (denoted $e_{i,t} \in \sigma_{j,t'}$) iff $i = j$ and $t' \geq t$.

As indicated in the Introduction, a local state of a process $P_i$ can be defined as being a local checkpoint. Note that the set of all local checkpoints is a subset of all local states (i.e., a local state is not necessarily a local checkpoint). Let $c_{i,t}$ denotes the $t$-th local checkpoint taken by $P_i$ (the integer $t$ is called the sequence number associated with $c_{i,t}$). This local checkpoint corresponds to some local state $t'$ with $t' \geq t$. We assume that each process (1) takes an initial checkpoint $c_{i,0}$ (corresponding to $\sigma_{i,0}$) and (2) eventually takes a final checkpoint $c_{i,\text{last}}$ (so, we assume that the computation terminates). In Figure 1, local checkpoints are represented by rectangles, while local
2.3 Direct Dependency between Local Checkpoints

When we consider a pair of processes, each message they exchange establishes dependencies between some of their local checkpoints. The associated dependency relation is called direct dependency. This section defines this relation and shows how it can be tracked during an execution.

Direct Dependency: Definition.

Definition 2.1 Let \((c_{i,t}, c_{j,t'})\) be a pair of local checkpoints. \(c_{i,t}\) depends on \(c_{j,t'}\) (denoted \(c_{i,t} \leadsto c_{j,t'}\)) if:

1) \(i = j\) and \(t' < t\), or

2) there exists a message \(m\) sent by \(P_j\) after \(c_{j,t'}\) (i.e., \(send(m) \notin c_{j,t'}\)) that is received by \(P_i\) before \(c_{i,t}\) (i.e., \(receive(m) \in c_{i,t}\)). Such a message \(m\) is called orphan with respect to the ordered pair of local checkpoints \((c_{i,t}, c_{j,t'})\).

As an example, let us examine the computation shown in Figure 1. We have: \(c_{2,0} \leadsto c_{1,1}\) (\(m_1\) is orphan) and \(c_{1,1} \leadsto c_{3,1}\) (\(m_3\) is orphan) while \(!c_{1,1} \leadsto c_{2,2}\) and \(!c_{2,2} \leadsto c_{1,1}\).

It is important to note that the relation "\(\leadsto\)" is not transitive. Let us consider \(c_{3,0}\) and \(c_{1,3}\). As there is no message sent by \(P_3\) to \(P_1\), there is no relation "\(\leadsto\)" from any checkpoint of \(P_3\) to any checkpoint of \(P_1\). So, although we have \(c_{3,0} \leadsto c_{2,1}\) and \(c_{2,1} \leadsto c_{1,3}\), we do not have \(c_{3,0} \leadsto c_{1,3}\).

Direct Dependency: Tracking.

The relation "\(\leadsto\)" can be tracked during the computation in the following way. Each process \(P_i\) is equipped with a sequence number \(next_sn_i\) (initialized to 1) and with a dependency vector \(DV_i[1..n]\) (initialized to 0) which contains sequence numbers. These local variables are managed according to the following rules:

1. **(R1) "Taking a Local Checkpoint" Rule.** When it defines a local checkpoint (e.g., \(c_{i,t}\)), \(P_i\) first atomically executes \([DV_i[i] \leftarrow next_sn_i; next_sn_i \leftarrow next_sn_i + 1]\) \(^2\). It then defines the timestamp of \(c_{i,t}\) as being the current value of \(DV_i\). This timestamp will be denoted \(DV_{i,t}\).

2. **(R2) "Message Sending" Rule.** Each time it sends a message \(m\), \(P_i\) appends to \(m\) the current value of \(next_sn_i\). Let \(m.next_sn\) denote this value.

\(^2\)Note that the relation \(DV_i[i] + 1 = next_sn_i\) is invariant.
Figure 2: Distributed Computation with Dependency Vectors

- (R3) “Message Reception” Rule. When $P_i$ receives a message $m$ from $P_j$, it updates the corresponding entry of its dependency vector: $DV_i[j] \leftarrow max(DV_i[j], m.new \_s[n])$.

Figure 2 shows the timestamp values of the local checkpoints of the running example. It is important to note that messages carry a single integer (sequence number) as additional control information.

The component $DV_i[i]$ corresponds to the sequence number of the last checkpoint taken by $P_i$. So, given any local checkpoint $c_{i,t}$ with its associated timestamp $DV_{i,t}$, we have $DV_{i,t}[i] = t$. For $j \neq i$, $DV_i[j]$ represents the best knowledge $P_i$ has received from $P_j$ concerning the next sequence number that $P_j$ will use. The previous tracking enjoys the following important property (which is a direct consequence of Definition 2.1 expressed with dependency vectors):

\[(P) \equiv (\forall (i, j, t) (\forall t' : (0 \leq t' < DV_{i,t}[j]) \Rightarrow c_{j,t'} \sim c_{i,t}))\]

Note that when we consider transitive dependencies (whose tracking requires each application message to piggyback an integer vector), due to the causal sequence of messages ($m_2, m_4$), the local checkpoint $c_{3,0}$ causally precedes (happens before) the local checkpoint $c_{1,3}$. It is because the relation “$\sim$” does not consider dependencies on local checkpoints that are created by causal sequence of messages, this is the reason why this relation is called direct dependency.

3 The Problem

3.1 Consistent Global Checkpoints

A global checkpoint $GC$ is a set of local checkpoints, one from each process. The set of local checkpoints $(c_{1,0}, c_{2,2}, c_{3,1})$ (Figure 2) is an example of global checkpoint. A fundamental issue is
the notion of consistent global checkpoint [2, 6, 11].

**Definition 3.1** A global checkpoint \( GC = (c_{1,t_1}, \ldots, c_{n,t_n}) \) is consistent if no message is orphan with respect to any of its ordered pairs \((c_{i,t_i}, c_{k,t_k})\) of local checkpoints with \( i \neq k \).

Let us again consider Figure 2. The global checkpoint \((c_{1,2}, c_{2,3}, c_{3,2})\) is not consistent (the message \( m_3 \) is orphan wrt \((c_{1,2}, c_{2,3})\)). The global checkpoint \((c_{1,2}, c_{2,2}, c_{3,1})\) is consistent (\( m_4 \) is not orphan). We can also remark that the local checkpoint \( c_{1,1} \) can not be member of a consistent global checkpoint \(^3\). When we consider the global checkpoint defined by the sequence numbers of a dependency vector, we get a global checkpoint. This checkpoint is not necessarily consistent. For example, the dependency vector \([0,0,0]\) that timestamps \( c_{1,0} \) defines the consistent global state \((c_{1,0}, c_{2,0}, c_{3,0})\). The dependency vector \((0,2,1)\) that timestamps \( c_{2,2} \) defines the global state \((c_{1,0}, c_{2,2}, c_{3,1})\) that is not consistent (\( m_3 \) is orphan).

### 3.2 The Structure of the Set of Global Checkpoint

Let \(GC\) and \(CGC\) be the set of all global checkpoints and the set of all consistent global checkpoints of a given computation, respectively.

**Definition 3.2** A set \( M \) of local checkpoints precedes a global checkpoint \( GC \in GC\), (denoted \( M \preceq GC \)) if \( \forall c_{i,t} \in M, \exists c_{i,t'} \in GC : t \leq t' \).

For example, the set \((c_{1,1}, c_{2,1}) \preceq (c_{1,2}, c_{2,2}, c_{3,1})\). An interesting point is to define the first consistent global checkpoint that includes or follows a given set of local checkpoints \( M \).

**Definition 3.3** Let \( M \) be a set of local checkpoints and \( CGC \) be a consistent global checkpoint. \( CGC \) is the first consistent global checkpoint that includes or follows \( M \) (denoted \( First(M) \)) if:

1) \( M \preceq CGC \).

2) \( \nexists CGC' \in CGC : M \preceq CGC' \preceq CGC \) with \( CGC' \neq CGC \).

It has been shown that \( GC \) constitutes a lattice and \( CGC \) a sublattice of \( GC \) [8, 11, 13, 17, 22]. Thus, for any \( M \), \( First(M) \) always exists and is unique. Let \( M1 = \{c_{2,2}, c_{3,1}\} \). Then \( First(M1) = (c_{1,2}, c_{2,2}, c_{3,1}) \). In that case \( First(M1) \) includes \( M1 \) \(^4\). Let now consider \( M2 = \{c_{1,1}\} \). We have \( First(M2) = (c_{1,2}, c_{2,1}, c_{3,1}) \). In that case \( First(M2) \) follows \( M2 \).

---

\(^3\)When considering the Z-path theory [19], \( c_{1,1} \) is included in a Z-cycle. This states the impossibility to include \( c_{1,1} \) in a consistent global checkpoint.

\(^4\)When considering the Z-path theory [19], \( First(M1) \) is the leading edge of the Z-cone associated with \( M1 \) [17].
3.3 Problem Statement

Given a set $M_x$ of $x$ ($1 \leq x \leq n$) local checkpoints from distinct processes, the goal is to find an algorithm that takes $M_x$ as an input and returns $First(M_x)$ \(^5\). To compute its result, the algorithm can use the direct dependency vector-based timestamps of the local checkpoints.

4 The Algorithm

4.1 The Checker Process

Given $M$, the determination of $First(M)$ can be done by a checker process in the following way. Each time a process takes a local checkpoint, it sends the checker process the corresponding timestamp. The checker has $n$ queues, one for each process, where it stores the timestamps (direct dependency vectors) received from each process. If the checker requires a timestamp which has not yet been deposited in the corresponding queue, it stops until it receives a message carrying the required information. The algorithm executed by the checker process is described in Figure 3. The pair $(i_k, t_k)$ identifies the local checkpoint $c_{i_k, t_k}$, whose timestamp vector is $DV_{i_k, t_k}$. The set $M$, given to the algorithm as an input, is a set of local checkpoint identities $((i_1, t_1), \ldots, (i_x, t_x))$ with $(x \geq 1)$ which corresponds to the set $(c_{i_1, t_1}, \ldots, c_{i_x, t_x})$. The function $\text{max}(V_1, V_2, \ldots)$ is defined in the following way:

$$\forall j \in \{1, \ldots, n\}: \text{max}(V_1, V_2, \ldots)[j] = \text{max}(V_1[j], V_2[j], \ldots)$$

```
procedure consistent_global_checkpoint(M : set of local checkpoint identities)
% M \equiv ((i_1, t_1), \ldots, (i_x, t_x)) with (x \geq 1) %
begin
(1) GC \leftarrow \text{max}(DV_{i_1, t_1}, \ldots, DV_{i_x, t_x});
(2) repeat
(2.1) GC' \leftarrow GC;
(2.2) for each k \in \{1, \ldots, n\} do GC \leftarrow \text{max}(GC, DV_{i_k, GC[k]}) enddo
(2.3) until (GC = GC');
(3) return(GC) % GC = First(M) %
end
```

Figure 3: Computing $First(M)$

Line (1) computes the global checkpoint which includes all the dependencies of the local checkpoints belonging to $M$. As an example, if we consider $M$ as the set of local checkpoint identities
\(^5\)From a formal point of view, the problem consists in computing a fixed point associated with $M_x$. 

9
corresponding to the set \((c_{1,1}, c_{2,1})\) (Figure 2), line (1) returns the dependency vector \([1, 1, 1]\) which corresponds to the global checkpoint \((c_{1,1}, c_{2,1}, c_{3,1})\). This global checkpoint is not consistent as \(c_{1,1} \sim c_{3,1}\), so other dependencies have to be incorporated in \(GC\) in order to get a consistent global checkpoint.

Then, starting from \(GC\), Step (2) computes a closure of the relation \(\sim\). Starting from direct dependencies defined by \(GC'\), the algorithm transitively captures all the dependencies between local checkpoints. The inner loop moves forward the global checkpoint \(GC\) by incorporating all the dependencies revealed by local checkpoints belonging to \(GC'\). This is done by computing the maximum among the all timestamps currently identified in \(GC\) (step (2.2)). The outer loop checks when all the dependencies have been incorporated in \(GC\). In this case the computation terminates and the result \(GC\) is the first consistent global checkpoint including or following \(M\), namely, \(First(M)\).

This algorithm can be distributed by a simple duplication technique. A watch-dog process is associated with each application process and each time a local checkpoint is taken by an application process its timestamp is sent to all the watch-dog processes. The algorithm of Figure 3 is then run locally by each watch-dog process.

**Example (1)** Let \(M = (c_{2,2})\). As \(M\) contains only one local checkpoint, after step (1) \(GC\) corresponds to \(DV_{2,2} \equiv [0, 2, 1]\), after step (2.2) \(GC\) becomes \([2, 2, 1]\). Then a new loop is started as \([0, 2, 1] \neq [2, 2, 1]\). After step (2.2) \(GC\) does not change, and consequently the vector \([2, 2, 1]\) is returned to the calling function. In this case, \(First(M) = [2, 2, 1]\), which includes \(M\).

**Example (2)** Let \(M = (c_{1,1}, c_{2,1})\). After step (1) \(GC = [1, 1, 1]\), and after step (2.2) \(GC\) becomes \([2, 1, 1]\). Then a new loop is started as \([1, 1, 1] \neq [2, 1, 1]\). After step (2.2) \(GC\) remains \([2, 1, 1]\), and \([2, 1, 1]\) is returned to the calling function. \(First(M) = [2, 1, 1]\) corresponds to the consistent global checkpoint \((c_{1,2}, c_{2,2}, c_{3,1})\). Here, \(First(M)\) does not include \(M\).

### 4.2 Proof of Correctness

The proof is composed of three parts. The first is related to the termination (Theorem 4.1). The other two are safety properties. One states that the global checkpoint computed by the algorithm is consistent (Theorem 4.2), while the other states that the result is actually \(First(M)\) (Theorem 4.3).

**Theorem 4.1 (Termination)** The algorithm terminates.
Proof This follows directly from the fact that sequence numbers in GC never decrease. As, by assumption, each process \( P_j \) takes a last local checkpoint \( c_{j, \text{last}_j} \) (after which \( P_j \) executes no more event), \( GC[j] \) is bounded by \( \text{last}_j \). As for any pair of processes \( P_j \) and \( P_k \), we have \( DV_{k, \text{last}_n}[j] \leq \text{last}_j \), we get, in the worst case, \( GC = [\text{last}_1, \ldots, \text{last}_n] \) that can not be changed. So, the algorithm exits from the outer loop and terminates.

\[ \square \]

**Theorem 4.2 (Consistency)** Let consider the vector GC output by the algorithm. The global checkpoint \( (c_1, GC[1], \ldots, c_n, GC[n]) \) is consistent.

Proof As we consider the value of GC at line (3), this means that the algorithm exited from the repeat loop and then \( GC = GC' \).

Then, according to the operations executed by the algorithm in the loop (line (2)-(2.3)), we have

\[ \forall (i,k) \in \{1, \ldots n\}^2 : GC[i] \geq DV_{k, GC[k]}[i] \]

Due to property (P) of Section 2.3, we have

\[ \forall (i,k) \in \{1, \ldots n\}^2 : GC[i] \geq DV_{k, GC[k]}[i] \Rightarrow \neg (c_{i, GC[i]} \sim c_{k, GC[k]}) \]

Then, from part (2) of Definition 2.1, we conclude that there is no orphan message between any ordered pair of checkpoints \((c_{i, GC[i]}, c_{k, GC[k]}) \in (c_1, GC[1], c_2, GC[2], \ldots, c_n, GC[n])\). As a consequence, from Definition 3.1, the set \((c_1, GC[1], c_2, GC[2], \ldots, c_n, GC[n])\) is consistent.

\[ \square \]

**Theorem 4.3 (Minimality)** The output of the algorithm is First(\( M \)).

Proof Let \( GC_{init} \) be the starting point of the algorithm and let \( CGC_2 \) be its output (Figure 4). Let us assume, by the way of contradiction, that \( \exists CGC_1 (\neq CGC_2) \) that defines a consistent global checkpoint such that \( GC_{init} \preceq CGC_1 \preceq CGC_2 \). We show that \( CGC_1 \) (which has not been output by the algorithm) does not define a consistent global checkpoint (so, \( CGC_2 \) is First(\( M \))).

Let us consider the first time the algorithm crossed from the left to the right of the global checkpoint defined by \( CGC_1 \). This crossing has necessarily occurred, because the initial vector defines a global checkpoint on the left of the one defined by \( CGC_1 \) (\( GC_{init} \preceq CGC_1 \)), and the result vector defines a global checkpoint that is on the right of the one defined by \( CGC_1 \) (\( CGC_1 \preceq CGC_2 \)).

Just before crossing \( CGC_1 \), the current global checkpoint considered by the algorithm is the one defined by \( GC \) (Figure 4). The crossing is due to the existence of a local checkpoint \( c_{j,t} \) of a process \( P_j \), such that, at line 2.2 an update of \( GC \) has been done and this update is such that \( \exists i \) such that
$DV_{j,t}[i] > CGC_1[i]$. This means that there is a message $m$ that has been sent by $P_i$ after $c_i, CGC_1[i]$ and that has been received by $P_j$ before $c_j, CGC_1[j]$. But, this message has been sent after the global checkpoint defined by $CGC_1$ and has been received before it: it is clearly orphan with respect to a pair of local checkpoints of $CGC_1$. So, $CGC_1$ does not define a consistent global checkpoint. \[\square\]

5 Related Work

This section discusses three algorithms, namely, Baldy et al.'s algorithm (BA) [1]\(^6\), Johnson and Zwaenepoel's algorithm (JZ) [13] and Wang's algorithm (WA) [22]. These algorithms compute consistent global checkpoints of a computation by means of a checker process. This checker uses vector timestamps associated with local checkpoints. These vector timestamps encode direct dependencies between pairs of local checkpoints.

BA is based on a model that does not abstract the computation: every local state is a local checkpoint, and consequently, every local checkpoint does belong to at least one consistent global checkpoint [21]. BA receives a single local checkpoint as an input, and returns the first consistent global checkpoint that includes that local checkpoint.

JZ works on a CCP abstraction based on checkpoints. It additionally assumes that each process follows a PieceWise Deterministic (PWD) behavior. Under this hypothesis each process execution is divided into deterministic intervals separated by non-deterministic events. Each time a non-deterministic event (such as a receive event) is executed, a local checkpoint is taken. It has been shown in [22] that if processes follows a PWD behavior, the resulting CCP has several noteworthy

\(^6\) A description of this algorithm can be also found in [21].
properties, including this one: each local checkpoint belongs to at least one consistent global checkpoint (and this global checkpoint can be directly determined without using control messages and without the help of an external checker process).

JZ allows to determine the consistent global checkpoint closest to the end of the computation (called maximum recoverable state) that includes a given local checkpoint. Each time a local checkpoint \( c \) is taken, a checker examines the \((n, n)\) matrix formed by piecing together the direct dependency vector of \( c \) and the ones of the other processes. The checker loops until each element of the diagonal of the matrix is greater than or equal to any element in that line. Due to the PWD property, JZ always succeeds and the diagonal of the matrix corresponds to the maximum recoverable state including \( c \).

From a practical point of view, the number of checkpoints will be greater than or (in the best case) equal to the number of receive events which is expected to be quite high in distributed computations.

As a final remark let us note that, neither JZ nor BA can be applied to checkpoint and communication patterns created by uncoordinated checkpointing protocols; this is because such patterns can contain local checkpoints that can never belong to a consistent global checkpoint (as \( c_{1,1} \) in Figure 2). Moreover, JZ and BA accept only a single local checkpoint as an input. But, as shown by Wang in [22], many checkpoint-related problems (for example deadlock detection and resolution, output commit, software-retry, just to name a few) need to form consistent global checkpoints that include a specific subset of local checkpoints.

WA abstracts the computation as a checkpoint graph, called a R-graph. Its nodes are the set of local checkpoints and its directed edges correspond to direct dependencies created by messages. The main difference with our approach is that WA explicitly builds the R-graph of a distributed computation. The checker process then performs a reachability analysis on the R-graph to compute a global consistent checkpoint including a given set of local checkpoints. Our algorithm actually does not require the construction of the R-graph.

6 Conclusion

The paper has presented an algorithm that, given a set \( M \) of local checkpoints from distinct processes as an input, returns the first consistent global checkpoint that includes or follows \( M \). The algorithm is executed by a checker process working with direct dependency vectors provided by application processes at the time they take a local checkpoint. This algorithm is well-suited to define consistent global checkpoints when local checkpoints are defined by uncoordinated checkpointing
protocols or by some communication-induced protocols.

Compared to algorithms that are based on the explicit tracking of transitive dependencies among local checkpoints to determine consistent global checkpoints, the one presented in this paper is less expensive as it only requires the tracking of direct dependencies. For such a tracking, each application message has to piggyback an integer rather than a vectors of integers. The area of usage of such an algorithm is quite large and includes consistent checkpoints-related problems such as rollback-recovery and global predicate detection.

References


