A Scalable Communication-Induced Checkpointing Algorithm for Distributed Systems

Alberto CALIXTO SIMON†, Student Member, Saul E. POMARES HERNANDEZ††, 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 Hélapy 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 \( h_i = e_1^i e_2^i \cdots e_x^i \) 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 \( C^x \) 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-
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 \( \text{send}(m) \) the emission of \( m \) and by \( \text{delivery}(p, m) \) the delivery event of \( m \) to participant \( p \in P \).

The set of events associated to \( M \) is the finite set \( E = \{ \text{send}(m) : m \in M \} \cup \{ \text{delivery}(p, m) : m \in M \wedge p \in P \} \). The whole set of events in the system is the finite set \( E = R \cup E \). The distributed computation is modeled by the partially ordered set \( \bar{E} = (E, \rightarrow) \), 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 \((\bar{E}, R_E)\) where \( \bar{E} \) is a partially ordered set modeling a distributed computation, and \( R_E \) is a set of local checkpoints defined on \( \bar{E} \) [5].

Figure 1 shows an example of a communication and checkpoint pattern. The sequence of events occurring at \( p_i \) between \( C_i^1 \) and \( C_i^{x+1} \) (\( x > 0 \)) 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 \( \text{send}(m) \) and \( b \) is the event \( \text{delivery}(m) \), 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:

\[ a \downarrow b \text{ if } a \rightarrow b \text{ and } \forall c \in E, \neg(a \rightarrow c \rightarrow b) \]

In our context, we are only interested in identifying the immediate dependency relations among the set of relevant events \( R \subset E \), which contains 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 \( C_j^i \) to another \( C_j^i \) iff there are messages \( m_1, m_2, \ldots, m_\ell \) such that:

1. \( m_1 \) is sent by process \( p \) after \( C_j^i \).
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. \( m_\ell \) is received by process \( q \) before \( C_j^i \).

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

**Definition 5.** A \( z \)-path \([m_1, \ldots, m_\ell]\) is causal, iff for each pair of consecutive messages \( m_\alpha \) 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 \( C_j^i \) \( Z \)-depends on a local checkpoint \( C_j^i \) if and only if:

1. \( j = i \) and \( y > x \), or
2. there is a \( z \)-path from \( C_j^i \) to \( C_j^y \).

**Definition 7.** A \( z \)-cycle is a \( Z \)-dependency from a local checkpoint \( C_j^i \) to itself: \( C_j^i \rightarrow Z C_j^i \).

In Fig. 1, the messages \([m_4, m_1]\) form a \( z \)-cycle involving \( C_j^1 \), and \([m_6, m_5, m_4, m_3]\) form a \( z \)-cycle in \( C_j^2 \).

**Theorem 1.** The following properties of a communication and checkpoint pattern \((\bar{E}, R_E)\) are equivalent:

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

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

\(^7\)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.
Theorem 1 and the forced checkpoint condition

3. S-FI Algorithm

The S-FI algorithm is based on the principles introduced in the FI checkpointing protocol proposed by Hélay et al. [5] and the IPT2 tracking protocol [9]. Specifically, S-FI uses Theorem 1 and the forced checkpoint condition $\mathcal{C}''_{2}'$ 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 $\mathcal{D}$. This condition is expressed, as well as $\mathcal{C}''_{2}'$, 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 $\mathcal{D}$ is equivalent to $\mathcal{C}''_{2}'$ to ensure Theorem 1. Then $\mathcal{D}$ is redefined by using dynamic structures and it is called $\mathcal{D}'$. 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 $\mathcal{C}''_{2}'$ 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 $\mathcal{C}''_{2}'$, as shown in [5], ensures Theorem 1. If in the reception of a message at a process $p_j$, the condition $\mathcal{C}''_{2}'$ 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.

\[
\mathcal{C}''_{2}' \equiv (\exists k : \text{sent}_t[k] \land \text{m} \text{.greater}[k]) \land \text{m} \text{.lc} > \text{lc}_j \lor (\text{ckpt}_i[k] = \text{m} \text{.ckpt}[i] \land \text{m} \text{.taken}[i]),
\]

where:

- $\text{sent}_t[0 \ldots n]$ is a boolean array. $\text{sent}_t[k]$ is true iff $p_i$ has sent messages to process $p_k$ since its last checkpoint.
- $\text{lc}_j$ is an integer that represents a Lamport’s logical clock managed by process $p_j$. When $p_i$ sends a message $m$, the current value of $\text{lc}_j$ is included in $m$ (denoted as $\text{m} \text{.lc}$).
- $\text{greater}[1 \ldots n]$ is a boolean array. $\text{greater}[k]$ is true if $\text{lc}_j > \text{lc}_k$. $\text{greater}[i]$ always keeps a false value. This array is updated as follows:

  - When $p_i$ takes a (local or forced) checkpoint, for each $k \neq i$, $\text{greater}[k]$ is set to true. When $p_i$ sends a message $m$, this array is included in $m$ (denoted as $\text{m} \text{.greater}[i]$).
  - When $p_i$ receives a message $m$, it performs the following updates:

\[
\begin{align*}
\forall k \neq i & \text{ do } \text{greater}[k] := \text{m} \text{.greater}[k]; \text{enddo} \\
\forall k & \text{ do } \\
& \text{greater}[k] := \text{greater}[k] \land \text{m} \text{.greater}[k]; \text{enddo} \\
\text{m} \text{.lc} < \text{lc}_i & \text{ do } \text{skip} \text{ enddo}
\end{align*}
\]

- $\text{ckpt}_i[1 \ldots n]$ is a vector clock [8] that counts how many checkpoints have been taken by each process. $\text{ckpt}_i[k]$ 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 $m \text{.ckpt}[i]$).

- $\text{taken}[1 \ldots n]$ is a boolean array. $\text{taken}[k]$ 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 $k \neq i$, $\text{taken}[k]$ is set to true. When $p_i$ sends a message $m$, this array is included in $m$ (denoted as $m \text{.taken}[i]$).
  - When $p_i$ receives a message $m$, it updates $\text{taken}[i]$ in the following way:

\[
\begin{align*}
\forall k \neq i & \text{ do } \\
& \text{case} \\
& m \text{.ckpt}[k] > \text{ckpt}[k] \rightarrow \\
& \text{taken}[k] := \text{m} \text{.taken}[k]; \\
& m \text{.ckpt}[k] = \text{ckpt}[k] \rightarrow \\
& \text{taken}[k] := (\text{m} \text{.taken}[k] \lor \text{taken}[k]); \\
& m \text{.ckpt}[k] < \text{ckpt}[k] \rightarrow \text{skip} \\
& \text{endcase} \\
\text{enddo}
\end{align*}
\]

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

\[
\mathcal{C}''_{2}' \equiv (\text{FI}_a \land \text{FI}_b) \lor \text{FI}_c,
\]

where:

\[
\begin{align*}
\text{FI}_a & \equiv (\exists k : \text{sent}_t[k] \land \text{m} \text{.greater}[k]) \\
\text{FI}_b & \equiv \text{m} \text{.lc} > \text{lc}_j \\
\text{FI}_c & \equiv \text{ckpt}_i[k] = \text{m} \text{.ckpt}[i] \land \text{m} \text{.taken}[i]
\end{align*}
\]

The aim of $\text{FI}_a$ and $\text{FI}_b$ is to detect non-causal z-paths, while $\text{FI}_c$ 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 $\mathcal{C}''_{2}'$ leveraging the IDR, we define an initial forced checkpoint condition called $\mathcal{D}$. The condition $\mathcal{D}$ has two main differences with respect
to ’c2‘. 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 lcckpt[i]. Secondly, the boolean array taken[i], used in FI, is replaced by the boolean array idrckpt[]. Through idrckpt[] 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 $\mathcal{D}$ is defined as follows.

$$\mathcal{D} \equiv (SFI_a \land SFI_b) \lor SFI_c,$$

where:

- $SFI_a \equiv (\exists k : sent_to[i][k] \land m.greater[k])$
- $SFI_b \equiv \max(m.lceckpt) > lci$
- $SFI_c \equiv lcckpt[i] = m.lceckpt[i] \land \neg m.idrckpt[i]$

$SFI_a$ and $SFI_b$ have the same aim as $FIA_a$ and $FIA_b$ of ’c2‘, respectively. $SFI_c$ as well as $FIC_c$ 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 lcckpt[]. 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 $\mathcal{D}$ are the following:

- The array sent_to[i] and the vector greater[i][] have the same meaning and management as in ’c2‘.
- lcj is the same Lamport’s clock used by ’c2‘; however, it is not included in the messages sent by $p_i$.
- lcckpt[1...n] is a vector of logical clocks. lcckpt[i] has the value of logical clock lcj when $p_i$ takes its last checkpoint. lcckpt[k] has the value of the logical clock lcj when $p_k$ takes its last checkpoint to $p_i$’s knowledge. This vector is managed in the following way:
  - When $p_i$ takes a checkpoint, it increments by one the current value of lcj; and the result is assigned to lcckpt[i]. When $p_i$ sends a message $m$, lcckpt[i] is included in $m$ (denoted as $m.lcckpt$).
  - When $p_i$ receives $m$, it updates lcckpt[i] as follows:

$$\forall k \neq i \begin{cases} 
  m.lcckpt[k] > lcckpt[k] \rightarrow \\
  lcckpt[k] := m.lcckpt[k]; \\
  m.lcckpt[k] < lcckpt[k] \rightarrow \text{skip} \\
  m.lcckpt[k] = lcckpt[k] \rightarrow \text{skip}
\end{cases}$$

max(u) is a function that obtains the maximum value stored in an array u. We note that the sender’s logical clock lcj is determined by the receiver $p_i$ from the array lcckpt[] included in $m$ ($lci = \max(m.lcckpt)$).
- idrckpt[1...n] is a boolean array. The value of idrckpt[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 idrckpt[i] to true, and for each $k \neq i$, idrckpt[k] is set to false.
- When $p_i$ sends a message $m$, it includes the array idrckpt[] ($m.idrckpt[]$) in $m$.
- When $p_i$ receives a message $m$, it updates idrckpt[] as following:

$$\forall k \neq i \begin{cases} 
  m.lcckpt[k] > lcckpt[k] \rightarrow \\
  idrckpt[k] := m.idrckpt[k]; \\
  m.lcckpt[k] < lcckpt[k] \rightarrow \text{skip}
\end{cases}$$

Now we state the equivalence of conditions as follows:

**Theorem 2.** Condition $\mathcal{D}$ is equivalent to the condition ’c2‘.

The proof of this theorem is given in Appendix A. For our problem, $\mathcal{D} \equiv 'c2'$ 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 $\mathcal{D}$ we need to attach the boolean arrays greater[] and idrckpt[], and the vector lcckpt[] 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 $\mathcal{D}$ 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:

$$\mathcal{D}' \equiv (SFI'_a \land SFI'_b) \lor SFI'_c,$$

where:

- $SFI'_a \equiv (\exists k : sent_to[i][k] \land (\exists y \in m.\psi, y.id = k : y.greater) \lor (\exists y \in m.\psi, y.id = k))$
- $SFI'_b \equiv \max(m.\psi) > lcj$
- $SFI'_c \equiv (\exists z \in m.\psi, z.id = i : lcckpt[i] = z.lcckpt \land \neg z.idrckpt)$

The parts $SFI'_a$, $SFI'_b$ and $SFI'_c$, in the $\mathcal{D}'$ condition correspond to the parts $SFI_a$, $SFI_b$ and $SFI_c$, of $\mathcal{D}$, respectively. The data structures and variables used in this condition are:
• The array send\_to[i], the vector lc\_ckpt[i] and the logical
  clock lc\_i have the same meaning and management
  as in \( \mathcal{S} \) 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 con-
  structed 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 construc-
  tion of \( m.\psi \) is presented below.

The function \( \text{max}(m.\psi) \) gets the maximum logical
  clock \( y.lc\_ckpt \) included in some tuple \( y \in m.\psi. \)

For the problem of immediate predecessors 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 necessary to be pig-
  gybacked in a message \( m \), and \( K2 \) is an implementation that
  approximates \( K \), which can be locally evaluated by a process.
  Based on \( K2 \), we define the condition \( K3 \) that is also an
  approximation of the abstract condition \( K \). \( K3 \) is oriented to
  satisfy \( \mathcal{S} \) and it is defined as follows:

\textbf{Definition 8.}

\( K3(m, k) \equiv (\text{send}(m).lc\_ckpt[k] = 0) \lor \)
\((\text{send}(m).T_i[j, k] = 1) \land (\text{send}(m).idr\_ckpt[k] = 1)) \)

where \( \text{send}(m).lc\_ckpt[k] \) and \( \text{send}(m).idr\_ckpt[k] \) 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:

\textbf{Property 1.} For each message sent by \( p_i \) to \( p_j \),

\( (\text{send}(m).T_i[j, k] = 1) \Rightarrow \)
\((\text{send}(m).lc\_ckpt[k] \leq \text{pred}(\text{receive}(m)).lc\_ckpt[k]) \land \)
\((\max(\text{send}(m).lc\_ckpt[k]) > \text{send}(m).lc\_ckpt[k]) \)

where \( \text{pred}(\text{receive}(m)) \) denotes the checkpoint event \( C^*_j \)
immediately preceding the reception of \( m \) in the sequence
\( H_j \). We note that \( \text{pred}(\text{receive}(m)).lc\_ckpt[k] \) is the most
recent value \( lc\_ckpt[k] \) known by \( p_i \) at the moment of
\( \text{send}(m) \).

When \( \text{send}(m).T_i[j, k] = 1 \) 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 re-
spect to the correct management (updating process) at \( p_j \)
of \( lc\_ckpt[k], idr\_ckpt[k] \) and \( greater[k] \), 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:

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

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

\textbf{T2} When \( p_i \) receives a message \( m \) from \( p_j \), \( T_j \) is updated as follows:

\begin{align*}
\forall w \in m.\psi & \text{ do} \\
\text{case} & \\
\text{w.lc\_ckpt} > \text{w.lc\_ckpt}[w.id] \rightarrow \\
\forall \ell \not= i & \text{ do } T_j[\ell, w.id] := 0; \\
\text{if } (\max(m.\psi) > \text{w.lc\_ckpt}) & \lor (l_i > \text{w.lc\_ckpt}) \\
\text{then } T_j[j, w.id] := 1; & \\
\text{endif} & \\
\text{w.lc\_ckpt} = \text{w.lc\_ckpt}[w.id] \rightarrow \\
\text{if } (\max(m.\psi) > \text{w.lc\_ckpt}) & \lor (l_i > \text{w.lc\_ckpt}) \\
\text{then } T_j[j, w.id] := 1; & \\
\text{endif} & \\
\text{m.lc\_ckpt}[k] < \text{w.lc\_ckpt}[k] & \rightarrow \text{skip} \\
\text{endcase} & \\
\text{enddo} & \\
\end{align*}

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

\textbf{Theorem 3.} Condition \( \mathcal{S} \) is equivalent to the condition \( \mathcal{G} \).

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 \( \mathcal{G} \) condition triggers the same
number of forced checkpoints as \( \mathcal{G}' \).

3.4 Description of the S-FI Algorithm

S-FI is composed by three parts: \( \omega_0 \), \( \omega_1 \) and \( \omega_2 \) (see Table
\ref{table}). The part \( \omega_0 \) initializes the logical clock \( lc\_i \), as well as the
data structures \( lc\_ckpt[], idr\_ckpt[], greater[] \) and \( T[],[] \)
described in Sect. 3.2 and 3.3 (see lines 2-6, Table \ref{table}). In addition,
it takes the initial checkpoint at a process \( p_i \). In part \( \omega_1 \), when a message \( m \) is sent to a process \( p_j \), the boolean array
\text{send}\_to[i] is updated, the set \( \psi \) is constructed (see lines
9-19, Table \ref{table}) and included in \( m \). In part \( \omega_2 \), the reception
of messages is managed. \( \omega_2 \) updates the data structures ac-
cording to the piggybacked IDR information (see lines
27-52, Table \ref{table}). Finally, in \( \omega_2 \), the forced checkpoint condi-
tion \( \mathcal{S} \) is evaluated to determine if \( p_i \) should take a forced
checkpoint (see lines 22-24, Table \ref{table}).

The overhead per message of S-FI is determined by the
amount of tuples in \( \psi \) (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,
\( 1/(n-1) \sum_{i=1}^{n-1} (t)(2s+2) \) 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 struc-
tures \( (n(s+2)) \) bits. In Table \ref{table}, we show the results of this
brief analysis and the overhead messages for the algorithms
FI and FINE.
Table 1  S-FI algorithm.

(ω₁) Initialization of process pᵢ,
1  k, l : 1 . . . n, where n is the number of processes.
2  ∀ k do lc.ckpt[k] := 0; enddo
3  ∀ k, l do T[k, l] := true; enddo
4  idr.ckpt[k] := true;
5  greater[k] := false;
6  lcᵢ := 0;
7  taken.checkpoint();
8  when pᵢ sends a message m to pⱼ,
9     sent.J₀[m] := true;
10    ψᵢ := Ø;
11    ∀ k do
12       if [¬ T[k, j] ∨ idr.ckpt[k] ∧ (lc.ckpt[k] > 0)] then
13          ψᵢ ← ψᵢ ∪ (k, lc.ckpt[k], idr.ckpt[k], greater[k]);
14       endif
15    enddo
16    s := 32; s is the #bits to represent a logical clock(lc.ckpt).
17    size(ψᵢ) returns the cardinality of ψᵢ.
18    if size(ψᵢ) > (n)(s + 2)/(2s + 2) then
19       ψᵢ := Ø;
20    ∀ k do ψᵢ ← ψᵢ ∪ (−, lc.ckpt[k], idr.ckpt[k], greater[k]); enddo
21    endif
22    enddo
23    when pᵢ receives the message m := (ψ, Data) from pⱼ,
24     maxlc.ckpt := max(ψ);
25     if [(∃k : sent.J₀[m][k] ∧ (3y ∈ ψ, y.id = k : y:greater ∨
26       (3y ∈ ψ, y.id = k) ∧ max lc.ckpt > lcᵢ)] then
27     foreach i do
28         if (max lc.ckpt ≠ w.lc.ckpt) ∨ (lcᵢ > w.lc.ckpt) then
29            Tₜ[w, idₜ] := true;
30        endif
31        enddo
32        for all i do
33            w.lc.ckpt = cl.ckpt[w, idₜ] →
34            idr.ckpt[w, idₜ] := (idr.ckpt[w, idₜ] ∧ w.idₜ); enddo
35        enddo
36        max lc.ckpt < w.lc.ckpt → skip
dophone case
37        endcase
38     endfor
39     enddo
40     case
41        max lc.ckpt > lcᵢ →
42        lcᵢ := max lc.ckpt;
43        ∀ k ≠ i do greater[k] := true; enddo
44        ∀ ω ∈ ψ, i ≠ idₜ ∧ i do greater[i] := ω:greater; enddo
45        if (max lc.ckpt ≠ w.lc.ckpt) ∨ (lcᵢ > w.lc.ckpt) then
46            Tₜ[w, idₜ] := true;
47        endif
48        w.lc.ckpt = cl.ckpt[w, idₜ] →
49            idr.ckpt[w, idₜ] := (idr.ckpt[w, idₜ] ∧ w.idₜ); enddo
50        enddo
51        max lc.ckpt < lcᵢ → skip
dophone case
52        endcase
53    delivery(m);

Procedure and functions used in S-FI
1  When pᵢ takes a local or forced checkpoint.
2  procedure taken.checkpoint()
3  ∀ k do sent.J₀[k] := false; enddo
4  ∀ k ≠ i do
5     idr.ckpt[k] := false;
6     greater[k] := true;
7     Tₜ[k, i] := false;
8     enddo
9  lcᵢ := lcᵢ + 1;
10  lc.ckpt[i] := lcᵢ;
11  endprocedure
12  ∀ max(α) gets the maximum logical clock in α.
13  function max(α)
14      max := 0;
15      ∀ x ∈ α do
16         if x.lc.ckpt > max then max := x.lc.ckpt; endif
17      enddo
18  endfunction

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

| Algorithm | Best-Case | Average-Case | Worst-Case |
|-----------|-----------|--------------|------------|
| S-FI      | (n)(s + 2) | (n)(s + 2)   | s          |
| FI        | (n)(s + 2) | (n)(s + 2)   | n          |

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 5000 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. Internatinoal 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. Drida, “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. Lampot, “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 $\mathcal{D}$ is equivalent to the condition $\mathcal{C}^2$.

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 $FI_s$ is equivalent to $SFI_s$. 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 $lc_{i_{SFI}}^x$ and $lc_{i_{FI}}^x$ of a process $pi$ for $FI_b$ and $SFI_b$, respectively, are equal. In other words:

$$V_i \in P: lc_{i_{SFI}}^x = lc_{i_{FI}}^x$$

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 $lc_{i_{SFI}}^x$ and $lc_{i_{FI}}^x$, we have two cases: The first case is when process $pi$ takes a checkpoint and updates its $lc_i$. The second case is when $pi$ 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:

$$lc_{i_{SFI}}^2 := lc_{i_{SFI}}^1 + 1, \quad lc_{i_{FI}}^2 := lc_{i_{FI}}^1 + 1, \quad lc_{ckpt[i]} := lc_{ckpt[i]}^2 = 2$$

Therefore, $lc_{i_{SFI}}^2 = lc_{i_{FI}}^2 = 2$.

- **$p_i$ receives a message $m$ from $pj$ immediately after of its first checkpoint and $m.lc = 2$**. In this case, in $FI$, $p_i$ updates $lc_{i_{FI}}^2$ in the following way:

$$\text{if } m.lc_{i_{FI}} > lc_{i_{FI}}^1 \text{ then } lc_{i_{FI}}^2 := m.lc_{i_{FI}}$$

Therefore, $lc_{i_{SFI}}^2$ is updated with the greatest logical clock seen by $p_i$ and $p_j$.

In S-FI, $lc_{i_{FI}}^2$ 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 $m$. $lc_{i_{SFI}}^2$ and $lc_{ckpt[]}$ are updated as follows:

$$\text{if } \max(m.lc_{ckpt[]}) > lc_{i_{SFI}}^1 \text{ then } \quad lc_{i_{SFI}}^2 = \max(m.lc_{ckpt[]})$$

$$\forall l \neq i: \text{if } m.lc_{ckpt[l]} > lc_{ckpt[l]} \text{ then } \quad lc_{ckpt[l]} := m.lc_{ckpt[l]}$$

Therefore, $lc_{i_{SFI}}^2 = lc_{i_{SFI}}^1 = 2$, 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:

$$lc_{i_{SFI}}^k = lc_{i_{FI}}^k$$

- **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 $pi$ takes a checkpoint and updates its logical clock. The second case is when a $pi$ receives a message $m$ and updates its logical clock according to the piggybacked information included in $m$.

- **$p_i$ takes a checkpoint.** Therefore, $pi$ updates its logical clock in the following way:

$$lc_{i_{SFI}}^{k+1} := lc_{i_{SFI}}^k + 1, \quad lc_{i_{FI}}^{k+1} := lc_{i_{FI}}^k + 1, \quad lc_{ckpt[i]}^{k+1} := lc_{ckpt[i]}^k + 1$$

Therefore, $lc_{i_{SFI}}^{k+1} = lc_{i_{SFI}}^{k+1}$. 

IEICE TRANS. INF. & SYST., VOL.E96-D, NO.4 APRIL 2013
- \( p_i \) receives a message \( m \) from \( p_j \). We note that in the algorithm \( F\ell \), the \( lc_{i(j)} \) \((j \neq i)\) included in a message \( m \) \((m.lc_{i(j)})\) corresponds to the greatest clock seen by \( p_j \).

In this case, \( lc_{i(j)}^{+1} \) is updated in the following way:

\[
\text{if } m.lc_{i(j)} > lc_{i(j)} \text{, then } lc_{i(j)}^{+1} := m.lc_{i(j)}
\]

Therefore, \( lc_{i(j)}^{+1} \) is updated with the greatest logical clock seen by \( p_i \) and \( p_j \).

In the S-Fi algorithm, \( lc_{i(j)}^{+1} \) is also the greatest logical clock seen by \( p_j \), but in this case, it is included in the vector \( le.ckpt[i] \) of \( m \), \((m.lc_{i(j)} \in m.le.ckpt[])\). The logical clock \( lc_{i(j)}^{+1} \), and the vector \( le.ckpt[i] \) are updated in the following way:

\[
\text{if } \max(m.le.ckpt[]) > lc_{i(j)} \text{, then } lc_{i(j)}^{+1} := \max(m.le.ckpt[])
\]

\[
\forall l \neq i : \text{if } m.le.ckpt[l] > lc\text{.ckpt}[l] \text{, then } lc\text{.ckpt}[l] := m.le.ckpt[l]
\]

Therefore, \( lc_{i(j)}^{+1} \) is also updated with the greatest \( lc_{i(j)} \) seen by \( p_i \) and \( p_j \), while the vector \( le.ckpt[i] \) is updated also with the greatest \( lc_{i(j)} \).

Therefore, \( lc_{i(j)}^{+1} = lc_{i(j)}^{+1} \). □Lemma 1

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

\[
le.ckpt[i] = (lc_{i(j)}^{+1} = \max(le.ckpt[]))
\]

Now using the Lemma 1 and the Proposition 1, we can state that:

\[
mlc > lc_j \equiv \max(mlc, ckpt) > lc_j
\]

Therefore, \( Fl_c = SFI_c \). □

Part II. Now we will prove that \( Fl_c \) is equivalent to \( SFI_c \). We divide this proof into two parts. In the first part, we prove that \( le.ckpt[i] \) has a similar behavior as \( ckpt[i] \) 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 \( F\ell \) algorithm.

- Part II.A. Since the logical clock \( ckpt[i] \) has a strictly increasing behavior, we prove that the logical clock \( le.ckpt[i] \) has the same property.

Lemma 2. The logical clock \( le.ckpt[i] \) of process \( p_i \) has a strictly increasing behavior, as follows:

\[
\forall i \in P : le.ckpt[i] \leq \cdots \leq le.ckpt[i]^{+1} \leq le.ckpt[i]^{+2},
\]

where \( x \) represents the \( x\)-th taken checkpoint of \( p_i \).

Proof Lemma 2. This part is demonstrated by direct proof. We note that \( lc_{i(j)} \) has a strictly increasing behavior [5]. From Lemma 1 we have that \( lc_{i(j)} = lc_{i(j)}^{+1} \), therefore the logical clock \( lc_{i(j)}^{+1} \) also has the same property. Since \( le.ckpt[i] \) is set to \( lc_{i(j)}^{+1} \), for each taken checkpoint at process \( p_i \), we have that the logical clock \( le.ckpt[i] \) has a strictly increasing behavior. □Lemma 2

Now using Lemma 2 and knowing that the logical clock \( le.ckpt[i] \) is only updated when \( p_i \) takes a local checkpoint, we can state that the logical clock \( le.ckpt[i] \) is constant during an interval.

- Part II.B. In \( F\ell \) algorithm \( taken[i] = \text{true} \) indicates that there is a causal zigzag path, including a checkpoint, from the last checkpoint \( C_j^i \), known by \( p_i \), to the next checkpoint \( C_j^{i+1} \). Specifically we are interested when \( taken[i] = \text{true} \) and \( j = i \) since there is a causal zigzag path that includes a checkpoint \( C_j^i \) in the interval defined by the checkpoints \( C_i^j \) and \( C_j^{i+1} \). 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 \( p_i \), \( i \neq j \)

\[
\text{if } m.idr.ckpt[i] = \text{false} \text{ then }\]

\[
\exists C_i \in R, k \neq i : C_i^j \rightarrow C_i^k \rightarrow C^j_i
\]

For Lemma 3, we give a sketch of proof. According to definition 3, we have that two checkpoints \( C_j^i \) and \( C_j^{i+1} \) are IDR related if \( \exists C_i \) : \( C_j^i \rightarrow C_i^k \rightarrow C_j^{i+1} \). During the message exchange between \( C_j^i \) and \( C_j^{i+1} \), in S-Fi the value \( idr.ckpt[i] = \text{true} \) is propagated between each pair of consecutive messages iff a checkpoint \( C_i \) 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 for message reception for \( idr.ckpt[i] \), page 889). Otherwise, when a local checkpoint is taken, the IDR history of \( p_i \) with respect to \( p_i \) is erased by reinitializing \( idr.ckpt[i] = \text{false} \) (see the updating process for \( idr.ckpt[i] \), page 889).

Therefore \( Fl_c \equiv SFI_c \). □

Appendix B

Theorem 3. Condition \( D' \) is equivalent to the condition \( D \).

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 predecessors without requiring to piggyback the whole control information in each message. Secondly, we demonstrate that \( SFI_c \land SFI_b \) is equivalent to \( SFI' \land SFI'_b \) 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 \( K3(m, k) \equiv (send(m).le.ckpt[k] = 0) \lor ((send(m).T[i][j, k] = 1) \land (send(m).idr.ckpt[k] = 1)) \). We have: \( K3(m, k) \Rightarrow K(m, k) \).

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

\[
K(m, k) \equiv (send(m).VC[i][k] = 0) \lor (send(m).VC[i][k] < pred(receive(m)).VC[i][k])\land (send(m).IP[i][k] = 1))
\]
here, $VC_i[]$ is a vector of logical clocks and $IP_i[]$ is a boolean array.

Let's 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. Specifically, it was demonstrated that the logical clocks of the vector $lc_ckpt_i[]$ 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. $(send(m), T_i[j, k] = 1) \Rightarrow ((send(m), lc_ckpt[k] ≤ pred(receive(m)), lc_ckpt[j][k]) ∧ (max(send(m), lc_ckpt[i][k]) > send(m), lc_ckpt[k][k]));$  

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

$$(e, T_i[j, k] = 1) \Rightarrow \neg \forall e' \in \{e | e' \rightarrow e\} \forall j, \forall k : \mathcal{I}(e', j, k) \text{ holds.}$$

$$(e, T_i[j, k] = 1) \Rightarrow \neg \forall e' \in \{e | e' \rightarrow e\} \forall j, \forall k : \mathcal{I}(e', j, k) \text{ holds.}$$

**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 $\mathcal{I}(e, j, k)$ 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 $max(e.lc_ckpt_i[]) = (e.lc_ckpt_i[]) = 1$ and:

    - $j = i, \forall k : (T_i[j, k] = 1) \Rightarrow A0.$
    - $\forall j \neq i, \forall k : (T_i[j, k] = 1) \Rightarrow A2 \land B0.$
    - $\forall j \neq i, k = i : (T_i[j, k] = 0).$

  - $e$ is the reception of a message $m$ from $p_j$ immediately after the first checkpoint of $p_i$. Then, from T2 and the management of $lc_ckpt_i[]$, we have:

    - $j = i, \forall k : (T_i[j, k] = 1) \Rightarrow \neg A0.$
    - $\forall x \neq i, \forall y, \forall z \in m.\psi, k = z.id : (e.T_i[x, y] = 1) \Rightarrow [(x = j) \land (y = k) \land (max(e.lc_ckpt_i[]) > z.lc_ckpt_i[k]) \lor \{
        (y \neq i) \land (y \neq k) \land (lc_ckpt_i[k] = 0) \}].$

First alternative holds. $m$ satisfies A3 and B0, $receive(m) = e \land send(m), lc_ckpt_i[k] = z.lc_ckpt_i[k] = max(e.lc_ckpt_i[])$.

Second alternative also holds. $e.lc_ckpt_i[k] = 0$ satisfies A2 and B0.

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

- **Inductive step:** let $e \in H_i$. We assume that $\forall e' \in \{e | e' \rightarrow e\}, \forall j, \forall k : \mathcal{I}(e', j, k) \text{ holds.}$

- **Inductive hypothesis:** we will prove that $\forall j, \forall k$, the property $\mathcal{I}(e, j, k)$ 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, $\mathcal{I}(e, j, k)$ holds.

Now, let $m$ be a message sent by $p_j$ to $p_i (e = send(m))$ 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:

  $$send(m).lc_ckpt_i[k] ≤ pred(receive(m)).lc_ckpt_i[k].$$

- From A2, $send(m).lc_ckpt_i[k] = 0$. Then, $send(m).lc_ckpt_i[k] ≤ pred(receive(m)).lc_ckpt_i[k].$

- From A3, we have: $send(m).lc_ckpt_i[k] = e.lc_ckpt_i[k] \leq pred(receive(m)).lc_ckpt_i[k]$. Hence, $send(m).lc_ckpt_i[k] ≤ pred(receive(m)).lc_ckpt_i[k].$

Therefore, $\forall e, e'.T_i[j, k] = 1 \Rightarrow (send(m).lc_ckpt_i[k] ≤ pred(receive(m)).lc_ckpt_i[k]).$

Hence, we have:

$K3(m, k) \equiv (send(m).lc_ckpt_i[k] = 0) \land (send(m).T_i[j, k] = 1) \land (send(m), idr_ckpt_i[k] = 1)) \Rightarrow (send(m).lc_ckpt_i[k] = 0) \land (send(m), idr_ckpt_i[k] = 1))$ 

$\Rightarrow (send(m).lc_ckpt_i[k] = 0) \land (send(m), idr_ckpt_i[k] = 1)) \Rightarrow (send(m).lc_ckpt_i[k] = 0) \land (send(m), idr_ckpt_i[k] = 1)).$

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:

$SFI_a \wedge SFI_b \Rightarrow SFI_a' \wedge SFI_b' \text{; where:}$

$SFI_a \equiv (\exists k : send_{fo}[k] \land m.greater[k])$

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

$SFI_a' \equiv (\exists k : send_{fo}[k] \land (3y \in m.\psi, y.id = k : y.greater) \lor (3y \in m.\psi, y.id = k))$

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

We note that the value of $send_{fo}[k]$ is equal for both $\mathcal{D}$ and $\mathcal{D}'$ (see 3.2 and 3.3). $max(m.lc_ckpt_i[]) = max(lc_ckpt_i[k])$ (see 3.2 and Lemma 4). In addition, $max(lc_ckpt_i[k])$ is always included in $m.\psi$ (see Lemma 4). Now, let $m$ sent by $p_j$ to $p_i$ and $send_{fo}[k] = true$. We have two cases to analyze:
Finally, in order to prove that Part II.b.

\[ SFI_a \land SFI_b \Rightarrow (SFI'_1 \land SFI'_2) \]

holds.

\[ \exists y \in m.\psi, y.id = k \]  In this case, \( (SFI_a \land SFI_b) \Rightarrow (SFI'_1 \land SFI'_2) \) holds.

\[ \exists y \in m.\psi, y.id = k \]  In this case, from Theorem 4, we also have two cases:

- \( send(m).lc\_ckpt[i][k] = 0 \). From management of \( greater_j[l] \) (see, Sect. 3.1) we have: \( lc_j \geq send(m).lc\_ckpt[i][j] \geq 1 > send(m).lc\_ckpt[i][k] = 0 \) \( \land lc_i \); thus, \( greater_j[k] \) is true. Therefore, \( (SFI_a \land SFI_b) \Rightarrow (SFI'_1 \land SFI'_2) \) holds.

- \( (send(m).T_j[i][k] = 1) \land (send(m).idr\_ckpt[i][k] = 1) \). Let \( e = send(m) \), from Lemma 4, we have (we note that \( j \neq i \) and \( e \) is not a receive event):

  - From A1, \( k = i \). Thus, \( send(m).lc\_ckpt[i][k] \leq pred(receive(m)).lc\_ckpt[i][k], max(e.lc\_ckpt[i][j]) > e.lc\_ckpt[i][k] \) and \( (send(m).idr\_ckpt[i][k] = 1) \). Let \( e.lc\_ckpt[i][s] = max(e.lc\_ckpt[i][j]) = lc_j \). Then exists a sequence of causal messages \( [m_1 \downarrow m_2 \downarrow \ldots \downarrow m_t] \) 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 \( p_j \). In this case, \( lc_j = lc_i = max(e.lc\_ckpt[i][j]) \). Thus, \( SFI_b = SFI'_1 = false \).
    * There is not a sequence of messages from \( p_s \) to \( p_j \). Thus, \( lc_j > lc_i \), then \( greater_j[k] = true \).

Therefore, \( SFI_a \land SFI_b \Rightarrow SFI'_1 \land SFI'_2 \) holds. From A2, \( send(m).lc\_ckpt[i][k] = 0 \). It was analyzed previously.

- From A3, \( \exists m' \) from \( p_i \) to \( p_j \). \( \forall z \in m'.\psi, \ldots \)

  Let \( e.lc\_ckpt[i][s] = max(e.lc\_ckpt[i][j]) \). Then 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, \( lc_i = lc_j = max(e.lc\_ckpt[i][j]) \). Therefore, \( SFI_b = SFI'_1 = false \).
    * There is not sequence of causal messages from \( p_s \) to \( p_j \). Thus, \( lc_j > lc_k \), therefore \( greater_j[k] = true \).

Thus, in all the cases \( (SFI_a \land SFI_b) \Rightarrow (SFI'_1 \land SFI'_2) \) holds.

Part II.b. Finally, in order to prove that \( SFI'_1 \) is equivalent to \( SFI_c \), we demonstrate by direct proof that: \( SFI_c \Rightarrow SFI'_2 \); where:

\[
SFI_c = \begin{cases} \exists z \in m.\psi, z.id = i : lc\_ckpt[i] = z.lc\_ckpt \land \neg m.idr\_ckpt[i] \\
SFI'_c = (\exists z \in m.\psi, z.id = i : lc\_ckpt[i] = z.lc\_ckpt \land \neg z.idr\_ckpt) 
\end{cases}
\]

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

- \( \exists z \in m.\psi, z.id = i \). In this case \( SFI_c \Rightarrow SFI'_1 \) holds.
- \( \neg \exists z \in m.\psi, z.id = i \). Thus, \( SFI'_c \) is always false for this case. Let \( e = send(m) \), from Theorem 4 we have two cases:

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

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).