IEICE TRANS. INF. & SYST., VOL.E96–D, NO.4 APRIL 2013

PAPER

A Scalable Communication-Induced Checkpointing Algorithm for Distributed Systems

Alberto CALIXTO SIMON[†], ***Student Member***, **Saul E. POMARES HERNANDEZ**[††a)], ***Member***,
Jose Roberto PEREZ CRUZ[†], **Pilar GOMEZ-GIL**[†], ***and*** **Khalil DRIRA**[††], ***Nonmembers***

SUMMARY *Communication-induced checkpointing* (CIC) has two main advantages: first, it allows processes in a distributed computation to take asynchronous checkpoints, and secondly, it avoids the domino effect. To achieve these, CIC algorithms piggyback information on the application messages and take forced local checkpoints when they recognize potentially dangerous patterns. The main disadvantages of CIC algorithms are the amount of overhead per message and the induced storage overhead. In this paper we present a communication-induced checkpointing algorithm called *Scalable Fully-Informed* (S-FI) that attacks the problem of message overhead. For this, our algorithm modifies the *Fully-Informed algorithm* by integrating it with the *immediate dependency principle*. The S-FI algorithm was simulated and the result shows that the algorithm is scalable since the message overhead presents an under-linear growth as the number of processes and/or the message density increase.

key words: *distributed systems, communication-induced checkpointing, immediate dependency relation*

1. Introduction

Communication-induced checkpointing (CIC) algorithms are useful for a wide range of problems that arise in distributed systems, such as: rollback recovery and software debugging. In CIC algorithms a process asynchronously cooperates by exchanging information about distinguished states of its execution called *local checkpoints*. CIC algorithms are oriented to form *global consistent snapshots* (GCS) by grouping local checkpoints (one by each process) in a non-coordinated way.

CIC algorithms have several advantages over other styles of checkpointing, namely *coordinated checkpointing* (CC) and *uncoordinated checkpointing* (UCC) [1]. The CC algorithms need to exchange extra control messages to coordinate a GCS while it is possible that some process remains blocked along the construction of the GCS. The UCC algorithms can asynchronously take local checkpoints at any time during the execution; nevertheless, they are susceptible to the *domino effect* [2]. CIC algorithms avoid the domino effect and allow an asynchronous execution. To achieve this, CIC algorithms piggyback information on the application messages to identify potentially dangerous checkpointing patterns. A dangerous pattern is broken before it occurs by locally triggering a *forced checkpoint*. The dangerous patterns are the Z-cycles identified by Netzer [3].

The main disadvantages of CIC algorithms are the amount of overhead per message and the induced storage overhead [4]. In the present paper we introduce a CIC algorithm called *Scalable Fully-Informed* (S-FI) that attacks the problem of message overhead. For this, our algorithm modifies the *Fully-Informed* (FI) algorithm of Helary et al. [5] by integrating the *immediate dependency relation* (IDR). The FI algorithm was chosen because it is one of the most important approaches, since it establishes relevant fundamentals for the CIC algorithms [6]. The IDR was used because it identifies the necessary and sufficient causal dependency constraints among events in a distributed system [7]. In summary, the aim of the S-FI is to take the same number of forced checkpoints as the work of Hélary et al. [5] but significantly reducing the overhead sent per message.

The S-FI algorithm was simulated, and the results show that the algorithm is scalable since the overhead per message presents an under-linear growth as the number of processes and the message density increase, which is defined as the number of messages sent per process in a period of time.

This paper proceeds as follows. In Sect. 2, we present the system model and background. In Sect. 3, the S-FI algorithm is presented. Next, in Sect. 4, we give the simulation results. Finally, in Sect. 5, some conclusions are presented.

2. Preliminaries

2.1 System Model

The system under consideration is composed of a finite set of *processes* $P = \{p_1, p_2, \ldots, p_n\}$. The processes present an asynchronous execution and communicate only by message passing. Moreover, processes fail according to the *fail-stop* model [1]. Let e_i^x be the *x-th event* produced by process p_i. The sequence $h_i = e_i^0 e_i^1 \ldots e_i^x \ldots$ constitutes the history of p_i, denoted by H_i. We consider two types of events: *internal* and *external* events. An internal event is a unique action that occurs at a process p and changes only its local state. The finite set of internal events is denoted by R. In this paper, we consider only the checkpoints as internal events, and we use C_i^x to denote the x^{th} checkpoint of process p_i. For the checkpointing problem, the set R represents the set of *relevant events*[†] to be considered. We assume that each pro-

Manuscript received September 27, 2012.
Manuscript revised November 20, 2012.

[†]The authors are with the Computer Science Department, Instituto Nacional de Astrofísica, Óptica y Electrónica in Tonantzintla, Puebla, Mexico.
[††]The authors are with LAAS-CNRS and Univ. de Toulouse, France.
a) E-mail: spomares@inaoep.mx

DOI: 10.1587/transinf.E96.D.886

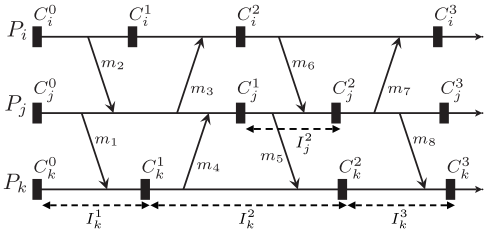

Fig. 1 A communication and checkpoint pattern.

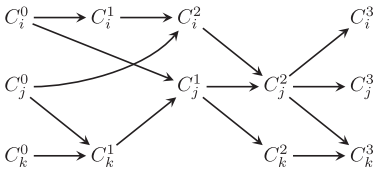

Fig. 2 IDR Graph of Fig. 1.

cess takes a checkpoint after execution begins (*initial checkpoint*) and before an execution ends (*final checkpoint*). On the other hand, an external event is also a unique action that occurs at a process, but it is seen by other processes and affects the global state of the system. The external events considered in this paper are the *send* and *delivery* events. We consider a finite set M of messages, where each message $m \in M$ is sent through an asynchronous reliable network that is characterized by transmissions with no time boundaries, no ordered delivery, and no lost messages. Let m be a message; we denote by $send(m)$ the emission of m and by $delivery(p, m)$ the delivery event of m to participant $p \in P$. The set of events associated to M is the set $E_m = \{send(m) : m \in M\} \cup \{delivery(p, m) : m \in M \wedge p \in P\}$. The whole set of events in the system is the finite set $E = R \cup E_m$. The distributed computation is modeled by the partially ordered set $\widehat{E} = (E, \rightarrow)$, where $\rightarrow$ denotes Lamport's well-known *happened-before* relation [8] (see Definition 2).

2.2 Background and Definitions

Definition 1. *A communication and checkpoint pattern (CCP) is a pair $(\widehat{E}, R_{\widehat{E}})$ where $\widehat{E}$ is a partially ordered set modeling a distributed computation, and $R_{\widehat{E}}$ is a set of local checkpoints defined on $\widehat{E}$ [5].*

Figure 1 shows an example of a communication and checkpoint pattern. The sequence of events ocurring at p_i between C_i^{x-1} and C_i^x $(x > 0)$ is called a *checkpoint interval*, denoted by I_i^x.

Definition 2. *The happened-before relation (HBR) [8], "$\rightarrow$", is the smallest relation on a set of events E satisfying the following properties:*

1. *If a and b are events of the same process, and a was originated before b, then $a \rightarrow b$.*
2. *If a is the event send(m) and b is the event delivery(m), then $a \rightarrow b$.*
3. *If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.*

Immediate Dependency Relation (IDR). The IDR is the *transitive reduction* of the HBR [7]. We denote the IDR by "$\downarrow$", and its formal definition is as follows:

Definition 3. *Two events $a, b \in E$ have an immediate dependency relation "$a \downarrow b$" if the following restriction is satisfied.*

$$a \downarrow b \;\; if \;\; a \rightarrow b \; and \; \forall c \in E, \neg(a \rightarrow c \rightarrow b)$$

In our context, we are only interested in identifying the immediate dependency relations among the set of relevant events $R \subset E$, which contain the checkpoint events. Therefore, we say that a pair of checkpoint (relevant) events $x, y \in R$ is IDR related if and only if no other relevant event $z \in R$ exists, such that z belongs to the causal future of x and to the causal past of y. The IDR graph of the scenario in Fig. 1 is shown in Fig. 2.

Next, we present the principles of communication-induced checkpointing.

Netzer and Xu [3] defined the notion of *zigzag path* (z-path) as a generalization of HBR, as follows:

Definition 4. *A z-path exists from C_p^i to another C_q^j iff there are messages $m_1, m_2, \ldots, m_\ell$ such that:*

1. *m_1 is sent by process p after C_p^i,*
2. *if m_k $(1 \leq k < \ell)$ is received by process r, then m_{k+1} is sent by r in the same or at a later checkpoint interval (although m_{k+1} may be sent before or after m_k is received), and*
3. *m_ℓ is received by process q before C_q^j.*

Hélary et al. defined the following in [5].

Definition 5. *A z-path $[m_1, \ldots, m_q]$ is causal, iff for each pair of consecutive messages m_α and $m_{\alpha+1}$: $delivery(m_\alpha) \rightarrow send(m_{\alpha+1})$. Otherwise, it is a non causal z-path.*

Definition 6. *A local checkpoint C_j^y Z-depends on a local checkpoint C_i^x, $C_i^x \xrightarrow{Z} C_j^y$, if:*

1. *$j = i$ and $y > x$, or*
2. *there is a z-path from C_i^x to C_j^y.*

Definition 7. *A* z-cycle *is a* Z-dependency *from a local checkpoint C_i^x to itself: $C_i^x \xrightarrow{Z} C_i^x$.*

In Fig. 1, the messages $[m_4, m_1]$ form a z-cycle involving C_k^1, and $[m_6, m_5, m_4, m_3]$ form a z-cycle in C_i^2.

Theorem 1. The following properties of a communication and checkpoint pattern $(\widehat{E}, R_{\widehat{E}})$ are equivalent:

1. $(\widehat{E}, R_{\widehat{E}})$ has no z-cycle.
2. It is possible to timestamp its local checkpoints in such a manner that $A \xrightarrow{Z} B \Rightarrow A.t < B.t$.

where t is a logical clock as defined by Lamport [8].

[†]A set R of relevant events is a subset of events of the distributed computation, such that R constitutes a major abstraction level of it.

3. S-FI Algorithm

The S-FI algorithm is based on the principles introduced in the FI checkpointing protocol proposed by Hélary et al. [5] and the IPT2 tracking protocol [9]. Specifically, S-FI uses Theorem 1 and the forced checkpoint condition $\mathscr{C}2''$ of FI to prevent z-cycles, and it uses the tracking approach of IPT2 that is based on the IDR to reduce the communication overhead.

To fuse such principles in S-FI, it was first necessary to define an initial forced checkpoint condition named $\mathscr{D}$. This condition is expressed, as well as $\mathscr{C}2''$, with static structures, but in terms of IDR related checkpoints. This means that the size of the structures used in both conditions is constant. We show that $\mathscr{D}$ is equivalent to $\mathscr{C}2''$ to ensure Theorem 1. Then $\mathscr{D}$ is redefined by using dynamic structures and it is called $\mathscr{D}'$. In this case, the size of the data structures to be analyzed is dynamically adapted according to the IDR checkpoint behavior of the system. Based on this last condition, the S-FI algorithm presented in Table 1 is designed.

Since the condition $\mathscr{C}2''$ of FI is fundamental for our work, we begin by giving a detailed description about its main components.

3.1 The FI Forced Checkpoint Condition

The forced checkpoint condition $\mathscr{C}2''$, as shown in [5], ensures Theorem 1. If in the reception of a message at a process p_i the condition $\mathscr{C}2''$ is true, then such process is forced to take a local checkpoint. This action breaks a z-path that contains a checkpoint which eventually can belong to a z-cycle. This condition is defined as follows.

$$\mathscr{C}2'' \equiv ((\exists k : sent\_to_i[k] \wedge m.greater[k]) \wedge m.lc > lc_i) \vee (ckpt_i[i] = m.ckpt[i] \ \wedge \ m.taken[i]),$$
where :

- $sent\_to_i[1 \ldots n]$ is a boolean array. $sent\_to_i[k]$ is *true* iff p_i has sent messages to process p_k since its last checkpoint.
- lc_i is an integer that represents a Lamport's logical clock managed by process p_i. When p_i sends a message m, the current value of lc_i is included in m (denoted by $m.lc$).
- $greater_i[1 \ldots n]$ is a boolean array. $greater_i[k]$ is true iff $lc_i > lc_k$. $greater_i[i]$ always keeps a false value. This array is updated as follows:
  - When p_i takes a (local or forced) checkpoint, for each $k \neq i$, $greater_i[k]$ is set to true. When p_i sends a message m, this array is included in m (denoted as $m.greater[]$).
  - When p_i receives a message m, it performs the following updates:

case
$m.lc > lc_i \ \rightarrow$
$\forall k \neq i$ **do** $greater_i[k] := m.greater[k]$;**enddo**
$m.lc = lc_i \ \rightarrow$
$\forall k$ **do**
$greater_i[k] := greater_i[k] \wedge m.greater[k]$;
enddo
$m.lc < lc_i \rightarrow$ **skip**
endcase

- $ckpt_i[1 \ldots n]$ is a vector clock [8] that counts how many checkpoints have been taken by each process. $ckpt_i[k]$ is the number of checkpoints taken by p_k to p_i's knowledge. When p_i sends a message m, this vector is included in m (denoted as $m.ckpt[]$).
- $taken_i[1 \ldots n]$ is a boolean array. $taken_i[k]$ 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. This array is managed in the following way:
  - When p_i takes a checkpoint, for each $k \neq i$, $taken_i[k]$ is set to true. When p_i sends a message m, this array is included in m (denoted as $m.taken[]$).
  - When p_i receives m, it updates $taken_i[]$ in the following way:

$\forall k \neq i$ **do**
case
$m.ckpt[k] > ckpt_i[k] \ \rightarrow$
$taken_i[k] := m.taken[k]$;
$m.ckpt[k] = ckpt_i[k] \ \rightarrow$
$taken_i[k] := (m.taken[k] \vee taken_i[k])$;
$m.ckpt[k] < ckpt_i[k] \rightarrow$ **skip**
endcase
enddo

The condition $\mathscr{C}2''$ can be organized in three parts, and expressed as follows:

$$\mathscr{C}2'' \equiv (FI_a \wedge FI_b) \vee FI_c,$$
where :
$FI_a \equiv (\exists k : sent\_to_i[k] \wedge m.greater[k])$
$FI_b \equiv m.lc > lc_i$
$FI_c \equiv ckpt_i[i] = m.ckpt[i] \ \wedge \ m.taken[i]$

The aim of FI_a and FI_b is to detect non-causal z-paths, while FI_c is oriented to identify causal z-paths (see Fig. 3).

3.2 The Initial S-FI Forced Checkpoint Condition

To capture the same behavior as $\mathscr{C}2''$ leveraging the IDR, we define an initial forced checkpoint condition called $\mathscr{D}$. The condition $\mathscr{D}$ has two main differences with respect

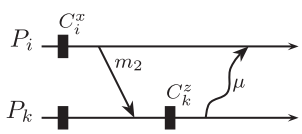

Fig. 3 A causal z-path.

to $\mathscr{C}2''$. First, the vector $ckpt_i[]$, which has a monotonic strictly increasing behavior, is replaced in S-FI by a vector that presents a non-constant monotonic increasing strictly behavior denoted $lc\_ckpt_i[]$. Secondly, the boolean array $taken_i[]$, used in FI, is replaced by the boolean array $idr\_ckpt[]$. Through $idr\_ckpt[]$ we identify if a pair of consecutive checkpoints taken by a process is IDR related. Two local consecutive IDR related checkpoints means that: a) there is a causal z-path between such checkpoints; b) there is not an intermediate checkpoint between them. On the other hand, if two consecutive checkpoints are not IDR related, this indicates that there is a causal z-path with an intermediate checkpoint between them. We are interested in this last behavior since this indicates that a z-cycle (see Definition 7) is detected.

The condition $\mathscr{D}$ is defined as follows.

$$\mathscr{D} \equiv (SFI_a \wedge SFI_b) \vee SFI_c,$$
$$where:$$
$$SFI_a \equiv (\exists k : sent\_to_i[k] \wedge m.greater[k])$$
$$SFI_b \equiv \mathbf{max}(m.lc\_ckpt) > lc_i$$
$$SFI_c \equiv lc\_ckpt_i[i] = m.lc\_ckpt[i] \wedge \neg m.idr\_ckpt[i]$$

SFI_a and SFI_b have the same aim as FI_a and FI_b of $\mathscr{C}2''$, respectively. SFI_c as well as FI_c is used to detect the causal z-paths (see Fig. 3), with the difference that SFI_c is based on IDR checkpoint dependencies. We note that SFI_b and SFI_c share the structure $lc\_ckpt[]$. This avoids the inclusion of the sender's logical clock at the emission of a message m as is detailed below. The variables and data structures used by $\mathscr{D}$ are the following:

- The array $sent\_to_i[]$ and the vector $greater_i[]$ have the same meaning and management as in $\mathscr{C}2''$.
- lc_i is the same Lamport's clock used by $\mathscr{C}2''$; however, it is not included in the messages sent by p_i.
- $lc\_ckpt_i[1 \ldots n]$ is a vector of logical clocks. $lc\_ckpt_i[i]$ has the value of logical clock lc_i when p_i takes its last checkpoint. $lc\_ckpt_i[k]$ has the value of the logical clock lc_k when p_k takes its last checkpoint to p_i's knowlegde. This vector is managed in the following way:
  - When p_i takes a checkpoint, it increments by one the current value of lc_i and the result is assigned to $lc\_ckpt_i[i]$. When p_i sends a message m, $lc\_ckpt_i[]$ is included in m (denoted as $m.lc\_ckpt[]$).
  - When p_i receives m, it updates $lc\_ckpt_i[]$ as follows:

```
∀k ≠ i do
 case
  m.lc_ckpt[k] > lc_ckpt_i[k] →
     lc_ckpt_i[k] := m.lc_ckpt[k];
  m.lc_ckpt[k] < lc_ckpt_i[k] → skip
  m.lc_ckpt[k] = lc_ckpt_i[k] → skip
 enddo
```

  max(u) is a function that obtains the maximum value stored in an array u. We note that the sender's logical clock lc_j is determined by the receiver p_i from the array $lc\_ckpt[]$ included in m ($lc_j = max(m.lc\_ckpt[])$).
- $idr\_ckpt_i[1 \ldots n]$ is a boolean array. The value of $idr\_ckpt_i[k]$ is true, if there is an IDR between the last checkpoint of p_k known by p_i and the next checkpoint of p_i.
  This array is managed in the following way:
  - When p_i takes a checkpoint, it sets $idr\_ckpt_i[i]$ to true, and for each $k \neq i$, $idr\_ckpt_i[k]$ is set to false. When p_i sends a message m, it includes the array $idr\_ckpt_i[]$ ($m.idr\_ckpt[]$) to m.
  - When p_i receives a message m, it updates $idr\_ckpt_i[]$ as follows:

```
∀k ≠ i do
 case
  m.lc_ckpt[k] > lc_ckpt_i[k] →
     idr_ckpt_i[k] := m.idr_ckpt[k];
  m.lc_ckpt[k] = lc_ckpt_i[k] →
     idr_ckpt_i[k] := (m.idr_ckpt[k] ∧ idr_ckpt_i[k]);
  m.lc_ckpt[k] < lc_ckpt_i[k] → skip
 endcase
enddo
```

Now we state the equivalence of conditions as follows:

Theorem 2. Condition $\mathscr{D}$ is equivalent to the condition $\mathscr{C}2''$.

The proof of this theorem is given in Appendix A. For our problem, $\mathscr{D} \equiv \mathscr{C}2''$ means that both conditions detect the same patterns; and therefore, they will trigger the same number of forced checkpoints.

3.3 The S-FI Forced Checkpoint Condition with Dynamic Structures

From an algorithmic point of view, to implement the condition $\mathscr{D}$ we need to attach the boolean arrays $greater[]$ and $idr\_ckpt[]$, and the vector $lc\_ckpt[]$ to each message. This implies a constant overhead per message equal to n integers plus $2n$ bits. By using the principles of the IPT2 protocol [9], the condition $\mathscr{D}$ can be evaluated by using only the information about IDR related checkpoints. This implies dynamically determining and adapting the control information to be sent, resulting in a significant reduction in the overhead sent per message. The condition based on IDR dependencies and expressed with dynamic structures is defined as follows:

$$\mathscr{D}' \equiv (SFI'_a \wedge SFI'_b) \vee SFI'_c,$$
$$where:$$
$$SFI'_a \equiv [\exists k : sent\_to_i[k] \ \wedge \ ((\exists y \in m.\psi,\ y.id = k : y.greater) \ \vee \ (\nexists y \in m.\psi,\ y.id = k))]$$
$$SFI'_b \equiv \mathbf{max}(m.\psi) > lc_i$$
$$SFI'_c \equiv (\exists z \in m.\psi,\ z.id = i : lc\_ckpt_i[i] = z.lc\_ckpt \wedge \neg z.idr\_ckpt)$$

The parts SFI'_a, SFI'_b and SFI'_c in the $\mathscr{D}'$ condition correspond to the parts SFI_a, SFI_b and SFI_c of $\mathscr{D}$, respectively. The data structures and variables used in this condition are:

- The array $sent\_to_i[]$, the vector $lc\_ckpt_i[]$ and the logical clock lc_i have the same meaning and management as in $\mathscr{D}$ condition.
- $m.\psi$ is a data structure made up by tuples. Each tuple contains: a process identifier id, a logical clock lc_ckpt, and two boolean values idr_ckpt and $greater$ ($tuple \equiv (id, lc\_ckpt, idr\_ckpt, greater)$). $m.\psi$ is constructed from the structures $lc\_ckpt_i[]$, $idr\_ckpt_i[]$ and $greater_i[]$ and therefore it is a partial or full copy of such structures. A detailed description of the construction of $m.\psi$ is presented below.
  The function $\mathbf{max}(m.\psi)$ gets the maximum logical clock ($y.lc\_ckpt$) included in some tuple $y \in m.\psi$.

For the problem of immediate predecesors tracking, to identify the control information that a process p_i requires to include in m, Anceaum et al. in [9] defined the abstract condition $K(m,k)$ and the condition $K2(m,k)$. K identifies which entries from the vectors are not neccessary to be piggybacked in a message m, and $K2$ is an implementation that aproximates K, which can be locally evaluated by a process. Based on $K2$, we define the condition $K3$ that is also an aproximation of the abstract condition K. $K3$ is oriented to satisfy $\mathscr{D}'$ and it is defined as follows:

Definition 8.

$$K3(m,k) \equiv (send(m).lc\_ckpt_i[k] = 0) \vee ((send(m).T_i[j,k] = 1) \wedge (send(m).idr\_ckpt_i[k] = 1))$$

where $send(m).lc\_ckpt_i[k]$ and $send(m).idr\_ckpt_i[k]$ are the k-ths values of logical clock vector and boolean array of p_i, respectively, when it sends m. T_i is a boolean matrix that satisfies the following property:

Property 1. For each message sent by p_i to p_j,

$$(send(m).T_i[j,k] = 1) \Rightarrow (send(m).lc\_ckpt_i[k] \leq pred(receive(m)).lc\_ckpt_j[k]) \wedge (\mathbf{max}(send(m).lc\_ckpt_i[]) > send(m).lc\_ckpt_i[k])$$

where $pred(receive(m))$ denotes the checkpoint event C_j^x immediately preceding the reception of m in the sequence H_j. We note that $pred(receive(m)).lc\_ckpt_j[k])$ is the most recent value $lc\_ckpt_j[k]$ known by p_i at the moment of $send(m)$.

When $send(m).T_i[j,k] = 1$ means that process p_i does not know more recent information than p_j with respect to process p_k.

In general, when $K3$ is $true$ means that the tuple $(k, lc\_ckpt[k], idr\_ckpt[k], greater[k]))$ is useless with respect to the correct management (updating process) at p_j of $lc\_ckpt_j[k]$, $idr\_ckpt_j[k]$ and $greater_j[k]$, and therefore it must not be piggybacked on m. The proof of $K3(m,k) \Rightarrow K(m,k)$ is presented in Appendix B (see Theorem 4).

In order to satisfy Property 1, matrix T_i is managed as follows:

T0 T_i is initialized to true. $\forall(j,k) : T_i[j,k] := 1$.

T1 When p_i takes a checkpoint, it resets the i-th column of its matrix T_i. $\forall j \neq i : T_i[j,i] := 0$. When p_i sends a message, the matrix T_i is not updated.

T2 When p_i receives a message m from p_j, T_i is updated as follows:

```
∀w ∈ m.ψ do
case
 w.lc_ckpt > lc_ckpt_i[w.id] →
    ∀ℓ ≠ i do T_i[ℓ, w.id] := 0;
    if (max(m.ψ) > w.lc_ckpt) ∨ (lc_i > w.lc_ckpt)
     then T_i[j, w.id] := 1;
    endif
 w.lc_ckpt = lc_ckpt_i[w.id] →
    if (max(m.ψ) > w.lc_ckpt) ∨ (lc_i > w.lc_ckpt)
     then T_i[j, w.id] := 1;
    endif
 m.lc_ckpt[k] < lc_ckpt_i[k] → skip
endcase
enddo
```

In Appendix B the proof that Property 1 is accomplished by the previous updating process (see Lemma 4) is presented.
Now we state the equivalence of conditions as follows:

Theorem 3. Condition $\mathscr{D}'$ is equivalent to the condition $\mathscr{D}$.

The proof of this theorem is given in Appendix B. In addition, numerical results are presented in Sect. 4 that attest that for all cases the S-FI's $\mathscr{D}'$ condition triggers the same number of forced checkpoints as $\mathscr{C}2''$.

3.4 Description of the S-FI Algorithm

S-FI is composed by three parts: ω_o, ω_1 and ω_2 (see Table 1). The part ω_0 initializes the logical clock lc_i, as well as the data structures $lc\_ckpt_i[]$, $idr\_ckpt_i[]$, $greater_i[]$ and $T_i[][]$ described in Sect. 3.2 and 3.3 (see lines 2-6, Table 1). In addition, it takes the initial checkpoint at a process p_i. In part ω_1, when a message m is sent to a process p_j, the boolean array $sent\_to_i[j]$ is updated, the set ψ_i is constructed (see lines 9-19, Table 1) and included in m. In part ω_2, the reception of messages is managed. ω_2 updates the data structures according to the piggybacked IDR information (see lines 27-52, Table 1). Finally, in ω_2, the forced checkpoint condition $\mathscr{D}'$ is evaluated to determine if p_i should take a forced checkpoint (see lines 22-24, Table 1).

The overhead per message of S-FI is determined by the amount of tuples in ψ (lines 11 to 19). Each tuple is formed by a process identifier (one integer), a logical clock (one integer) and two boolean values (two bits). If an integer is represented by s bits, and t tuples are sent, then for each message we have $t(2s+2)$ bits. Therefore, in the best case, $(2s+2)$ bits are sent; in the average case, $1/(n-1)\sum_{i=1}^{n-1}(i)(2s+2)$ bits are sent; and in the worst case, $(n-1)(2s+2)$ bits are sent. Nevertheless, for the worst case, it is better to send all the information of the static data structures ($n(s+2)$ bits). In Table 2, we show the results of this brief analysis and the overhead messages for the algorithms FI and FINE.

Table 1 S-FI algorithm.

```
    (ω0) Initialization of process p_i.
 1  k, l : 1 ... n, where n is the number of processes.
 2  ∀k do lc_ckpt_i[k] := 0; enddo
 3  ∀k, l do T_i[k, l] := true; enddo
 4  idr_ckpt_i[i] := true;
 5  greater_i[i] := false;
 6  lc_i := 0;
 7  taken_checkpoint();
    (ω1) When p_i sends a message m to p_j.
 8  sent_to_i[j] := true;
 9  ψ_i ← ∅;
10  ∀k do
11    if [(¬T_i[j, k] ∨ ¬idr_ckpt_i[k]) ∧ (lc_ckpt_i[k] > 0)] then
12      ψ_i ← ψ_i ∪ (k, lc_ckpt_i[k], idr_ckpt_i[k], greater_i[k]);
13    endif
14  enddo
15  s := 32; //s is the #-bits to represent a logical clock(lc_ckpt_i).
    //size(ψ_i) returns the cardinality of ψ_i.
16  if size(ψ_i) > (n)(s + 2)/(2s + 2) then
17   ψ_i ← ∅;
18   ∀k do ψ_i ← ψ_i ∪ (−, lc_ckpt_i[k], idr_ckpt_i[k], greater_i[k]); enddo
19  endif
20  send(m:=(ψ_i, Data)) to p_j;
    (ω2) When p_i receives the message m := (ψ, Data) from p_j.
21  max_lc_ckpt := max(ψ);
22  if [(∃k : sent_to_i[k] ∧ (∃y ∈ ψ, y.id = k : y.greater ∨
23      ∄y ∈ ψ, y.id = k)) ∧ max_lc_ckpt > lc_i] ∨
24    [∃z ∈ ψ, z.id = i : lc_ckpt_i[i] = z.lc_ckpt ∧ ¬z.idr_ckpt]
25     then take_checkpoint();
26  endif
27  ∀w ∈ ψ do
28   case
29    w.lc_ckpt > lc_ckpt_i[w.id] →
30      lc_ckpt_i[w.id] := w.lc_ckpt;
31      idr_ckpt_i[w.id] := w.idr_ckpt;
32      ∀ l ≠ i do T_i[l, w.id] := false; enddo
33      if (max_lc_ckpt ≠ w.lc_ckpt) ∨ (lc_i > w.lc_ckpt) then
34        T_i[j, w.id] := true;
35      endif
36    w.cl_ckpt = cl_ckpt_i[w.id] →
37      idr_ckpt_i[w.id] := (idr_ckpt_i[w.id] ∧ w.idr_ckpt);
38      if (max_lc_ckpt ≠ w.lc_ckpt) ∨ (lc_i > w.lc_ckpt) then
39        T_i[j, w.id] := true;
40      endif
41    w.cl_ckpt < cl_ckpt_i[w.id] → skip
42   endcase
43  enddo
44  case
45   max_lc_ckpt > lc_i →
46     lc_i := max_lc_ckpt;
47     ∀k ≠ i do greater_i[k] := true; enddo
48     ∀ℓ ∈ ψ, ℓ.id ≠ i do greater_i[ℓ.id] := ℓ.greater; enddo
49   max_lc_ckpt = cl_i →
50     ∀ℓ ∈ ψ do greater_i[ℓ.id] := greater_i[ℓ.id] ∧ ℓ.greater; enddo
51   max_lc_ckpt < lc_i → skip
52  endcase
53  delivery(m);
    Procedures and functions used in S-FI.
    //When p_i takes a local or forced checkpoint.
54  procedure taken_checkpoint()
55   ∀k do sent_to_i[k] := false; enddo
56   ∀ k ≠ i do
57     idr_ckpt_i[k] := false;
58     greater_i[k] := true;
59     T_i[k, i] := false;
60   enddo
61   lc_i := lc_i + 1;
62   lc_ckpt_i[i] := lc_i;
63  endprocedure
    // max(α) gets the maximum logical clock in α.
64  function max(α)
65  max := 0;
66  ∀x ∈ α do
67   if x.lc_ckpt > max then max := x.lc_ckpt; endif
68  enddo
69  endfunction
```

Table 2 *Overhead* per message (bits) to S-FI, FI and FINE.

Algorithm	Best-Case	Average-Case	Worst-Case
S-FI	$2s+2$	$(n)(s+1)$	$(n)(s+2)$
FI	$(n)(s+2)+s$		
FINE	$(n)(s+1)$		

s-number of bits to represent an integer.
n-number of processes.

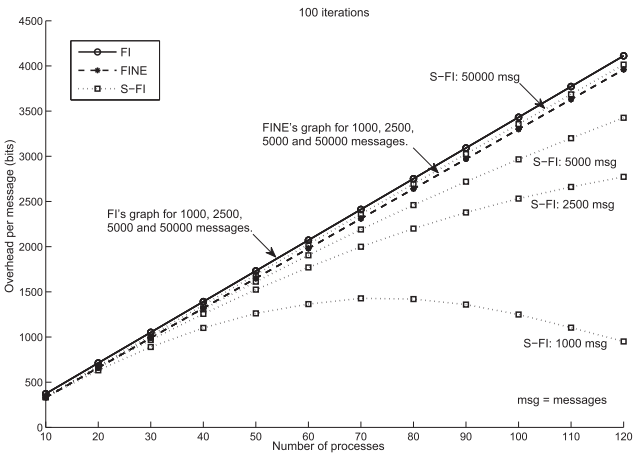

Fig. 4 Simulation results.

4. Simulation Results

We compare the performance of S-FI versus two checkpointing algorithms: FI and FINE [10]. We chose FINE since it is a recent algorithm also based on FI.

The algorithms S-FI, FI and FINE were simulated and analyzed using the simulator for distributed checkpointing ChkSim [11]. ChkSim follows a deterministic simulation model that allows us to reproduce a simulation as many times as necessary and compare two or more algorithms. For the analysis, we use two metrics: the number of forced checkpoints and the overhead per message.

The performance was analyzed for four scenarios of 1000, 2500, 5000 and 50000 messages, with a uniform distribution among the send events, and by varying the number of processes from 10, 20,..., 120. For each scenario, 100 iterations were executed with different communication and checkpoint patterns.

In Fig. 4 we can observe that the overhead per message presented by S-FI is dynamic since it depends on the density of messages and not on the number of processes. Instead, the overhead of FI and FINE present a constant linear growth according to the number of processes. Furthermore, the overhead per message of S-FI has an under linear growth where the upper limit is determined by the FI overhead.

On the other hand, S-FI and FI generate the same number of forced checkpoints, while FINE generates a lower amount [10] that represents on average only a 3% gain with respect to FI.

5. Conclusions

In this article the S-FI checkpointing algorithm was presented. The S-FI algorithm was compared with the FI and FINE algorithms. The results show that the overhead per message presented by S-FI is scalable because it presents an under-linear growth as the number of processes and/or the message density increase. Instead, the overhead of FI and FINE are not scalable since they present a constant linear growth according to the number of processes. On the other hand, the results show that S-FI and FI generate the same number of forced checkpoints.

References

[1] E.N.M. Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," ACM Comput. Surv., vol.34, pp.375–408, Sept. 2002.
[2] B. Randell, "System structure for software fault-tolerance," SIGPLAN Not., vol.10, pp.437–449, April 1975.
[3] R.H.B. Netzer and J. Xu, "Necessary and sufficient conditions for consistent global snapshots," IEEE Trans. Parall. Distrib. Syst., vol.6, no.2, pp.165–169, Feb. 1995.
[4] L. Alvisi, S. Rao, S.A. Husain, A. de Mel, and E. Elnozahy, "An analysis of communication-induced checkpointing," Proc. 29th Annu. Internatio. Sympo. Fault-Tolerant Comp., FTCS '99, pp.242–249, IEEE Computer Society, 1999.
[5] J.M. Hélary, A. Mostefaoui, R.H.B. Netzer, and M. Raynal, "Communication-based prevention of useless checkpoints in distributed computations," Distributed Computing, vol.13, pp.29–43, Jan. 2000.
[6] J. Tsai and J.W. Lin, "On the fully-informed communication-induced checkpointing protocol," Proc. 11th Pacific Rim International Symposium on Dependable Computing, pp.151–158, IEEE Computer Society, 2005.
[7] S.E. Pomares, J. Fanchon, and K. Drira, "The immediate dependency relation: an optimal way to ensure causal group communication," Annu. Rev. Scal. Compt., Ser. Scal. Compt., 6, pp.61–79, 2004.
[8] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Commun. ACM, vol.21, pp.558–565, July 1978.
[9] E. Anceaume, J.M. Hélary, and M. Raynal, "Tracking immediate predecessors in distributed computations," Proc. 40th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '02, pp.210–219, ACM, 2002.
[10] Y. Luo and D. Manivannan, "Fine: A fully informed and efficient communication-induced checkpointing protocol for distributed systems," J. Parallel and Distributed Computing, vol.69, pp.153–167, Feb. 2009.
[11] G.M.D. Vieira and L.E. Buzato, "Chksim: A distributed checkpointing simulator," Tech. Rep. IC-05-34, Institute of Computing, University of Campinas, Dec. 2005.

Appendix A

Theorem 2. Condition $\mathscr{D}$ is equivalent to the condition $\mathscr{C}2''$.

Proof. We divide the proof into two parts. In the first part we demonstrate that FI_b is equivalent to SFI_b; and in the second part, we demonstrate that FI_c is equivalent to SFI_c. We do not demonstrate that FI_a is equivalent to SFI_a because both manage and modify the arrays $sen\_to_i[]$ and $greater_i[]$ in the same way.

Part I. To demonstrate that FI_b is equivalent to SFI_b, we formulate and prove the following Lemma:

Lemma 1. The *x-th* value of the logical clocks $lc^x_{i_{(FI)}}$ and $lc^x_{i_{(SFI)}}$ of a process p_i for FI_b and SFI_b, respectively, are equal. In other words:

$$\forall i \in P : \; lc^x_{i_{(FI)}} = lc^x_{i_{(SFI)}}$$

Proof of Lemma 1. Using induction, we have:

- *Base case* ($k = 2$): at the beginning, these variables are initialized to 1. For the second value of $lc^x_{i_{(FI)}}$ and $lc^x_{i_{(SFI)}}$, we have two cases: The first case is when process p_i takes a checkpoint and updates its lc_i. The second case is when p_i receives a message m and updates its lc_i according to the piggybacked information in m.
  - *p_i takes a checkpoint*. p_i updates its lc_i as follows:

    $$lc^2_{i_{(FI)}} := lc^1_{i_{(FI)}} + 1, \;\; lc^2_{i_{(SFI)}} := lc^1_{i_{(SFI)}} + 1,$$
    $$lc\_ckpt^2_i[i] := lc^2_{i_{(SFI)}} = 2$$

    Therefore, $lc^2_{i_{(FI)}} = lc^2_{i_{(SFI)}} = 2$.
  - *p_i receives m from p_j immediatly after of its first checkpoint and $m.lc = 2$*. In this case, in FI p_i updates $lc^2_{i_{(FI)}}$ in the following way:

    $$if \; m.lc_{(FI)} > lc^1_{i_{(FI)}} \; then \; lc^2_{i_{(FI)}} := m.lc_{(FI)}$$

    Therefore, $lc^2_{i_{(FI)}}$ is updated with the greatest logical clock seen by p_i and p_j.
    In S-FI, $lc^2_{i_{(SFI)}}$ is also updated with the greatest logical clock seen by p_i and p_j, with the difference that the greatest logical clock seen by p_j is extracted from the vector $lc\_ckpt[]$ included in m. $lc^2_{i_{(SFI)}}$ and $lc\_ckpt_i[]$ are updated as follows:

    $$if \; \mathbf{max}(m.lc\_ckpt[]) > lc^1_{i_{(SFI)}} \; then$$
    $$lc^2_{i_{(SFI)}} := \mathbf{max}(m.lc\_ckpt[])$$
    $$\forall \; l \neq i : if \; m.lc\_ckpt[l] > lc\_ckpt_i[l] \; then$$
    $$lc\_ckpt_i[l] := m.lc\_ckpt[l]$$

    Therefore, $lc^2_{i_{(FI)}} = lc^2_{i_{(SFI)}} = 2$, since for both FI and S-FI, each process locally updates its logical clock in the same way.
- *Inductive step:* we assume now that the result holds for $k > 2$, thus: $lc^k_{i_{(FI)}} = lc^k_{i_{(SFI)}}$.
- *Inductive hypothesis*: we will prove that it holds for $k + 1$. This part of the proof is divided into two cases. The first case is when a p_i takes a checkpoint and updates its logical clock. The second case is when a p_i receives a message m and updates its logical clock according to the piggybacked information included in m.
  - *p_i takes a checkpoint*. Therefore, p_i updates its logical clock in the following way:

    $$lc^{k+1}_{i_{(FI)}} := lc^k_{i_{(FI)}} + 1, \;\; lc^{k+1}_{i_{(SFI)}} := lc^k_{i_{(SFI)}} + 1,$$
    $$lc\_ckpt^{k+1}_i[i] := lc^{k+1}_{i_{(SFI)}}$$

    Therefore, $lc^{k+1}_{i_{(FI)}} = lc^{k+1}_{i_{(SFI)}}$.

- p_i *receives a message m from* p_j. We note that in the algorithm FI, the $lc_{j_{(FI)}}(j \neq i)$ included in a message m ($m.lc_{(FI)}$) corresponds to the greatest clock seen by p_j. In this case, $lc^{k+1}_{i_{(FI)}}$ is updated in the following way:

$$if\ m.lc_{(FI)} > lc^{k}_{i_{(FI)}}\ then\ lc^{k+1}_{i_{(FI)}} := m.lc_{(FI)}$$

Therefore, $lc^{k+1}_{i_{(FI)}}$ is updated with the greatest logical clock seen by p_i and p_j.

In the S-FI algorithm, $lc_{j_{(SFI)}}$ is also the greatest logical clock seen by p_j, but in this case, it is included in the vector $lc\_ckpt_j[]$ of m, ($lc_{j_{(SFI)}} \in m.lc\_ckpt[]$). The logical clock $lc^{k+1}_{i_{(SFI)}}$ and the vector $lc\_ckpt_i[]$ are updated in the following way:

$$\begin{aligned} &if\ \mathbf{max}(m.lc\_ckpt[]) > lc^{k}_{i_{(SFI)}}\ then \\ &\qquad lc^{k+1}_{i_{(SFI)}} := \mathbf{max}(m.lc\_ckpt[]) \\ &\forall\ l \neq i : if\ m.lc\_ckpt[l] > lc\_ckpt_i[l]\ then \\ &\qquad lc\_ckpt_i[l] := m.lc\_ckpt[l] \end{aligned}$$

Therefore, $lc^{k+1}_{i_{(SFI)}}$ is also updated with the greatest $lc_{(SFI)}$ seen by p_i and p_j, while the vector $lc\_ckpt_i[]$ is updated also with the greatest $lc_{(SFI)}$.

Therefore, $lc^{k+1}_{i_{(FI)}} = lc^{k+1}_{i_{(SFI)}}$. $\square_{Lemma\ 1}$

Proposition 1. As a consequence of Lemma 1 and the inductive proof, we can state that:

$$lc^{x}_{i_{(FI)}} = (lc^{x}_{i_{(SFI)}} = \mathbf{max}(lc\_ckpt_i[]))$$

Now using the Lemma 1 and the Proposition 1, we can state that: $m.lc > lc_i \equiv \mathbf{max}(m.lc\_ckpt) > lc_i$.
Therefore, $FI_b \equiv SFI_b$. $\square$

Part II. Now we will prove that FI_c is equivalent to SFI_c. We divide this proof into two parts. In the first part, we prove that $lc\_ckpt_i[i]$ has a similar behavior as $ckpt_i[i]$ during an interval. For the second part we show that by identifying the immediate dependency relations among checkpoints, we can detect the same pattern than the array $taken_i[]$ of FI algorithm.

- *Part II.A*. Since the logical clock $ckpt_i[i]$ has a strictly increasing behavior, we prove that the logical clock $lc\_ckpt_i[i]$ has the same property.

Lemma 2. The logical clock $lc\_ckpt_i[i]$ of process p_i has a strictly increasing behavior, as follows:

$$\forall i \in P:\ lc\_ckpt^{1}_{i}[i] < \cdots < lc\_ckpt^{x-1}_{i}[i] < lc\_ckpt^{x}_{i}[i],$$

where x represents the *x-th* taken checkpoint of p_i.

Proof Lemma 2. This part is demonstrated by direct proof. We note that $lc_{i_{(FI)}}$ has a strictly increasing behavior [5]. From Lemma 1 we have that $lc^{x}_{i_{(SFI)}} = lc^{x}_{i_{(FI)}}$; therefore the logical clock $lc_{i_{(SFI)}}$ also has the same behavior. Since $lc\_ckpt^{x}_{i}[i]$ is set to $lc^{x}_{i_{(SFI)}}$, for each taken checkpoint at process p_i, we have that the logical clock $lc\_ckpt_i[i]$ has a strictly increasing behavior. $\square_{Lemma\ 2}$

Now using Lemma 2 and knowing that the logical clock $lc\_ckpt_i[i]$ is only updated when p_i takes a local checkpoint, we can state that the logical clock $lc\_ckpt^{k}_{i}[i]$ is constant during at an interval.

- *Part II.B*. In FI algorithm $taken_i[j] = true$ indicates that there is a causal zigzag path, including a checkpoint, from the last checkpoint C^{y}_{j}, known by p_i, to the next checkpoint C^{x+1}_{i}. Specifically we are interested when $taken_i[j] = true$ and $j = i$ since there is a causal zigzag path that includes a checkpoint C^{z}_{k} in the interval defined by the checkpoints C^{x}_{i} and C^{x+1}_{i}. For this, we state in the Lemma 3 that the S-FI detects this pattern by identifying the immediate dependency relations among checkpoints.

Lemma 3. For a message m sent by p_j and received at p_i, $i \neq j$

$$\begin{aligned} &if\ (m.idr\_ckpt[i] = false)\ then \\ &\quad \exists C^{z}_{k} \in R, k \neq i : C^{x}_{i} \rightarrow C^{z}_{k} \rightarrow C^{x+1}_{i} \end{aligned}$$

For Lemma 3, we give a sketch of proof. According to definition 3, we have that two checkpoints C^{x}_{i} and C^{x+1}_{i} are IDR related if $\nexists C^{z}_{k} : C^{x}_{i} \rightarrow C^{z}_{k} \rightarrow C^{x+1}_{i}$.
During the message exchange between C^{x}_{i} and C^{x+1}_{i}, in S-FI the value $idr\_ckpt_i[i] = true$ is propagated between each pair of consecutive messages iff a checkpoint C^{z}_{k} does not exist. This is accomplished since at the reception of a message, the vector $idr\_ckpt_i[]$ is updated with the last IDR information (see updating process of message reception for $idr\_ckpt_i[]$, page 889). Otherwise, when a local checkpoint is taken, the IDR history of p_k with respect to p_i is erased by reinitializing $idr\_ckpt_k[i] = false$ (see the updating process for $idr\_ckpt_i[]$, page 889).

Therefore $FI_c \equiv SFI_c$. $\square$

Appendix B

Theorem 3. Condition $\mathscr{D}'$ is equivalent to the condition $\mathscr{D}$.

Proof. We divide the proof into two parts. First, we demonstrate that the condition $K3(m,k)$ implies $K(m,k)$ which ensures the tracking of checkpoints that are immediate predecesors without requiring to piggyback the whole control information in each message. Secondly, we demonstrate that $SFI_a \wedge SFI_b$ is equivalent to $SFI'_a \wedge SFI'_b$ and SFI_c is equivalent to SFI'_c.

Part I. To demonstrate that $K3(m,k)$ implies $K(m,k)$ we state and prove Theorem 4.

Theorem 4. Let $K3(m,k) \equiv (send(m).lc\_ckpt_i[k]=0)$ $\vee$ $((send(m).T_i[j,k] = 1) \wedge (send(m).idr\_ckpt_i[k]=1))$. We have: $K3(m,k) \Rightarrow K(m,k)$.

where, the abstract condition $K(m,k)$ was defined by Anceaume et al. in [9] as follows:

$$\begin{aligned} K(m,k) \equiv\ &(send(m).VC_i[k] = 0) \\ &\vee (send(m).VC_i[k] < pred(receive(m)).VC_j[k]) \\ &\vee ((send(m).VC_i[k] = pred(receive(m)).VC_j[k]) \\ &\quad \wedge (send(m).IP_i[k] = 1)) \end{aligned}$$

here, $VC_i[]$ is a vector of logical clocks and $IP_i[]$ is a boolean array.

Lets consider the following. The management (updating process) of $VC_i[]$ and $IP_i[]$ is equal to the management of the vector $ckpt_i[]$ of FI (see Sect. 3.1) and the boolean array $idr\_ckpt_i[]$ of our proposal, respectively. We recall that the vector $ckpt_i[]$ was replaced in S-FI by the vector $lc\_ckpt_i[]$ without affecting the desired results as is demonstrated in Theorem 2. Specfically, it was demonstrated that the logical clocks of the vector $lc\_ckpt_i[]$ present a strictly increasing behaviour as well as the logical clocks of the vector $ckpt_i[]$ and consequently, as the logical clocks of the vector $VC_i[]$ (see Lemma 1 and Lemma 2) as well. Taking into account these comments, we present the proof of Theorem 4 as follows.

Proof. We begin by showing that the matrix T_i provides a correct meaning to p_i's knowledge. $(send(m).T_i[j,k] = 1) \Rightarrow ((send(m).lc\_ckpt_i[k] \leq pred(receive(m)).lc\_ckpt_j[k]) \wedge (max(send(m).lc\_ckpt_i[]) > send(m).lc\_ckpt_i[k]))$.

Lemma 4. Let $\mathcal{IT}(e,j,k)$ the following property:

$$(e.T_i[j,k] = 1) \Rightarrow A0 \vee ((A1 \vee A2 \vee A3) \wedge B0),$$

where :

$A0 \equiv (j{=}i)$, $A1 \equiv (j{=}k)$, $A2 \equiv (e.lc\_ckpt_i[k]{=}0)$,

$A3 \equiv \exists\, m'$ *from* p_j *to* $p_i, \forall z \in m'.\psi, k = z.id$:

$((receive(m') \rightarrow e) \vee (receive(m') = e)) \wedge$

$(send(m').lc\_ckpt_j[k] = z.lc\_ckpt = e.lc\_ckpt_i[k])$,

$B0 \equiv (max(e.lc\_ckpt_i[]) > e.lc\_ckpt_i[k])$.

$\forall i, \forall e \in H_i, \forall j, \forall k$: $\mathcal{IT}(e,j,k)$ holds.

Proof. The proof is by induction on $\hat{E}$. We consider only the events e such that $e.T_i[j,k] = 1$. When $e.T_i[j,k] = 0$, the property $\mathcal{IT}(e,j,k)$ trivially holds.

- *Base case*: let e be the first event of p_i. We have that $e.T_i[j,k] = 1$ only in the following cases:
  - e is the first checkpoint of p_i. Thus, from T0, T1 and the management of $lc\_ckpt_i[]$ (see, Sect. 3.2 and 3.3); we have that $max(e.lc\_ckpt_i[]) = (e.lc\_ckpt_i[i]) = 1$ and:
    - $j = i, \forall k : (T_i[j,k] = 1) \Rightarrow A0$.
    - $\forall j \neq i, \forall k \neq i : (T_i[j,k] = 1) \Rightarrow A2 \wedge B0$.
    - $\forall j \neq i, k = i : T_i[j,k] = 0$.
  - e is the reception of a message m from p_j immediatly after the first checkpoint of p_i. Then, from T2 and the management of $lc\_ckpt_i[]$, we have:
    - $j = i, \forall k : (T_i[j,k] = 1) \Rightarrow A0$.
    - $\forall x \neq i, \forall y, \forall z \in m.\psi, k = z.id$: $(e.T_i[x,y] = 1) \Rightarrow [(x = j) \wedge (y = k) \wedge (max(e.lc\_ckpt_i[]) > z.lc\_ckpt)] \vee [(y{\neq}i) \wedge (y{\neq}k) \wedge (lc\_ckpt_i[k]{=}0)]$. First alternative holds. m satisfies $A3$ and $B0$, $receive(m) = e \wedge send(m).lc\_ckpt_j[k] = z.lc\_ckpt = e.lc\_ckpt_i[k] < max(e.lc\_ckpt_i[])$. Second alternative also holds. $e.lc\_ckpt_i[k] = 0$ satisfies $A2$ and $B0$.

  Thus, in every case, $\mathcal{IT}(e,j,k)$ holds.
- *Inductive step*: let $e \in H_i$. We assume that $\forall e' \in \{e' \mid e' \rightarrow e\}, \forall j, \forall k : \mathcal{IT}(e',j,k)$ holds.
- *Inductive hypothesis*: we will prove that $\forall j, \forall k$, the property $\mathcal{IT}(e,j,k)$ holds. We proceed by case analysis about the type of event.
  - e is a checkpoint. p_i resets the *i-th* column of T_i (see T1), $\forall \neq i : T_i[j,i] := 0$.
  - e is a send event. There are no updates in the matrix T_i (see T1). Therefore, $\mathcal{IT}(e,j,k)$ holds.
  - e is the reception of m from p_j. p_i only updates the row j of T_i (see T2). Then: $x = j, \forall z \in m.\psi, z.id = k : (e.T_i[x,k] = 1) \Rightarrow ((x = j) \wedge (max(e.lc\_ckpt_i[]) > z.lc\_ckpt)$. Thus, e satisfies $A3$ and $B0$. $receive(m) = e \wedge send(m).lc\_ckpt_j[k] = z.lc\_ckpt = e.lc\_ckpt_i[k] < max(e.lc\_ckpt_i[])$.

  Thus, in every case, $\mathcal{IT}(e,j,k)$ holds.

$\square_{Lemma\ 4}$

Now, let m be a message sent by p_i to p_j ($e = send(m)$) and $send(m).T_i[j,k] = 1$. From Lemma 4, we have three cases (we note that $j \neq i$ and e never can be a receive event).

- From $A1$, $j = k$. Thus, from the properties of vector clocks, we have: $send(m).lc\_ckpt_i[k] \leq pred(receive(m)).lc\_ckpt_j[k]$.
- From $A2$, $send(m).lc\_ckpt_i[k] = 0$. Then, $send(m).lc\_ckpt_i[k] \leq pred(receive(m)).lc\_ckpt_j[k]$.
- From $A3$, we have: $send(m').lc\_ckpt_j[k] = e.lc\_ckpt_i[k] \leq pred(receive(m)).lc\_ckpt_j[k]$. Hence, $send(m).lc\_ckpt_i[k] \leq pred(receive(m)).lc\_ckpt_j[k]$.

Therefore, $(sent(m).T_i[j,k] = 1) \Rightarrow (send(m).lc\_ckpt_i[k] \leq pred(receive(m)).lc\_ckpt_j[k])$.

Hence, we have:

$K3(m,k) \equiv \quad (send(m).lc\_ckpt_i[k] = 0)$

$\vee((send(m).T_i[j,k] = 1) \wedge (send(m).idr\_ckpt_i[k]{=}1))$

$\Rightarrow \quad (send(m).lc\_ckpt_i[k]{=}0) \vee ((send(m).lc\_ckpt_i[k] \leq pred(receive(m)).lc\_ckpt_j[k]) \wedge (send(m).idr\_ckpt_i[k]{=}1))$

$\Rightarrow \quad (send(m).VC_i[k]{=}0) \vee ((send(m).VC_i[k] \leq pred(receive(m)).VC_j[k]) \wedge (send(m).IP_i[k]{=}1))$

$\equiv K(m,k)$ $\square_{Theorem\ 4}$

Part II.a. In order to demonstrate that $SFI'_a \wedge SFI'_b$ is equivalent to $SFI_a \wedge SFI_b$, we demonstrated by direct proof that: $SFI_a \wedge SFI_b \Rightarrow SFI'_a \wedge SFI'_b$; where:

$SFI_a \equiv (\exists k : sent\_to_i[k] \wedge m.greater[k])$

$SFI_b \equiv (max(m.lc\_ckpt[]) > lc_i)$

$SFI'_a \equiv (\exists k : sent\_to_i[k] \wedge ((\exists y \in m.\psi, y.id = k : y.greater) \vee (\nexists y \in m.\psi, y.id = k)))$

$SFI'_b \equiv (max(m.\psi) > lc_i)$

We note that the value of $sent\_to_i[k]$ is equal for both $\mathscr{D}$ and $\mathscr{D}'$ (seen 3.2 and 3.3), and $max(m.lc\_ckpt[]) = max(lc\_ckpt_j[]) = max(m.\psi)$ (see 3.2 and Lemma 4). In addition, $max(lc\_ckpt_j[])$ is always included in $m.\psi$ (see Lemma 4). Now, let m sent by p_j to p_i and $sent\_to_i[k] = true$. We have two cases to analyze:

- $\exists y \in m.\psi, y.id = k$. In this case, $(SFI_a \wedge SFI_b) \Rightarrow (SFI'_a \wedge SFI'_b)$, holds.
- $\nexists y \in m.\psi, y.id = k$. In this case, from Theorem 4, we also have two cases:

– $send(m).lc\_ckpt_j[k] = 0$. From management of $greater_j[]$ (see, Sect. 3.1) we have: $lc_j \geq send(m).lc\_ckpt_j[j] \geq 1 > send(m).lc\_ckpt_j[k] = 0 = lc_k$; thus, $greater_j[k]$ is true. Therefore, $(SFI_a \wedge SFI_b) \Rightarrow (SFI'_a \wedge SFI'_b)$ holds.

– $(send(m).T_j[i,k] = 1) \wedge (send(m).idr\_ckpt_j[k] = 1)$. Let $e = send(m)$, from Lemmma 4, we have (we note that $j \neq i$ and e is not a receive event):

  – From $A1$, $k = i$. Thus, $send(m).lc\_ckpt_j[k] \leq pred(receive(m)).lc\_ckpt_i[k]$, $max(e.lc\_ckpt_j[]) > e.lc\_ckpt_j[k]$ and $(send(m).idr\_ckpt_j[k] = 1)$. Let $e.lc\_ckpt_j[s] = max(e.lc\_ckpt_j[]) = lc_j$. Then exists a sequence of causal messages $[m_1 \downarrow m_2 \downarrow \ldots \downarrow m_\ell]$ from p_s to p_j. Hence, we have two cases:

    * There is a sequence of causal messages from p_s to p_i and another from p_i to p_j exists. In this case, $lc_i = lc_j = max(e.lc\_ckpt_j[])$. Thus, $SFI_b = SFI'_b = false$.
    * There is not a sequence of messages from p_s to p_i. Thus, $lc_j > lc_i$, then $greater_j[k] = true$.

    Therefore, $SFI_a \wedge SFI_b \Rightarrow SFI'_a \wedge SFI'_b$ holds.
  – From $A2$, $send(m).lc\_ckpt_j[k] = 0$. It was analized previously.
  – From $A3$, $\exists m'$ from p_i to p_j, $\forall z \in m'.\psi, \ldots$
  Let $e.lc\_ckpt_j[s] = max(e.lc\_ckpt_j[]) = lc_j$ and as in the case $k = i$, we have:

    * There is a sequence of causal messages from p_s to p_k and another from p_k to p_j. Thus, $lc_k = lc_i = lc_j = max(e.lc\_ckpt_j[])$. Therefore, $SFI_b = SFI'_b = false$.
    * There is not sequence of causal messages from p_s to p_k. Thus, $lc_j > lc_k$, therefore $greater_j[k] = true$.

    Thus, in all the cases $(SFI_a \wedge SFI_b) \Rightarrow (SFI'_a \wedge SFI'_b)$ holds.

Part II.b. Finally, in order to prove that SFI'_c is equivalent to SFI_c, we demonstrate by direct proof that: $SFI_c \Rightarrow SFI'_c$; where:

$$SFI_c \equiv lc\_ckpt_i[i] = m.lc\_ckpt[i] \wedge \neg m.idr\_ckpt[i]$$
$$SFI'_c \equiv (\exists z \in m.\psi, z.id = i : lc\_ckpt_i[i] = z.lc\_ckpt \wedge \neg z.idr\_ckpt)$$

Proof. In this proof we have two cases to analyze:

- $\exists z \in m.\psi, z.id = i$. In this case $SFI_c \Rightarrow SFI'_c$ holds.
- $\nexists z \in m.\psi, z.id = i$. Thus, SFI'_c is always false for this case. Let $e = send(m)$, from Theorem 4 we have two cases:

  – $send(m).lc\_ckpt_j[k] = 0$. If $k = i$ then we have $\nexists e' \in E$ such that $e' \in H_i \wedge e' \rightarrow e$. Thus, $SFI_c = false$ and SFI'_c is false. Therefore, $SFI_c \Rightarrow SFI'_c$ holds.
  – $(send(m).T_j[i,k] = 1) \wedge (send(m).idr\_ckpt_j[k] = 1)$. Here, if $k = i$ we have that $\exists e' \in E$ such that $e' \in H_i \wedge e' \downarrow e$. Thus, $SFI_c = false$ $(e.idr\_ckpt_j[i] = m.idr\_ckpt[i] = true)$ and $SFI'_c = false$. Therefore, $SFI_c \Rightarrow SFI'_c$ holds.

$\square_{Theorem\ 3}$

Alberto Calixto Simon is currently a Ph.D student in the Department of Computer Science at INAOE. His research interests include partial order and checkpointing algorithms. His post graduate studies are supported by CONACYT, and the University of Papaloapan (UNPA) of Mexico

Saul E. Pomares Hernandez is a researcher in the Department of Computer Science at INAOE, in Mexico. He completed his Ph.D Degree at the LAAS of CNRS, France in 2002. Since 1998, he has been researching in the field of distributed systems and partial order algorithms.

Jose Roberto Perez Cruz is currently a Ph.D student in the Department of Computer Science at INAOE. His research interests include partial order algorithms, sensor networks and secure group communications. His post graduate studies are supported by CONACYT.

Pilar Gomez-Gil received the MSc and PhD degrees from Texas Tech University, USA, in computer science. She is a researcher in computer science at INAOE. Her research interests include the design and use of artificial neural networks, pattern recognition and sensor-related applications. She is an active member of the IEEE and the ACM.

Khalil Drira obtained the Ph.D. and HDR degrees in Computer Science from University Toulouse III, in 1992 and 2005 respectively. He is Research Director at LAAS-CNRS. His research interests include formal design, implementation provisioning of distributed communicating systems and cooperative networked services (see his wiki).