Optimal Asynchronous Garbage Collection for Checkpointing Protocols with Rollback-Dependency Trackability

Rodrigo Schmidt* Islene C. Garcia† Fernando Pedone* Luiz E. Buzato†

*École Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland
Phone: +41 21 693 4797 Fax: +41 21 693 6600
E-mail: {rodrigo.schmidt, fernando.pedone}@epfl.ch

†Instituto de Computação – Unicamp, 13083-970 Campinas – SP, Brazil
Phone: +55 19 788 5876 Fax: +55 19 788 5840
E-mail: {islene, buzato}@ic.unicamp.br

EPFL Technical Report IC/2004/45
14 May 2004

Abstract

Communication-induced checkpointing protocols that ensure rollback-dependency trackability (RDT) guarantee important properties to the recovery system without explicit coordination. However, to the best of our knowledge, there was no garbage collection algorithm for them which did not use some type of process synchronization, like time assumptions or reliable control message exchanges. This paper addresses the problem of garbage collection for RDT checkpointing protocols and presents an optimal solution for the case where coordination is done only by means of timestamps piggybacked in application messages. Our algorithm uses the same timestamps as off-the-shelf RDT protocols and ensures the tight upper bound on the number of uncollected checkpoints for each process during all the execution.

Keywords: garbage collection, distributed checkpointing, rollback-dependency trackability, rollback-recovery, fault tolerance.

1 Introduction

Motivation. Checkpointing is a well-known technique used to build fault-tolerant distributed applications based on rollback-recovery. Briefly, every process periodically saves the application's local state as a *checkpoint* and when a failure occurs, the distributed computation restarts from its most recent *consistent global checkpoint*, or *recovery line*. A global checkpoint is a set composed of one local checkpoint for each process and it is consistent if it includes the sending of every received message [6, 8].

Netzer and Xu [16] have shown that checkpoint dependencies are created by sequences of messages called *zigzag paths*. Two checkpoints can take part in the same consistent global checkpoint if and only if no zigzag path connects them. A zigzag path can be either causal or non-causal depending on whether the receipt of a message always precedes the sending of the next one. Non-causal zigzag paths may connect a checkpoint to itself and preclude it from taking part in any consistent global checkpoint. A checkpoint involved in such a *zigzag cycle* is called *useless*.

If checkpoints are taken autonomously by processes (called *basic checkpoints*) and no coordination exists, they may become useless and a failure could force the application to roll back to a very initial state, a phenomenon known as the *domino effect* [17]. *Communication-induced checkpointing protocols* [8, 15, 20] avoid the domino effect by piggybacking control information in the application messages and having processes take *forced checkpoints*, besides the basic ones, to break the non-causal zigzag paths that could create useless checkpoints.

Absence of useless checkpoints is the minimal desired property for communication-induced protocols. Another important property is the possibility of tracking checkpoint dependencies on-the-fly during the application execution using a transitive dependency vector, called *rollback-dependency trackability* (RDT). Besides ensuring that all checkpoints are useful, the RDT property eases the determination of minimum and maximum consistent global checkpoints containing a given set of local checkpoints, and allows decentralized solutions for recovery line calculation, which has been shown to be helpful in many contexts (e.g., software error recovery, causal distributed breakpoints, deadlock recovery and mobile computing [20]). Moreover, the RDT property minimizes the amount of lost work in a distributed rollback when compared to other domino-free properties [1]. Protocols enforcing the rollback-dependency trackability are called *RDT checkpointing protocols* [3, 10, 19, 20].

The price of autonomy in communication-induced checkpointing protocols is storage space [2]. The absence of explicit coordination makes it difficult to identify *obsolete checkpoints*, that is, those not necessary for future recoveries. Existent garbage collection algorithms eliminate all or a subset of the obsolete checkpoints [5, 8, 14, 21]. However, all of them rely either on time assumptions or reliable control message exchanges.

This paper addresses the problem of garbage collection for RDT checkpointing protocols where coordination relies only on information propagated in application messages. We call such garbage collection algorithms *asynchronous*. In this context, we present an asynchronous garbage collection algorithm and prove its optimality. Our algorithm is optimal in the sense that no more checkpoints can be eliminated without time assumptions or control messages.

Related work. Rollback-dependency trackability was originally presented by Wang [20], who introduced efficient distributed algorithms for calculating minimum and maximum consistent global checkpoints containing a given set of local checkpoints, when the RDT property holds. This seminal work also discusses the application of these algorithms in different scenarios.

RDT can be also defined as the absence of non-causal dependencies, since non-causal zigzag paths must be doubled by a causal relation to ensure on-the-fly trackability [4, 15]. This observation provided a new perspective on RDT algorithms. Based on it, Agbaria *et al.* [1] showed that in case of failure, RDT ensures better bounds on the number of rolled back checkpoints than other domino-free properties.

Much research has been pursued in reducing the number of forced checkpoints in RDT checkpointing protocols. Baldoni *et al.* [3] and Garcia *et al.* [10] presented protocols that take fewer forced checkpoints than the protocols presented by Wang [20]. Important results in this context are related to the minimal visible characterization of the RDT property [4, 9], which gives the strongest condition to be tested for taking forced checkpoints in order to ensure RDT. However, Tsai *et al.* [19] showed that strong conditions not always translate into a fewer number of forced checkpoints.

Although garbage collection incurs overhead, being an important pragmatic issue in rollback-recovery, it has received little attention in the literature [8]. A simple approach based on the recovery line for the failure of all processes is presented in [5, 8]. Although simple, this algorithm requires process synchronization, blocking processes while reliable control messages are exchanged, and does not bound the number of uncollected checkpoints. Wang *et al.* [21] presented a general characterization of obsolete checkpoints and developed an algorithm that discards all of them and ensures a limit on the number of uncollected checkpoints. However, like the previous approach, it involves process blocking and reliable control messages. The strategy proposed by Manivannan *et al.* [14] does not exchange control messages, but requires processes to take basic checkpoints in precise and known time intervals, which is unfeasible in many practical scenarios.

Our results. This paper makes the following contributions. We present a characterization of obsolete checkpoints for RDT scenarios and a new algorithm for garbage collection. Differently from the previous approaches, ours does not rely on time assumptions or control messages. It runs locally to each process and is based only on the timestamps already propagated by the checkpointing protocol, increasing neither the amount of control information piggybacked nor the execution complexity of the checkpointing middleware. Moreover, we prove that no algorithm can eliminate more checkpoints based only on timestamps piggybacked in application messages.

The rest of this paper is organized as follows. Section 2 introduces our model and definitions. Section 3 describes the necessary and sufficient conditions for a checkpoint to be obsolete when RDT holds. Section 4 presents and analyzes our algorithm and Section 5 concludes the paper.

2 System model and definitions

A distributed system is composed of a set $\Pi = \{p_1, p_2, \ldots, p_n\}$ of processes that communicate only by exchanging messages. The system is asynchronous: there are no assumptions about the time it takes for processes to execute and for messages to be exchanged. Moreover, processes do not share a common clock. Although messages cannot be corrupted, they can be lost or delivered out of order. Process p_i's execution is a sequence of events $e_i^0, e_i^1, \ldots$ *Internal events* are related to the local execution of a process (e.g., local checkpoints) and *communication events* are related to sending and receiving messages.

A process can fail by crash, stopping its execution and losing its volatile state, but it eventually recovers. Its stable storage persists through failures, preserving the stored information. Finally, we do not assume piecewise determinism, and therefore cannot use event logging during recovery [8].

2.1 Causality and consistency

Throughout the paper we use the definitions of causal precedence and consistent cuts, presented next.

Definition 1 ***Causal precedence*** *[13] – Event* e_a^α *causally precedes* e_b^β $(e_a^\alpha \rightarrow e_b^\beta)$ *iff*
(i) $a = b \wedge \beta = \alpha + 1$*; or*
(ii) $\exists m \mid e_a^\alpha = send(m) \wedge e_b^\beta = receive(m)$*; or*
(iii) $\exists e_c^\gamma \mid e_a^\alpha \rightarrow e_c^\gamma \wedge e_c^\gamma \rightarrow e_b^\beta$.

A cut of a distributed computation contains an initial prefix of the sequence of executed events for each process. A consistent cut is left-closed under causal precedence and represents an instant in a distributed computation, as defined below.

Definition 2 ***Consistent cut*** *[7] – A cut $\mathcal{C}$ is consistent iff $e \in \mathcal{C} \ \wedge \ e' \rightarrow e \Rightarrow e' \in \mathcal{C}$.*

2.2 Checkpointing

A local checkpoint written on stable storage is a *stable checkpoint*. We use s_i^γ to represent the γ-th stable checkpoint taken by process p_i and call γ its index. Every process p_i starts its execution by storing a stable checkpoint s_i^0. This ensures the existence of at least one global recoverable state. The volatile state of a process p_i is called a *volatile checkpoint* and denoted by v_i. The set of all checkpoints taken by all the processes in a consistent cut and the dependency relation between them created by the exchanged messages (excluding lost and in-transit messages) form a *Checkpoint and Communication Pattern* (CCP). We use $last\_s(i)$ to refer to the index of the last stable checkpoint taken by process p_i in a given CCP and denote $s_i^{last\_s(i)}$ by s_i^{last} for simplicity. Moreover, we define c_i^γ as a *general checkpoint* (or simply checkpoint) of a CCP as follows:

$$c_i^\gamma = \begin{cases} s_i^\gamma, & \gamma \leq last\_s(i); \\ v_i, & \gamma = last\_s(i) + 1. \end{cases} \quad (1)$$

A *checkpoint interval* I_i^γ is the set of events occurred in process p_i between checkpoints $c_i^{\gamma-1}$ and c_i^γ (including $c_i^{\gamma-1}$ but not c_i^γ). Figure 1 gives an example of CCP and depicts selected examples of the previous definitions.

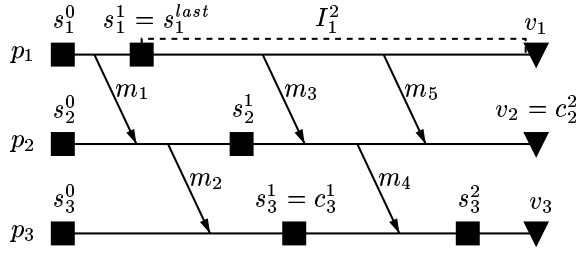

Figure 1: Example of CCP.

Two checkpoints are inconsistent if they are causally related and consistent otherwise. As a result, a global checkpoint is consistent if, and only if, all its checkpoints are pairwise consistent. In Figure 1, $\{v_1, s_2^1, s_3^1\}$ is consistent and $\{s_1^0, s_2^1, s_3^1\}$ is inconsistent. A consistent global checkpoint always represents a consistent cut.

Two consistent checkpoints are not necessarily part of the same consistent global checkpoint. Checkpoint dependencies are created by sequences of messages called *zigzag paths* [16]. We use the relation $c_a^\alpha \rightsquigarrow c_b^\beta$ to represent the existence of a zigzag path from c_a^α to c_b^β. A checkpoint c_i^γ is useless if, and only if, $c_i^\gamma \rightsquigarrow c_i^\gamma$ [16].

Definition 3 ***Zigzag path*** *[16] – A sequence of messages $\mu = [m_1, \ldots, m_k]$ is a zigzag path which connects c_a^α to c_b^β iff the conditions below hold:*
(i) p_a sends m_1 after c_a^α;
(ii) if $m_i, 1 \leq i < k$, is received by p_c, then m_{i+1} is sent by p_c in the same or a later checkpoint interval; and
(iii) p_b receives m_k before c_b^β.

A zigzag path can be causal (C-path) or not (Z-path). It is causal if the receipt of each message but the last one causally precedes the send event of the next one in the sequence. In Figure 1, $[m_1, m_2]$ and $[m_1, m_4]$ are examples of C-paths, and $[m_5, m_4]$ is an example of Z-path.

2.3 Rollback-dependency trackability

Rollback-dependency trackability is given by the absence of Z-paths which (a) connect a checkpoint to itself or (b) are not doubled by C-paths. This ensures that all checkpoint dependencies are causal and can be tracked by using transitive dependency vectors.

Definition 4 ***Rollback-dependency trackability*** *[15] – A CCP satisfies rollback-dependency trackability (is RD-trackable) iff for any two checkpoints c_i^γ and c_j^ι, $c_i^\gamma \rightsquigarrow c_j^\iota \Rightarrow c_i^\gamma \rightarrow c_j^\iota$.*

In RD-trackable checkpoint and communication patterns, there are no useless checkpoints, since $c_i^\gamma \rightsquigarrow c_i^\gamma$ implies $c_i^\gamma \rightarrow c_i^\gamma$, which is impossible. The CCP presented in Figure 1 is RD-trackable. It would not be in the absence of message m_3 because $[m_5, m_4]$ is a Z-path from s_1^1 to s_3^2. Therefore, without m_3 we would have $s_1^1 \rightsquigarrow s_3^2$ and $s_1^1 \not\rightarrow s_3^2$.

RDT checkpointing protocols rely on the model we presented and ensure that the CCP of any consistent cut of the distributed computation is RD-trackable [9]. Therefore, henceforth we assume that all the checkpoint and communication patterns are RD-trackable and we omit this condition in statements of definitions, lemmas and theorems.

2.4 Rollback-recovery

The system execution alternates between normal execution periods and recovery sessions, started after some failure. There are many possible approaches to orchestrate recovery sessions [11, 12, 14]. We do not address this problem in this paper and simply assume the existence of a recovery manager which stops the execution of non-faulty processes, takes their volatile state, calculates and propagates the recovery line, defined below.

Definition 5 ***Recovery line*** *[21] – Given a CCP and a set of faulty processes $F \subseteq \Pi$, the recovery line R_F is the consistent global checkpoint which does not include a volatile checkpoint of a faulty process and minimizes the number of general checkpoints rolled back.*

3 Characterization of obsolete checkpoints

As execution progresses, new checkpoints are taken and new recovery lines are formed for the possible sets of faulty processes. This makes some stable checkpoints obsolete, allowing the application to discard them in order to save stable storage space.

Definition 6 ***Obsolete checkpoint*** *– A stable checkpoint is obsolete iff it cannot take part in any future recovery line, even after rollbacks.*

Definition 6 is based on the future execution of the distributed application and cannot be used to identify all the obsolete checkpoints in a given CCP. We need a practical characterization of obsolete checkpoints and our starting point is recovery line determination. It is known that the recovery line of a faulty set F is unique [21]. The following lemma characterizes it for RD-trackable CCPs. (Due to space limitations, all lemma proofs are presented in Appendix A.)

Lemma 1 *Given a CCP and a set F of faulty processes, the recovery line R_F is determined by:*

$$R_F = \bigcup_{i=0}^{n-1} \{c_i^{max(k)} \mid \forall p_f \in F, s_f^{last} \not\rightarrow c_i^k\}.$$

Informally, the recovery line is composed of the last checkpoint of each process, volatile or not, which is not causally preceded by the last stable checkpoint of any faulty process. Figure 2 gives an example of recovery line determination in a CCP for $F = \{p_2, p_3\}$. The gray checkpoints are causally preceded by s_2^{last} or s_3^{last}. Thus, by Lemma 1, the recovery line is composed of the last black checkpoint of each process. Notice that s_3^{last} is not part of the recovery line because it is causally preceded by s_2^{last}.

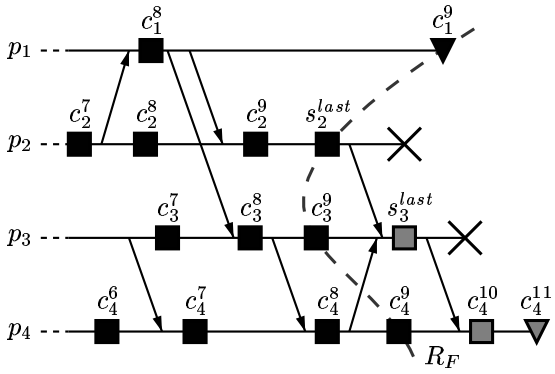

Figure 2: Recovery line determination.

A necessary condition for a checkpoint to be obsolete in a CCP defined by a consistent cut is that it not take part in any of the recovery lines for the 2^n possible sets of faulty processes (subsets of Π). A checkpoint which does not satisfy this condition in a consistent cut $\mathcal{C}$ is called *$\mathcal{C}$-needless.*

Definition 7 ***Needlessness*** *– A stable checkpoint s_i^γ is needless in a consistent cut $\mathcal{C}$ (is $\mathcal{C}$-needless) iff*

$$s_i^\gamma \in \mathcal{C} \ \wedge \ \forall F \subseteq \Pi : s_i^\gamma \notin R_F.$$

Lemmas 2 and 3 describe, respectively, an easier way to identify needless checkpoints in RD-trackable CCPs and the complete relation between needless and obsolete checkpoints. Similar lemmas have been presented in [21] under different assumptions.

Lemma 2 *Every stable checkpoint s_i^γ, part of the recovery line for a set of faulty processes F in a CCP, is also part of the recovery line for a single faulty process p_f in the same CCP, that is,*

$$s_i^\gamma \in R_F \Rightarrow \exists p_f \in \Pi \mid s_i^\gamma \in R_{\{p_f\}}.$$

Lemma 3 *A stable checkpoint s_i^γ is obsolete in the CCP defined by a consistent cut $\mathcal{C}$ iff it is $\mathcal{C}$-needless.*

Now we have means to characterize obsolete checkpoints in RD-trackable CCPs using a condition that does not need future knowledge, as we present in Theorem 1.

Theorem 1 ***Characterization of obsolete checkpoints*** *– A stable checkpoint s_i^γ is obsolete iff there is no process p_f such that*

$$s_f^{last} \rightarrow c_i^{\gamma+1} \wedge s_f^{last} \not\rightarrow s_i^\gamma.$$

Proof: By Lemmas 1, 2, 3 and Definition 7. □

Theorem 1 says that a process p_i must retain the most recent stable checkpoint which is not causally preceded by s_f^{last} for every process $p_f \in \Pi$ such that $s_f^{last} \rightarrow v_i$. All the other checkpoints of p_i are obsolete and may be eliminated. Clearly, the checkpoint s_i^{last} of every process p_i is not obsolete because $s_i^{last} \rightarrow v_i \wedge s_i^{last} \not\rightarrow s_i^{last}$. In Figure 2, for example, there are exactly five obsolete checkpoints: $\{c_2^7, c_2^9, c_3^8, c_4^6, c_4^8\}$.

4 Asynchronous garbage collection

Theorem 1 can be used to identify all the existing obsolete checkpoints with a simple algorithm like the one presented by Wang *et al.* [21]. However, this type of algorithm is based on reliable control messages and forces processes to block during their execution. Ideally, garbage collection should be as little intrusive as possible, not introducing any overhead in the normal computation or blocking the execution of processes. We capture this intuition with the notion of asynchronous garbage collection algorithms, as described next. In this section we also provide such an algorithm and prove its correctness and optimality.

Definition 8 *A garbage collection algorithm is asynchronous iff it relies only on information piggybacked in the existent application messages.*

4.1 Sufficient condition for garbage collection

We develop next a sufficient condition for asynchronous garbage collection based on causal knowledge only. Let $last\_k_i(j)$ denote the index of the last stable checkpoint of process p_j known by process p_i, that is, the last checkpoint of p_j which causally precedes the current volatile state of p_i. If no such stable checkpoint exists, let $last\_k_i(j) = -1$. For simplicity, we denote $s_j^{last\_k_i(j)}$ by $s_j^{lastk_i}$. Using this terminology, we show in Theorem 2 how to weaken Theorem 1 to get a sufficient condition for garbage collection in RD-trackable CCPs based on causal knowledge.

Theorem 2 *A stable checkpoint s_i^γ is obsolete if there is no process p_f such that*

$$last\_k_i(f) \geq 0 \ \wedge \ s_f^{lastk_i} \rightarrow c_i^{\gamma+1} \ \wedge \ s_f^{lastk_i} \not\rightarrow s_i^\gamma.$$

Proof: Suppose, by contradiction, that s_i^γ satisfies this condition and is not obsolete. By Theorem 1, there is a process p_f such that $s_f^{last} \rightarrow c_i^{\gamma+1} \wedge s_f^{last} \not\rightarrow s_i^\gamma$. As $s_f^{last} \rightarrow c_i^{\gamma+1}$, p_i knows s_f^{last} and $last\_k_i(f) = last\_s(f)$. Therefore, $last\_k_i(f) \geq 0 \ \wedge \ s_f^{lastk_i} \rightarrow c_i^{\gamma+1} \ \wedge \ s_f^{lastk_i} \not\rightarrow s_i^\gamma$, contradicting our initial assumption. □

Based on this condition, a process p_i could safely retain only its last stable checkpoint that is not causally preceded by $s_f^{lastk_i}$ for every process p_f such that $last\_k_i(f) \geq 0$, being sure that all non-obsolete checkpoints are preserved.

4.2 Dependency vectors

To implement the condition stated in Theorem 2, we need a dependency tracking mechanism. *Dependency vectors* [18] capture causal dependencies among checkpoints and are commonly used in RDT checkpointing protocols [3, 10, 20]. In this mechanism, each process p_i maintains and propagates inside application messages a size-n dependency vector DV, initially $(0, \ldots, 0)$. Entry $DV[i]$ represents the current checkpoint interval of p_i and is incremented immediately after a new checkpoint is taken. Every other entry $DV[j]$, $j \neq i$, represents the highest interval index of p_j upon which p_i depends and is updated every time a message m with a greater value of $m.DV[j]$ arrives at p_i. When a stable checkpoint is taken, the current dependency vector is stored with it for recovery purposes. We use $DV(c_i^\gamma)$ to refer to the dependency vector of checkpoint c_i^γ. The following equation derive from the propagation mechanism of dependency vectors [18]:

$$c_a^\alpha \rightarrow c_b^\beta \iff \alpha < DV(c_b^\beta)[a]. \qquad (2)$$

Moreover, as $DV(v_i)[j]$ represents the most recent checkpoint interval from p_j known by p_i, we have that

$$last\_k_i(j) = DV(v_i)[j] - 1. \qquad (3)$$

Based on it, Corollary 1 restates Theorem 2 in terms of dependency vectors.

Corollary 1 *A stable checkpoint s_i^γ is obsolete if there is no process p_f such that*

$$DV(v_i)[f] = DV(c_i^{\gamma+1})[f] \wedge DV(v_i)[f] > DV(s_i^\gamma)[f].$$

Proof: By Theorem 2 and Equations 2 and 3.□

Notice that Corollary 1 relies only on values of DV local to process p_i and allows it to eliminate obsolete checkpoints without exchanging information with other processes. In the next section we present our complete garbage collection algorithm.

4.3 Algorithm description

Our algorithm, named RDT-LGC, simply implements the idea of Theorem 2, identifying obsolete checkpoints as soon as they satisfy the condition of Corollary 1. We assume that the CCPs created during the execution of the distributed application are always RD-trackable. In Section 4.5 we show how checkpointing and garbage collection could be merged in a single algorithm.

Theorem 2 states that a process p_i can retain, for every process p_f, only the most recent checkpoint not causally preceded by $s_f^{lastk_i}$. Therefore, p_i can maintain a simple size-n vector UC (Uncollected Checkpoints) that maps p_f to the checkpoint retained because of p_f. Notice, however, that more than one process can break the condition of Theorem 2 for the same checkpoint of p_i. Thus, we use a different structure called CCB (Checkpoint Control Block) to represent an uncollected stable checkpoint of p_i. A CCB keeps track of the checkpoint index and a reference counter storing how many processes deny the checkpoint elimination. UC entries reference CCBs to simplify their update when new causal information is received.

Algorithm 1 presents these data structures, together with the dependency vector, and the basic procedures to manipulate them. Every process has its own instances of the presented data structures. Procedure **release** decrements the reference counter of the referenced CCB and, if there is no other reference, collects the obsolete checkpoint. Procedure **link** makes $UC[j]$ reference the same CCB of $UC[i]$. Procedure **newCCB** creates a new CCB and makes $UC[j]$ reference it. In the following, we explain the RDT-LGC algorithm during normal execution periods and recovery sessions separately.

Normal execution periods. In these periods, RDT-LGC simply updates the data structures mentioned above in order to identify obsolete checkpoints as soon as they satisfy the condition presented in Corollary 1, as shown in Algorithm 2. When a message is received by p_i and a new causal dependency from process p_j is noticed (line 2), p_i must keep track that now, by

Algorithm 1 Data structures of RDT-LGC

Data structures

```
1: Type
2:    CCB: record of          {checkpoint control block}
3:              IND: integer          {checkpoint index}
4:              RC: integer           {reference counter}
5: Var
6:    UC : array[1 .. n] of ↑CCB
7:    DV : array[1 .. n] of integer
```

Procedure initialize()

```
1: for j ← 1 to n do
2:    UC[j] ← Null
3:    DV[j] ← 0
```

Procedure release(*j:integer*)

```
1: if UC[j] ≠ Null then
2:    UC[j]↑.RC← UC[j]↑.RC−1
3:    if UC[j]↑.RC = 0 then
4:       eliminate checkpoint UC[j]↑.IND
5:       delete UC[j]
6:    UC[j] ← Null
```

Procedure link(*j:integer*, *i:integer*)

```
1: UC[j] ← UC[i]
2: UC[j]↑.RC ← UC[j]↑.RC+1
```

Procedure newCCB(*j:integer*, *ind:integer*)

```
1: UC[j] ← new CCB
2: UC[j]↑.IND← ind
3: UC[j]↑.RC← 1
```

Algorithm 2 RDT-LGC at process p_i

Initialization

```
1: initialize()
```

Before *sending m*

```
1: m.DV ← DV                          {piggybacks DV on m}
```

On *receiving m*

```
1: for j ← 1 to n do
2:    if m.DV[j] > DV[j] then          {new causal info}
3:       DV[j] ← m.DV[j]
4:       release(j)
5:       link(j, i)
```

On *taking checkpoint*

```
1: store DV with the checkpoint
2: release(i)
3: newCCB(i, DV[i])                   {create new CCB}
4: DV[i] ← DV[i] + 1
```

Theorem 2, p_j is denying the collection of the last stable checkpoint taken by p_i. As we show in the sequence, the CCB of this checkpoint is always referenced by $UC[i]$. Therefore, p_i updates $DV(v_i)[j]$, releases $UC[j]$, and links it to the CCB referenced by $UC[i]$ (lines 3-5). When a new checkpoint is taken, a new CCB is created and, by Theorem 2, $UC[i]$ is updated to reference it (lines 2-3). As p_i cannot receive new causal information about itself in a message, this is the only way the entry $UC[i]$ is updated, ensuring that it always references the last stable checkpoint stored by p_i. The rest of the algorithm refers to dependency vector propagation.

Figure 3 depicts a normal execution of RDT-LGC. For each event shown, we present the contents of DV and UC (in Figure 3, DV is depicted on top of UC). For simplicity, we show only the checkpoint index of the CCB referenced by an entry $UC[j]$ and represent null references by "$*$". Therefore, $UC = (0, 0, *)$ means that $UC[0]$ and $UC[1]$ reference the CCB of the first checkpoint taken (index 0), and $UC[2] = Null$. Remember that $DV[i]$, for a process p_i, is incremented only after a local checkpoint is taken. By the end of this execution, checkpoints s_2^2, s_3^1 and s_3^2 (empty squares) have been eliminated. The only obsolete checkpoint not identified by RDT-LGC is s_2^1. It is retained by p_2 because p_2 does not know that p_3 has taken other checkpoints after s_3^1.

Recovery sessions. A simple way to orchestrate a recovery session is through process synchronization [8, 12]. If global information is available in a single process during recovery, it is possible to eliminate all obsolete checkpoints based on Theorem 1. Let us suppose that every process receives a last interval vector LI such that $LI[j] = last\_s(j) + 1$ in the CCP defined by cut R_F. This cut represents the global state in which the application starts the following normal execution period. In this context, a process

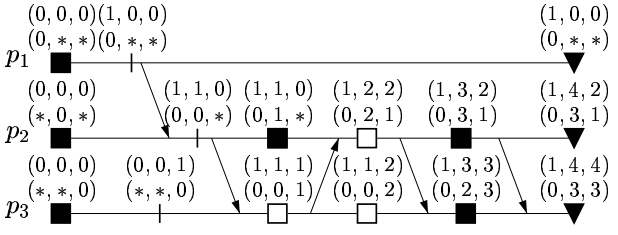

Figure 3: Execution of RDT-LGC.

p_i that must roll back to a previous checkpoint runs Algorithm 3, where RI indicates the index of the checkpoint to which p_i must roll back. Initially, p_i eliminates the checkpoints rolled back and calculates the new dependency vector DV (lines 4-6). After that, p_i finds for every process p_f, based on Theorem 1, the stable checkpoint that must be retained by p_i because of p_f and updates $UC[f]$ accordingly (lines 9-14). Finally, p_i eliminates all checkpoints identified as obsolete (lines 15-17). A process p_i whose component in R_F is its volatile checkpoint does not run this algorithm and can just release any entry $UC[f]$ such that $DV[f] < LI[f]$. If $DV[f] < LI[f]$, the last stable checkpoint of p_f does not causally precede v_i and, by Theorem 1, no checkpoint of p_i must be retained because of p_f.

Algorithm 3 RDT-LGC in a rollback of p_i

```
1:  Input
2:    LI: array[1 .. n] of integer
3:    RI: integer                          {component of p_i in R_F}
4:  eliminate checkpoints s_i^γ | γ > RI
5:  DV ← DV(s_i^RI)                        {recreates DV}
6:  DV[i] ← DV[i] + 1
7:  create a new CCB for every checkpoint s_i^γ stored
8:  for f ← 1 to n do
9:    find γ | LI[f] = DV(c_i^(γ+1))[f] ∧ LI[f] > DV(s_i^γ)[f]
10:   if found γ then
11:     UC[f] ← CCB of s_i^γ               {updates UC[f]}
12:     UC[f]↑.RC ← UC[f]↑.RC+1
13:   else
14:     UC[f] ← Null
15: for all {CCB | RC = 0} do              {obsolete}
16:   eliminate represented checkpoint
17:   delete CCB
```

If an uncoordinated algorithm is used for recovery line calculation [20] and no global information is available during the recovery session, a process that must roll back executes Algorithm 3 replacing LI by DV in line 9. This means that the garbage collection will be based on Theorem 2 instead of Theorem 1. A process that does not roll back simply continues its execution, since its volatile state is ensured to be consistent (w.r.t. the recovery line).

4.4 Correctness and optimality

RDT-LGC ensures that during the execution, every process p_i satisfies the following invariant (see Equation 4). $UC[f] \equiv s_i^\gamma$ means that the entry $UC[f]$ references the CCB of s_i^γ. A formal proof of this invariant is given in Appendix B.

$$s_f^{last} \rightarrow c_i^{\gamma+1} \wedge s_f^{last} \not\rightarrow s_i^\gamma \Rightarrow UC[f] \equiv s_i^\gamma. \quad (4)$$

From this invariant, we can derive a correctness proof for RDT-LGC, as presented in Theorem 3.

Theorem 3 *If Equation 4 holds during the execution of every process p_i, all checkpoints eliminated by RDT-LGC are obsolete.*

Proof: By Theorem 1, if the invariant defined by Equation 4 holds, then every non-obsolete checkpoint has at least one entry $UC[j]$ referencing its CCB. However, in RDT-LGC, a checkpoint is collected only when there is no entry $UC[j]$ referencing its CCB. □

We have defined that an asynchronous garbage collection algorithm relies only on causal knowledge and does not exchange control messages. Now we define optimality in this context.

Definition 9 *An asynchronous garbage collection algorithm is optimal if it collects all obsolete checkpoints that can be identified using causal knowledge.*

Our algorithm is clearly asynchronous, and Theorem 4 shows that it is also optimal.

Theorem 4 *RDT-LGC is an optimal asynchronous garbage collection algorithm.*

Proof: Suppose, by contradiction, that there is an obsolete checkpoint s_i^γ that can be identified with causal knowledge and is not eliminated by RDT-LGC. As it is not collected by RDT-LGC, there is an entry $UC[f]$ in p_i that references its CCB, which means that $s_f^{lastk_i} \rightarrow c_i^{\gamma+1} \wedge s_f^{lastk_i} \not\rightarrow s_i^\gamma$. However, as it is obsolete and can be identified by causal knowledge, by Theorem 1 p_i must have known a checkpoint of p_f taken after $s_f^{lastk_i}$, contradicting the definition of $s_f^{lastk_i}$. □

4.5 Analysis and optimizations

A stable checkpoint s_i^γ is retained by RDT-LGC only if its CCB is referenced by an entry $UC[j]$ in p_i. Therefore, RDT-LGC retains at most n stable checkpoints in a process during its normal execution, which is the least upper bound on checkpoint space overhead in a single process [21]. Actually, a process may need to store $n+1$ checkpoints for a brief period if lines 1 and 2 of event **taking checkpoint** in Algorithm 2 cannot be executed atomically. This happens when a process retaining n checkpoints decides to take a new checkpoint, since the last stable checkpoint previously taken will become obsolete only after the new checkpoint is completely stored in stable storage. There are executions where every process reaches this bound, which gives to RDT-LGC a global space reclamation of at most $n(n+1)$ stable checkpoints. When all obsolete checkpoints are identified and eliminated, the global upper bound is $n(n+1)/2$ checkpoints [21]. However, Theorem 4 shows that it is impossible to give better bounds than RDT-LGC based only on causal knowledge.

Our algorithm runs locally to each process and relies only on dependency vectors propagated as timestamps in the application messages. Moreover, this task actually drives the time complexity of RDT-LGC in normal execution periods. As efficient RDT checkpointing protocols [3, 10, 20] also rely on dependency vector propagation, it becomes straightforward to come up with a merged implementation of checkpointing and garbage collection, without increasing the asymptotic time complexity of the former. The only special remark on a merged implementation concerns the treatment of forced checkpoints. As they are triggered by the receipt of a message and supposed to have been taken before its receipt, the implementor must be careful to ensure that forced checkpoints are indeed stored before the execution of the garbage collection related to the receipt of the message. Appendix C shows how RDT-LGC can be integrated in a well-known RDT protocol (FDAS [20]).

Except for **initialize**, all procedures in Algorithm 1 execute in $O(1)$ time, which gives an $O(n)$ complexity for all the events shown in Algorithm 2. Algorithm 3 runs in $O(n \log n)$ time if line 9 is implemented using a binary search and $O(n)$ checkpoints are stored. Moreover, for cases in which a process has to roll back without having failed, the algorithm can be improved to a time complexity of $O(n)$.

5 Concluding remarks

Garbage collection is highly necessary in systems where the storage space is limited or expensive, like embedded systems and mobile computing. The garbage collection algorithm we presented in this paper ensures that a process will not maintain more than n stored checkpoints during normal execution and does not rely on explicit process synchronization like previous approaches. It relies on the propagation of the same control information as many RDT checkpointing protocols, allowing an efficient merged implementation.

An interesting extension to this work concerns its evaluation in a practical environment. This is motivated by the fact that the theoretical bound on uncollected checkpoints presented in the paper is reached in executions not likely to happen often in practice. Moreover, merged implementations can also be explored in the search for performance improvements. Finally, RDT-LGC is the first garbage collection algorithm based on application messages only. A similar approach could be used to create new efficient garbage collection algorithms based on other properties ensured by checkpointing protocols.

References

[1] A. Agbaria, H. Attiya, R. Friedman, and R. Vitenberg. Quantifying Rollback Propagation in Distributed Checkpointing. In *Proceedings of the 20th Symposium on Reliable Distributed Systems*, pages 36–45, Oct. 2001.

[2] L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain, and A. D. Mel. An Analysis of Communication-Induced Checkpointing. In *Proceedings of the*

29th IEEE Symposium on Fault-Tolerant Computing, pages 242–249, June 1999.

[3] R. Baldoni, J. M. Hélary, A. Mostéfaoui, and M. Raynal. A Communication-Induced Checkpoint Protocol that Ensures Rollback Dependency Trackability. In *Proceedings of the 27th IEEE Symposium on Fault-Tolerant Computing*, pages 68–77, June 1997.

[4] R. Baldoni, J. M. Hélary, and M. Raynal. Rollback-Dependency Trackability: Visible Characterizations. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, pages 33–42, May 1999.

[5] B. Bhargava and S. R. Lian. Independent Checkpointing and Concurrent Rollback for Recovery - An Optimistic Approach. In *Proceedings of the 7th Symposium on Reliable Distributed Systems*, pages 3–12, 1988.

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

[7] Ö. Babaoğlu and K. Marzullo. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.

[8] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, Sept. 2002.

[9] I. C. Garcia and L. E. Buzato. On the Minimal Characterization of the Rollback-Dependency Trackability Property. In *Proceedings of the 21th IEEE Int. Conf. on Distributed Computing Systems*, Apr. 2001.

[10] I. C. Garcia, G. M. D. Vieira, and L. E. Buzato. RDT-Partner: An Efficient Checkpointing Protocol that Enforces Rollback-Dependency Trackability. In *Proceedings of the 19th Brazilian Symposium on Computer Networking*, May 2001.

[11] Y. Huang and C. Kintala. Software Implemented Fault Tolerance: Technologies and Experiences. In *Proceedings of the 16th IEEE Fault-Tolerant Computing Symp.*, pages 2–9, June 1993.

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

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

[14] D. Manivannan and M. Singhal. A Low-Overhead Recovery Technique Using Quasi-Synchronous Checkpointing. In *Proceedings of the 16th IEEE Int. Conf. on Distributed Computing Systems*, May 1996.

[15] D. Manivannan and M. Singhal. Quasi–Synchronous Checkpointing: Models, Characterization, and Classification. *IEEE Trans. on Parallel and Distributed Systems*, 10(7):703–713, July 1999.

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

[17] B. Randell. System Structure for Software Fault Tolerance. *IEEE Trans. on Software Engineering*, 1(2):220–232, June 1975.

[18] R. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Trans. on Computing Systems*, 3(3):204–226, Aug. 1985.

[19] J. Tsai, S. Y. Kuo, and Y. M. Wang. Theoretical Analysis for Communication-Induced Checkpointing Protocols with Rollback-Dependency Trackability. *IEEE Trans. on Parallel and Distributed Systems*, 9(10):963–971, Oct. 1998.

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

[21] Y. M. Wang, P. Y. Chung, I. J. Lin, and W. K. Fuchs. Checkpoint Space Reclamation for Uncoordinated Checkpointing in Message-Passing Systems. *IEEE Trans. on Parallel and Distributed Systems*, 6(5):546–554, May 1995.

Appendix A

Lemma 1 *Given a CCP and a set F of faulty processes, the recovery line R_F is determined by:*

$$R_F = \bigcup_{i=0}^{n-1} \{c_i^{max(k)} \mid \forall p_f \in F, s_f^{last} \not\rightarrow c_i^k\}.$$

Proof: Three things must be shown about our determination of R_F (hereafter referred just by R_F): it is well-defined, it is a consistent global checkpoint, and it minimizes the number of checkpoints rolled back. First, we have seen in Section 2.2 that the first checkpoint of each process, s_i^0, is not causally preceded by any other checkpoint and, thus, by the maximization condition of k for each process, R_F is composed of exactly one local checkpoint of each process.

Now suppose, by contradiction, that R_F is inconsistent. Thus, there are two checkpoints in it, say c_a^α and c_b^β, such that $c_a^\alpha \rightarrow c_b^\beta$. This causal precedence implies that c_a^α is stable and, thus, there is a checkpoint $c_a^{\alpha+1}$ causally preceded by the checkpoint s_f^{last} of a faulty process p_f. If $f = a$, $c_a^\alpha = s_f^{last}$ and $s_f^{last} \rightarrow c_b^\beta$, contradicting the definition of R_F. If $f \neq a$, we have a C-path ζ from s_f^{last} to $c_a^{\alpha+1}$ and a C-path η from c_a^α to c_b^β. As $s_f^{last} \not\rightarrow c_a^\alpha$, the last message of ζ must have been received in $I_a^{\alpha+1}$, forming a zigzag path from s_f^{last} to c_b^β by the concatenation of ζ and η. However, by the definition of the RDT property, $s_f^{last} \rightsquigarrow c_b^\beta \Rightarrow s_f^{last} \rightarrow c_b^\beta$, contradicting our definition of R_F.

Lastly, suppose R_F does not minimize the number of checkpoints rolled back. Therefore, there is a recovery line R'_F which uses at least one checkpoint $c_e^{\epsilon'}$ such that, given the unique checkpoint $c_e^\epsilon \in R_F$ of process p_e, $\epsilon' > \epsilon$. By the definition of R_F, there is a checkpoint s_f^{last} of a faulty process p_f such that $s_f^{last} \rightarrow c_e^{\epsilon'}$, otherwise $c_e^{\epsilon'}$ would be chosen to compose R_F. As p_f is a faulty process, it must be rolled back to one of its stable local checkpoints. Nevertheless, if $s_f^{last} \rightarrow c_e^{\epsilon'}$ then every stable checkpoint of process p_f causally precedes c_e^ϵ and, thus, cannot compose a consistent global checkpoint with it, contradicting our assumption about the existence of the recovery line R'_F. □

Lemma 2 *Every stable checkpoint s_i^γ, part of the recovery line for a set of faulty processes F in a CCP, is also part of the recovery line for a single faulty process p_f in the same CCP, that is,*

$$s_i^\gamma \in R_F \Rightarrow \exists p_f \in \Pi \mid s_i^\gamma \in R_{\{p_f\}}.$$

Proof: By Lemma 1, there exists at least one faulty process p_f such that $s_f^{last} \rightarrow c_i^{\gamma+1}$. As $s_f^{last} \not\rightarrow s_i^\gamma$, s_i^γ would be a member of the recovery line $R_{\{p_f\}}$. □

Lemma 3 *A stable checkpoint s_i^γ is obsolete in the CCP defined by a consistent cut $\mathcal{C}$ iff it is $\mathcal{C}$-needless.*

Proof: *Sufficiency*($\Leftarrow$): We prove the sufficiency of this condition through Claims 1 and 2, presented next. Claim 1 shows that a needless checkpoint remains needless during normal execution periods whereas Claim 2 shows that it is rolled back or remains needless during rollbacks. Therefore, by applying both claims together we show that a needless checkpoint remains needless for the rest of the execution, that is, it is obsolete. *Necessity*($\Rightarrow$): If s_i^γ is not $\mathcal{C}$-needless, there is a process p_f such that $s_i^\gamma \in R_{\{p_f\}}$. So, if p_f fails, s_i^γ will be part of the recovery line. Therefore, it is not obsolete. □

Claim 1 *If s_i^γ is $\mathcal{C}$-needless, it is $\mathcal{L}$-needless for every consistent cut $\mathcal{L}$ such that $\mathcal{C} \subset \mathcal{L}$ ($\mathcal{L}$ is in the future of $\mathcal{C}$).*

Proof: Suppose, by contradiction, that s_i^γ is $\mathcal{C}$-needless and there exists a consistent cut $\mathcal{L}$ such that $\mathcal{C} \subset \mathcal{L}$ and s_i^γ takes part in a recovery line of $\mathcal{L}$. By Lemma 1 we have that s_i^γ is not the last stable checkpoint of p_i in cut $\mathcal{C}$, otherwise it would be part of the recovery line $R_{\{p_i\}}$. Moreover, by Lemma 2 there must be a process p_f such that s_f^{last}, in cut $\mathcal{L}$, causally precedes $s_i^{\gamma+1}$ and does not precede s_i^γ. However, by the definition of a consistent cut, s_f^{last} in $\mathcal{L}$ must also be

part of consistent cut $\mathcal{C}$ (Figure 4). As s_f^{last} is the last stable checkpoint of p_f in $\mathcal{L}$ (and, thus, also in $\mathcal{C}$), s_i^γ must be part of the recovery line $R_{\{p_f\}}$ in $\mathcal{C}$, contradicting the assumption that it is $\mathcal{C}$-needless. □

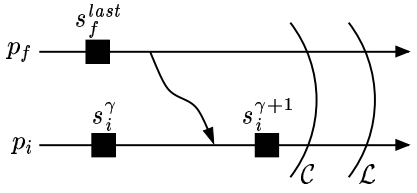

Figure 4: Needless checkpoint in a future cut.

Claim 2 *If s_i^γ is $\mathcal{C}$-needless, it is either nonexistent or needless in R_F for any set $F \subseteq \Pi$.*

Proof: Let $\mathcal{C}'$ be the leftmost consistent cut containing s_i^γ, such that all the consistent cuts containing s_i^γ are in the future of $\mathcal{C}'$. Suppose, by contradiction, that there is a recovery line R_F in $\mathcal{C}$ that rolls the application back to a consistent state in which s_i^γ exists and is not needless. As s_i^γ is $\mathcal{C}$-needless, it is not the last stable checkpoint of p_i in $\mathcal{C}$ and R_F does not contain it. Therefore, the consistent cut defined by R_F is in the future of $\mathcal{C}'$ and in the past of $\mathcal{C}$ ($\mathcal{C}' \subseteq R_F \subseteq \mathcal{C}$). Moreover, as s_i^γ is not R_F-needless, by Lemma 2 there must be a process p_f such that $s_f^{last} \rightarrow s_i^{\gamma+1} \wedge s_f^{last} \not\rightarrow s_i^\gamma$. For consistency, the relation $s_f^{last} \rightarrow s_i^{\gamma+1}$ must be contained in R_F and implies that the checkpoint of process p_f in R_F is volatile as we show in Figure 5. But, if it is volatile, it represents the state of process p_f in $\mathcal{C}$. As a result, in $\mathcal{C}$, $s_i^\gamma \in R_{\{p_f\}}$, contradicting our assumption that s_i^γ is $\mathcal{C}$-needless. □

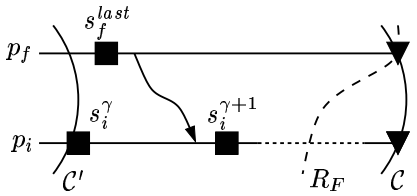

Figure 5: Needless checkpoint in a rollback.

Appendix B

Theorem 5 *RDT-LGC satisfies the invariant described by Equation 4 during the execution of every process p_i:*

$$s_f^{last} \rightarrow c_i^{\gamma+1} \wedge s_f^{last} \not\rightarrow s_i^\gamma \Rightarrow UC[f] \equiv s_i^\gamma. \quad (4)$$

Proof: Initially, the invariant is trivially true. Now let us analyze the events where the terms of Equation 4 are modified.

New causal precedence from p_f to p_i. When a new dependency $s_f^\iota \rightarrow v_i$ is created by the receipt of a message, RDT-LGC makes $UC[f]$ reference s_i^{last}. If $\iota = last\_s(f)$, the new reference makes Equation 4 hold. If $\iota < last\_s(f)$, no relation $s_f^{last} \rightarrow c_i^{\gamma+1}$ exists and Equation 4 holds independently of $UC[f]$'s value.

New checkpoint taken by p_i. When p_i takes a new checkpoint, RDT-LGC makes $UC[i]$ reference s_i^{last}, since $s_i^{last} \rightarrow v_i \wedge s_i^{last} \not\rightarrow s_i^{last}$ (left term of Equation 4 when $p_i = p_f$).

New checkpoint taken by p_f. When $p_f \neq p_i$ takes a new checkpoint, the left term of Equation 4 becomes false and the equation as a whole holds independently of $UC[f]$'s value in p_i.

Rollback of p_i. When global information is available during a rollback, Algorithm 3 updates entries $UC[f]$ to satisfy Equation 4 and makes them *Null* if the left term is false. If there is no global information, Algorithm 3, modified to use DV instead of LI, will make this update based on $last\_k_i(f)$ instead of $last\_s(f)$. If $last\_k_i(f) = last\_s(f)$, the update satisfies the invariant, and if $last\_k_i(f) < last\_s(f)$, no relation $s_f^{last} \rightarrow c_i^{\gamma+1}$ exists and Equation 4 holds independently of $UC[f]$'s value.

Rollback of p_f. If $p_f \neq p_i$ rolls back to a stable checkpoint s_f^ϵ, this checkpoint becomes s_f^{last} and will not precede any checkpoint of p_i in the following normal execution period. Therefore the left term of the equation will be false and the invariant will hold. □

Appendix C

Algorithm 4 presents a merged implementation of the well-known RDT checkpointing protocol FDAS (Fixed-Dependency-After-Send) [20] together with RDT-LGC. The main difference from a simple implementation of FDAS are the calls to the procedures presented in Algorithm 1, shown in bold. As we can easily see, there is almost no overhead in implementing RDT-LGC with RDT checkpointing protocols.

Algorithm 4 FDAS with RDT-LGC at p_i

Data Structures {*besides those in Algorithm 1*}

```
1: Var
2:    sent: boolean
```

Initialization

```
1: sent ← false
2: initialize()
```

Before *sending* m

```
1: sent ← true
2: m.DV ← DV
```

On *receiving* m

```
1: forced ← true
2: for j ← 0 to n do
3:    if m.DV[j] > DV[j] then
4:       if forced then
5:          take checkpoint                {forced checkpoint}
6:          forced ← false
7:       release(j)
8:       link(j, i)
9:       DV[j] ← m.DV[j]
```

On *taking checkpoint* {*basic or forced*}

```
1: sent ← false
2: store DV with the checkpoint
3: release(i)
4: newCCB(i, DV[i])
5: DV[i] ← DV[i] + 1
```