An Index-Based Checkpointing Algorithm for Autonomous Distributed Systems*

Roberto BALDONI Francesco QUAGLIA Paolo FORNARA
Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italy
E.mail: {baldoni,quaglia}@dis.uniroma1.it

Abstract

This paper presents an index-based checkpointing algorithm for distributed systems with the aim of reducing the total number of checkpoints while ensuring that each checkpoint belongs to at least one consistent global checkpoint (or recovery line). The algorithm is based on an equivalence relation defined between pairs of successive checkpoints of a process which allows, in some cases, to advance the recovery line of the computation without forcing checkpoints in other processes.

The algorithm is well suited for *autonomous* and *heterogeneous* environments where each process does not know any private information about other processes and private information of the same type of distinct processes is not related (e.g., clock granularity, local checkpointing strategy, etc.). We also present a simulation study which compares the checkpointing-recovery overhead of this algorithm to the ones of previous solutions.

Index terms: Checkpointing, Causal Dependency, Protocols, Timestamp Management, Global Snapshot, Fault-Tolerance, Rollback-Recovery, Distributed Systems, Performance Evaluation.

*This paper appeared in IEEE Transactions on Parallel and Distributed Systems (vol.10, no.2, February 1999). It is an expanded version of a paper by the same title and the same authors appeared in Proc. 16th Symposium on Reliable Distributed Systems (SRDS).

1 Introduction

Checkpointing is one of the techniques for providing fault-tolerance in distributed systems [5]. A *global checkpoint* consists of a set of local checkpoints, one for each process, from which a distributed computation can be restarted after a failure. A *local checkpoint* is a state of a process saved onto stable storage. Informally, a global checkpoint is *consistent* if no local checkpoint in that set *happens before* [8] another one [3, 9].

Three classes of algorithms have been proposed in the literature to determine consistent global checkpoints: *uncoordinated*, *coordinated* and *communication-induced* [5]. In the first class, processes take local checkpoints independently of each other and upon the occurrence of a failure, a procedure of rollback-recovery tries to build a consistent global checkpoint. Note that, a recent consistent global checkpoint might not exist producing a *domino effect* [13] which, in the worst case, rolls back the computation to its initial state.

In the second class, an initiator process forces other processes, during a failure-free computation, to take a local checkpoint by using control messages. The coordination can be either blocking [3] or non-blocking [7]. However, in both cases, the last local checkpoint of each process belongs to a consistent global checkpoint.

In the third class, the coordination is done in a lazy fashion by piggybacking control information on application messages. Each process takes some local checkpoints, namely *basic* checkpoints, at its own pace, then the lazy coordination induces some additional local checkpoints, namely *forced* checkpoints, in order to determine consistent global checkpoints. Communication-induced checkpointing algorithms can be classified in two distinct categories: *model-based* and *index-based* [5]. Algorithms in the first category, for example [1, 16], have the target to mimic a piece-wise deterministic behavior for each process [6, 15] as well as providing the domino-free property. Index-based algorithms associate each local checkpoint with a sequence number and try to enforce consistency among local checkpoints with the same sequence number [2, 4, 10]. Index-based algorithms ensure domino-free rollback with, generally, less overhead, in terms of number of checkpoints and control information, than model-based ones.

In this paper we present an index-based checkpointing algorithm that reduces the checkpointing overhead, in terms of number of forced checkpoints, compared to previous index-based algorithms. Our algorithm is well suited for *autonomous* and *heterogeneous* environments where each process does not have any private information of other processes and private information of the same type in distinct processes is not related (e.g., clock granularity, local checkpointing strategy, etc.).

To design our algorithm, we extract the rules, used by index-based algorithms, to update the sequence number (i.e., timestamp management rules). This points out that forced checkpoints are due to the process of fast increasing of the sequence numbers. So, in order to slow down this phenomenon, we define an equivalence relation between successive checkpoints of a process.

This relation allows a recovery line to advance without increasing its sequence number. From an operational point of view, the equivalence between checkpoints can be detected by a process exploiting causal dependencies between checkpoints.

The algorithm proposed in this paper embeds such a mechanism to detect equivalences between checkpoints by using a vector of integers piggybacked on application messages. In the worst case, our algorithm takes the same number of checkpoints as the algorithm in [10]. The advantages of our algorithm are quantified by a simulation study showing that the checkpointing overhead can be reduced up to 30% compared to the best previous solution. The price we pay is that each application message piggybacks more control information (one vector of integers) compared to previous proposals. We also investigate the impact of the reduction of the checkpointing overhead on the rollback extent during a recovery, and we show that the amount of undone computation is very close to the one of the algorithm in [10].

The paper is organized as follows. Section 2 presents the system model. Section 3 presents the relation of equivalence between checkpoints. Then, we introduce the data structures and processes actions required by an index-based algorithm to track on-the-fly the equivalence relation. Section 4 describes the proposed index-based algorithm and its correctness proof. Section 5 presents the simulation study.

2 Model of the Distributed Computation

We consider a distributed computation consisting of n processes $\{P_1, P_2, \ldots, P_n\}$ which interact by message passing. Each pair of processes is connected by a two-way reliable channel whose transmission delay is unpredictable but finite.

Processes are *autonomous* in the sense that: they do not share memory, do not share a common clock value[1] and do not have access to private information of other processes such as clock drift, clock granularity, clock precision and speed. Moreover, processes are *heterogeneous* in the sense that private information of the same type of distinct processes is not correlated. We assume, finally, processes follow a fail-stop behavior [14].

A process produces a sequence of events and the h-th event in process P_i is denoted as $e_{i,h}$; each event moves the process from one state to another. We assume events are produced by the execution of internal, send or receive statements.

The send and receive events of a message m are denoted respectively with $send(m)$ and $receive(m)$. A *distributed execution* $\hat{E}$ can be modeled as a partial order of events $\hat{E} = (E, \rightarrow)$ where E is the set of all events and $\rightarrow$ is the *happened-before relation* [8] defined as follows:

[1]The index-based algorithm presented in [4] assumes, for example, a standard clock synchronization algorithm, which provides a common clock value to each process.

Definition 2.1 *An event $e_{i,h}$ precedes an event $e_{j,k}$, denoted $e_{i,h} \rightarrow e_{j,k}$, iff:*

- $i = j$ *and* $k = h + 1$; *or*
- $e_{i,h} = send(m)$ *and* $e_{j,k} = receive(m)$; *or*
- $\exists e_{l,z} \ : \ (e_{i,h} \rightarrow e_{l,z}) \wedge (e_{l,z} \rightarrow e_{j,k})$

A checkpoint C dumps the current process state onto stable storage. A checkpoint of process P_i is denoted as $C_{i,sn}$ where sn is called the *index*, or *sequence number*, of a checkpoint. Each process takes checkpoints either at its own pace (*basic* checkpoints) or induced by some communication pattern (*forced* checkpoints). We assume that each process P_i takes an initial basic checkpoint $C_{i,0}$ and that, for the sake of simplicity, basic checkpoints are taken by a periodic algorithm. We use the notation $next(C_{i,sn})$ to indicate the successive checkpoint, taken by P_i, after $C_{i,sn}$. A checkpoint interval $I_{i,sn}$ is the set of events between $C_{i,sn}$ and $next(C_{i,sn})$. Checkpoints are ordered by a relation of precedence, denoted $\rightarrow_C$, and defined as follows:

Definition 2.2 *A checkpoint $C_{i,h}$ precedes a checkpoint $C_{j,k}$, denoted $C_{i,h} \rightarrow_C C_{j,k}$, iff:*

$$\exists e_{i,l} \in I_{i,g},\ \exists e_{j,m} \in I_{j,a} \ : \ (g \geq h) \wedge (a < k) \wedge (e_{i,l} \rightarrow e_{j,m})$$

More simply, a checkpoint $C_{i,h}$ precedes a checkpoint $C_{j,k}$ if there is a causal path of messages starting after $C_{i,h}$ and ending before $C_{j,k}$.

A *global checkpoint* $\mathcal{C}$ is a set of local checkpoints $\{C_{1,sn_1}, C_{2,sn_2}, \ldots, C_{n,sn_n}\}$ one for each process.

Definition 2.3 *A global checkpoint $\mathcal{C} = \{C_{1,sn_1}, C_{2,sn_2}, \ldots, C_{n,sn_n}\}$ is consistent iff*

$$\forall i, j \in [1, n] \ : \ i \neq j \Rightarrow \neg(C_{i,sn_i} \rightarrow_C C_{j,sn_j})$$

In the following, we denote with $\mathcal{C}_{sn}$ a global checkpoint formed by checkpoints with sequence number sn and use the term consistent global checkpoint $\mathcal{C}_{sn}$ and recovery line $\mathcal{L}_{sn}$ interchangeably.

3 The Relation of Equivalence

In this section, we first recall a classical index-based algorithm showing the basic rules to generate a recovery line $\mathcal{L}_{sn}$. After introducing the equivalence relation, we point out the new data structures and processes actions, required by an index-based algorithm, to track such a relation on-the-fly.

3.1 How to Form a Recovery Line $\mathcal{L}_{sn}$

The simplest way to form a recovery line is, each time a basic checkpoint $C_{i,sn}$ is taken by process P_i, to start an explicit coordination. This coordination results in a recovery line $\mathcal{L}_{sn}$ associated to $C_{i,sn}$. This strategy induces $n-1$ forced checkpoints (one for each process) per basic checkpoint. Briatico at al. [2] argued that the previous "centralized" strategy can be "decentralized" in a lazy fashion by piggybacking on each application message m the index sn of the last checkpoint taken (denoted $m.sn$).

Let us assume each process P_i has a variable sn_i which represents the sequence number of the last checkpoint. Then, the Briatico-Ciuffoletti-Simoncini (BCS) algorithm can be sketched by using the following rules associated with the action to take a local checkpoint:

`take-basic(BCS)` :

When a basic checkpoint is scheduled:

$sn_i \leftarrow sn_i + 1$;

a checkpoint C_{i,sn_i} is taken;

`take-forced(BCS)` :

Upon the receipt of a message m

if $sn_i < m.sn$

then $sn_i \leftarrow m.sn$;

a checkpoint $C_{i,m.sn}$ is taken;

the message is processed;

By using the above rules, it has been proved that the set of checkpoints with the same sequence number sn is a recovery line $\mathcal{L}_{sn}$ [2]. Note that, due to the rule `take-forced(BCS)`, there could be some gap in the index assigned to checkpoints by a process. Hence, if a process has not assigned the index sn, the first local checkpoint of the process with sequence number greater than sn can be included in the recovery line $\mathcal{L}_{sn}$.

Each time a basic checkpoint is taken, sn is increased by one and the process starts a lazy coordination to build the recovery line $\mathcal{L}_{sn}$. In the worst case, the number of forced checkpoints induced by a basic one is $n-1$. In the best case, if all processes take a basic checkpoint at the same physical time, the number of forced checkpoints per basic one is zero. However, in an autonomous and heterogeneous environment, periods of basic checkpoints in distinct processes are not related and, in any case, they would tend to diverge due to many causes (clock speed, process speed, temperature etc.). This pushes the sequence numbers of some processes higher and each time one of such processes sends a message to another one, it is likely that a number of forced checkpoints, close to $n-1$, will be induced.

From the above discussion, it follows that, the cause of the forced checkpoints is the increasing of the sequence number done in the `take-basic(BCS)` rule whenever a basic checkpoint is scheduled. So, in the next subsection we introduce an equivalence relation, defined on pairs of successive checkpoints of a process, which allows the recovery line to advance without increasing its sequence number.

3.2 Equivalence Between Checkpoints

Definition 3.1 *Two local checkpoints $C_{i,sn}$ and $next(C_{i,sn})$ of process P_i are equivalent with respect to the recovery line $\mathcal{L}_{sn}$ (including $C_{i,sn}$), denoted $C_{i,sn} \overset{\mathcal{L}_{sn}}{\equiv} next(C_{i,sn})$, if*

$$\forall C_{j,sn} \in \mathcal{L}_{sn} \ : \ j \neq i \Rightarrow \neg(C_{j,sn} \rightarrow_C next(C_{i,sn}))$$

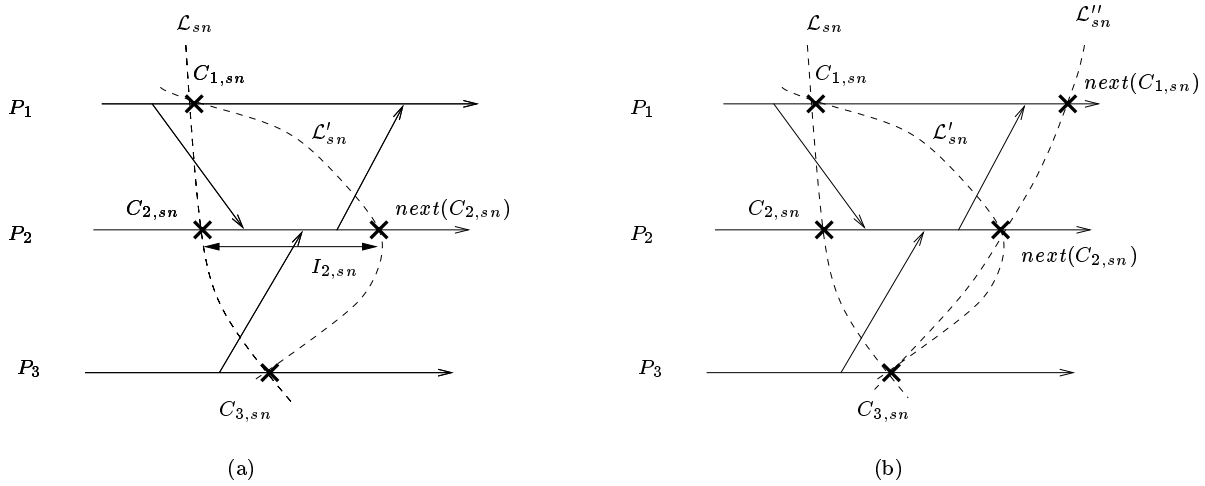

Figure 1: Examples of Pairs of Equivalent Checkpoints.

As an example, consider the recovery line $\mathcal{L}_{sn}$ depicted in Figure 1.a, where checkpoints are depicted by thick crosses and arrows between processes represent messages. If in $I_{2,sn}$ process P_2 executes either send events or receive events of messages which have been sent from the left side of $\mathcal{L}_{sn}$, then $C_{2,sn} \overset{\mathcal{L}_{sn}}{\equiv} next(C_{2,sn})$ and a recovery line $\mathcal{L}'_{sn}$ can be created by replacing $C_{2,sn}$ with $next(C_{2,sn})$ from $\mathcal{L}_{sn}$. Figure 1.b shows the construction of the recovery line $\mathcal{L}''_{sn}$ starting from $\mathcal{L}'_{sn}$ by using the equivalence between $C_{1,sn}$ and $next(C_{1,sn})$ with respect to $\mathcal{L}'_{sn}$. Hence we can say, $C_{i,sn}$ is not equivalent to $next(C_{i,sn})$ with respect to $\mathcal{L}_{sn}$ if at least one message is sent from the right side of $\mathcal{L}_{sn}$ and is received by P_i in $I_{i,sn}$.

From the above examples, a simple property follows:

Property 3.1
If $C_{i,sn} \overset{\mathcal{L}_{sn}}{\equiv} next(C_{i,sn})$ **then** $\mathcal{L}'_{sn} = \mathcal{L}_{sn}\text{-}\{C_{i,sn}\} \cup \{next(C_{i,sn})\}$ *is a recovery line.*

Proof If $C_{i,sn} \overset{\mathcal{L}_{sn}}{\equiv} next(C_{i,sn})$, as $\mathcal{L}_{sn}$ is a recovery line including $C_{i,sn}$, then $\forall C_{j,sn} \in \mathcal{L}_{sn} \ : \ j \neq i \Rightarrow \neg(C_{j,sn} \rightarrow_C next(C_{i,sn}))$, so the set of local checkpoints $\mathcal{L}_{sn} - \{C_{i,sn}\} \cup \{next(C_{i,sn})\}$ is a consistent one (see Definition 2.3). □

Hence, if a process detects a pair of equivalent checkpoints, it can advance the recovery line without updating its sequence number. With this aim, in the next subsection we show what a process needs to track pairs of equivalent checkpoints.

3.3 Sequence and Equivalence Numbers of a Recovery line

Suppose process P_i owns two local variables: sn_i and en_i. The variable sn_i stores the number of the current recovery line. The variable en_i represents the number of equivalent local checkpoints with respect to the current recovery line (both sn_i and en_i are initialized to zero).

Let us denote as $C_{i,sn,en}$ the checkpoint of P_i with the sequence number sn and the equivalence number en; the pair $< sn, en >$ is also called the index of a checkpoint. Thus, the initial checkpoint of process P_i will be denoted as $C_{i,0,0}$. The index of a checkpoint is updated according to the following rule:

$$\textbf{if } C_{i,sn,en} \overset{\mathcal{L}_{sn}}{\equiv} next(C_{i,sn,en}) \textbf{ then } next(C_{i,sn,en}) = C_{i,sn,en+1} \textbf{ else } next(C_{i,sn,en}) = C_{i,sn+1,0}$$

Process P_i also has a vector EQ_i of n integers. The j–th entry of the vector represents the knowledge of P_i about the equivalence number of P_j with the current sequence number sn_i (thus the i–th entry corresponds to en_i).

EQ_i is updated according to the following rule: each application message m sent by process P_i piggybacks the current sequence number sn_i ($m.sn$) and the current EQ_i vector ($m.EQ$). Upon the receipt of a message m at process P_i, if $m.sn = sn_i$, EQ_i is updated from $m.EQ$ by taking a component-wise maximum. If $m.sn > sn_i$, the values in $m.EQ$ and $m.sn$ are copied in EQ_i and sn_i ([2]). An example of the updating of the vector EQ is shown in Figure 2. Message m' brings to P_1 the information about the increasing of the equivalence number of P_2.

Let us remark that the set $\cup_{\forall j} C_{j,sn,EQ_i[j]}$ is a recovery line (a formal proof of this property is given in Theorem 4.8). So, to the knowledge of P_i, the vector EQ_i actually represents the most recent recovery line with sequence number sn_i.

3.4 Tracking the Equivalence Relation On-The-Fly

When considering an index-based algorithm, as the one presented in Section 3.1, we have to define which type of checkpoint plays a role in the equivalence relation. The events influencing the

[2]The vector EQ can be seen as a vector timestamp [11] when considering checkpoints with the same sequence number sn as relevant events of a distributed computation.

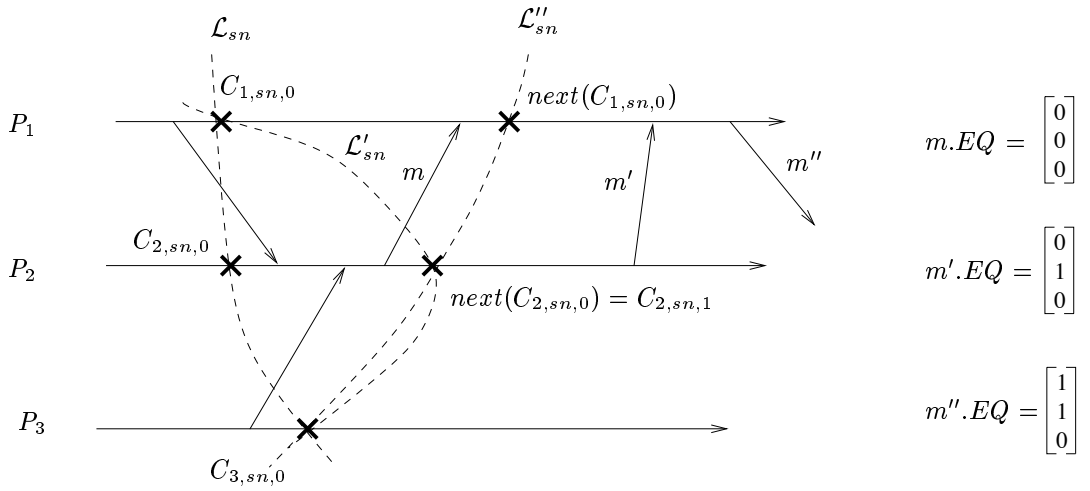

Figure 2: Upon the Receipt of m', P_1 Detects $C_{1,sn,0} \overset{\mathcal{L}'_{sn}}{\equiv} next(C_{1,sn,0})$.

detection of the equivalence are: the arrival of a message (which enlarge the knowledge about the causal past of a process) and the event of taking a basic checkpoint.

Upon the arrival of a message m at P_i in the checkpoint interval $I_{i,sn,en}$: one of the following three cases is true:

1) $(m.sn < sn_i)$ or $((m.sn = sn_i)$ and $(\forall j\ m.EQ[j] < EQ_i[j]))$;
   % m has been sent from the left side of the recovery line $\cup_{\forall j} C_{j,sn,EQ_i[j]}$ %
2) $(m.sn = sn_i)$ and $(\exists j :\ m.EQ[j] \geq EQ_i[j])$;
   % m has been sent from the right side of the recovery line $\cup_{\forall j} C_{j,sn,EQ_i[j]}$ %
3) $(m.sn > sn_i)$;
   % m has been sent from the right side of a recovery line of which P_i was not aware %

A message m which falls in case 3) directs P_i to take a forced checkpoint $C_{i,m.sn,0}$([3]). So, the only interesting cases for tracking the equivalence are 1) and 2).

At the time of the basic checkpoint $next(C_{i,sn,en})$: P_i falls in one of the following two alternatives:

(i) If no message is received in $I_{i,sn,en}$ that falls in case 2, then $C_{i,sn,en} \overset{\mathcal{L}_{sn}}{\equiv} next(C_{i,sn,en})$. That equivalence can be tracked by a process using its local context at the time of the checkpoint $next(C_{i,sn,en})$. Thus $next(C_{i,sn,en}) = C_{i,sn,en+1}$ (the equivalence between $C_{2,sn,0} \overset{\mathcal{L}_{sn}}{\equiv} next(C_{2,sn,0})$, shown in Figure 2, is an example of such a behavior);

[3] After taking a forced checkpoint, message m falls in case 2) with respect to the checkpoint interval $I_{i,m.sn,0}$.

(ii) If there exists at least a message m received in $I_{i,sn,en}$ which falls in case 2, one checkpoint belonging to the recovery line $\cup_{\forall j} C_{j,sn,EQ_i[j]}$ precedes $next(C_{i,sn,en})$ (this communication pattern is shown in Figure 2 where, $\cup_{\forall j} C_{j,sn,EQ_1[j]} = \{C_{1,sn,0}, C_{2,sn,0}, C_{3,sn,0}\}$ and due to m, $C_{2,sn,0} \rightarrow_C next(C_{1,sn,0})$). The consequence is that process P_i cannot determine, at the time of taking the basic checkpoint $next(C_{i,sn,en})$, if $C_{i,sn,en}$ is equivalent to $next(C_{i,sn,en})$ with respect to some recovery line. As an example, in Figure 2 process P_1 cannot determine if $C_{1,sn,0}$ is equivalent to $next(C_{1,sn,0})$ with respect to some recovery line when taking $next(C_{1,sn,0})$.

To solve the problem raised in point (ii), two approaches can be pursued. If, at the time of the basic checkpoint $next(C_{i,sn,en})$, the equivalence between $C_{i,sn,en}$ and $next(C_{i,sn,en})$ is undetermined then:

Pessimistic Approach. Process P_i assumes pessimistically $next(C_{i,sn,en}) = C_{i,sn+1,0}$ even though this determination could be revealed wrong in the future of the computation. Figure 2 shows a case in which message m' brings the information (encoded in $m'.EQ$) to P_1 that $C_{2,sn,0} \overset{\mathcal{L}_{sn}}{\equiv} next(C_{2,sn,0})$ and that the recovery line was advanced, by P_2, from $\mathcal{L}_{sn}$ to $\mathcal{L}'_{sn}$. In such a case, P_1 can determine $C_{1,sn,0}$ is equivalent to $next(C_{1,sn,0})$ with respect to $\mathcal{L}'_{sn}$ ([4]).

Optimistic Approach. Process P_i assumes *optimistically* (and *provisionally*) that $C_{i,sn,en}$ is equivalent to $next(C_{i,sn,en})$. So the index of $next(C_{i,sn,en})$ becomes $< sn, en+1 >$. As provisional indices cannot be propagated in the system, if *at the time of the first send event after* $next(C_{i,sn,en})$ the equivalence is still undetermined, then $next(C_{i,sn,en}) = C_{i,sn+1,0}$ (thus, $sn_i = sn_i + 1$, $en_i = 0$, and $\forall j : EQ_i[j] = 0$). Otherwise, the provisional index becomes permanent. Figure 2 shows a case in which $C_{1,sn,0} \overset{\mathcal{L}'_{sn}}{\equiv} next(C_{1,sn,0})$ and this is detected by P_i before sending m''. After m'' is sent, the index $< sn, 1 >$ of $next(C_{1,sn,0})$ becomes permanent.

4 An Index-Based Checkpointing Algorithm

In this section we propose an index-based checkpointing algorithm that follows an optimistic approach. The algorithm is based also on other two "practical" observations with the aim to reduce the total number of checkpoints. The first comes from the Manivannan-Singhal algorithm [10] and will be explained in the following subsection. The second observation is shown in Subsection 4.2.

[4] A simple implementation of the pessimistic approach requires each process P_i to be endowed with a boolean variable $equiv_i$. P_i sets $equiv_i$ to TRUE each time a new checkpoint interval $I_{i,sn,0}$ starts and $equiv_i$ is set to FALSE whenever a message m such that $m.sn = sn$ is received in $I_{i,sn,0}$. Upon scheduling $next(C_{i,sn,0})$, if $\neg(equiv_i)$ then $next(C_{i,sn,0}) = C_{i,sn+1,0}$. This implementation does not require to piggyback the vector EQ.

4.1 The Manivannan-Singhal Algorithm

To reduce the number of checkpoints, an interesting observation comes from the Manivannan-Singhal algorithm [10] which has been designed for non-autonomous distributed systems.

Observation 4.1 *There is no reason to take a basic checkpoint if at least one forced checkpoint has been taken during the interval between two scheduled basic checkpoints.*

So, let us assume process P_i has a flag $skip_i$ which indicates if at least one forced checkpoint is taken in the current checkpoint period (this flag is set to FALSE each time a basic checkpoint is scheduled, and set to TRUE each time a forced checkpoint is taken). A version of Manivannan-Singhal (MS) algorithm, derived from the BCS one, well suited for autonomous environment can be sketched by the following rules:

`take-basic(MS)` :

When a basic checkpoint is scheduled:

if $skip_i$ **then** $skip_i = FALSE$

else $sn_i \leftarrow sn_i + 1$;

a checkpoint C_{i,sn_i} is taken;

`take-forced(MS)` :

Upon the receipt of a message m:

if $sn_i < m.sn$

then $sn_i \leftarrow m.sn$;

$skip_i \leftarrow TRUE$;

a checkpoint $C_{i,m.sn}$ is taken;

the message is processed;

4.2 The Algorithm

The checkpointing algorithm we propose (BQF) consists of three rules `take-basic(BQF)`, `take-forced(BQF)` and `send-message(BQF)` as it follows an optimistic approach.

`take-basic(BQF).` It is similar to `take-basic(MS)` rule. However, it does not update the sequence number by optimistically assuming that each basic checkpoint is equivalent to the previous one. Hence, each process P_i has a boolean variable $provisional_i$ which is set to TRUE whenever a provisional index assignment occurs. It is set to FALSE whenever the index becomes permanent. So we have:

`take-basic(BQF)` :

When a basic checkpoint is scheduled:

if $skip_i$ **then** $skip_i = FALSE$;

else $en_i \leftarrow en_i + 1$;

Take a checkpoint C with a provisional index $< sn_i, en_i >$;

$provisional_i \leftarrow TRUE$;

`send-message(BQF).` Due to the presence of provisional indices caused by the equivalence relation, our algorithm needs an additional rule, when sending a message, in order to disseminate only permanent indices of checkpoints. Let us then assume each process P_i has a boolean variable $after\_first\_send_i$ which is set to TRUE if at least one send event has occurred in the current checkpoint interval. It is set to FALSE each time a checkpoint is taken. The actions of the rule `send-message(BQF)` are the following:

`send-message(BQF)` :

Before sending a message m in I_{i,sn_i,en_i}:

if $\neg(after\_first\_send_i)$ and $provisional_i$ **then**

if $\neg(C_{i,sn_i,en_i-1} \overset{\mathcal{L}_{sn}}{\equiv} C_{i,sn_i,en_i})$

then $sn_i \leftarrow sn_i + 1$; $en_i \leftarrow 0$; $\forall j\ EQ_i[j] \leftarrow 0$;

the index $< sn_i, en_i >$ of the last checkpoint becomes permanent;

$provisional_i \leftarrow FALSE$;

$EQ_i[i] \leftarrow en_i$;

the message m is sent piggybacking sn_i and EQ_i;

`take-forced(BQF).` The last rule of our algorithm `take-forced(BQF)` refines BCS's one by using a simple observation.

Observation 4.2 *Upon the receipt of a message m in I_{i,sn_i,en_i} such that $m.sn > sn_i$, there is no reason to take a forced checkpoint if there has been no send event in I_{i,sn_i,en_i}.*

Indeed, no $\rightarrow_C$ relation can be established between the last checkpoint C_{i,sn_i,en_i} and any checkpoint belonging to the recovery line $\mathcal{L}_{m.sn}$ and, thus, the index of C_{i,sn_i,en_i} can be replaced permanently with the index $< m.sn, 0 >$ ([5]).

[5] The Observation 4.2 has been used for the first time by Wang in [16] to develop the Fixed-Dependency-After-Send checkpointing algorithm. This model-based algorithm was designed to ensure the rollback-dependency trackability (RDT) property to a checkpoint and communication pattern. i.e., if there is a dependency between two checkpoints

`take-forced(BQF)` :

Upon the receipt of a message m in I_{i,sn_i,en_i}:

case

$sn_i < m.sn$ **and** $after\_first\_send_i \rightarrow$ /* part (a) */

a forced checkpoint $C_{i,m.sn,0}$ is taken and its index is permanent;

$sn_i \leftarrow m.sn$; $en_i \leftarrow 0$; $skip_i \leftarrow TRUE$; $provisional_i \leftarrow FALSE$;

$\forall j\ EQ_i[j] \leftarrow m.EQ[j]$;

$sn_i < m.sn$ **and** $\neg(after\_first\_send_i) \rightarrow$ /* part (b) */

the index of the last checkpoint C_{i,sn_i,en_i} is replaced permanently with $< m.sn, 0 >$;

$sn_i \leftarrow m.sn$; $en_i \leftarrow 0$; $provisional_i \leftarrow FALSE$;

$\forall j\ EQ_i[j] \leftarrow m.EQ[j]$;

$sn_i = m.sn \rightarrow$ /* part (c) */

$\forall j\ EQ_i[j] \leftarrow max(m.EQ[j], EQ_i[j])$;

end case;

the message m is processed;

For example, in Figure 3.a, the local checkpoint C_{3,sn,en_3} can belong to the recovery line $\mathcal{L}_{sn+1}$ (so the index $< sn, en_3 >$ can be replaced with $< sn+1, 0 >$) given that process P_3 did not send any message between C_{3,sn,en_3} and the receipt of message m, so no causal path of messages starts after C_{3,sn,en_3}, and consequently, no $\rightarrow_C$ relation has been established with other checkpoints. On the contrary, due to the send event of message m' in I_{3,sn,en_3} depicted in Figure 3.b, a forced checkpoint with index $< sn+1, 0 >$ has to be taken before the processing of message m. In this case, as P_3 issued a message, there could be a $\rightarrow_C$ relation between C_{3,sn,en_3} and other checkpoints.

Part (b) of `take-forced(BQF)` decreases the number of forced checkpoints compared to BCS. The **then** alternative of `send-message(BQF)` represents the cases in which the action to take a basic checkpoint leads to update the sequence number with the consequent induction of forced checkpoints in other processes.

4.3 Data Structures and Process Behavior

We assume each process P_i has the following data structures:

sn_i, en_i: **integer**;

$after\_first\_send_i$, $skip_i$, $provisional_i$: **boolean**;

$past_i$, $present_i$, EQ_i: ARRAY[1,n] of **integer**.

due to a "non-causal" sequence of messages, then there must exist a causal sequence of messages which establishes the same dependency.

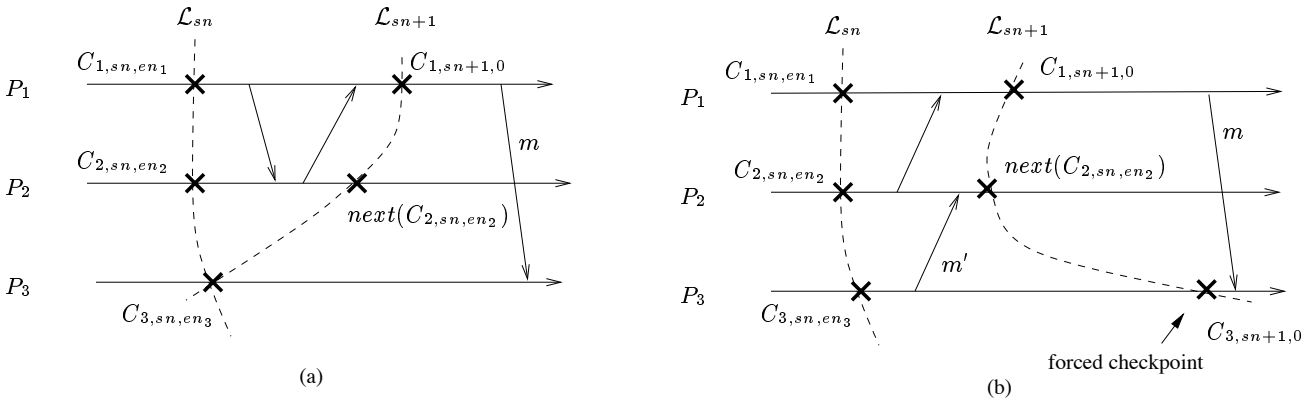

Figure 3: Upon the Receipt of m, C_{3,sn,en_3} can be Part of $\mathcal{L}_{sn+1}$ (a); C_{3,sn,en_3} Cannot Belong to $\mathcal{L}_{sn+1}$ (b).

$present_i[j]$ represents the maximum equivalence number en_j sent by P_j and received by P_i in the current checkpoint interval, and piggybacked on a message that falls in the case 2 of Section 3.4. Upon taking a checkpoint or when updating the sequence number, all the entries of $present_i$ are initialized to -1. If the checkpoint is basic, $present_i$ is copied in $past_i$ before its initialization. Each time a message m is received such that $past_i[h] < m.EQ[h]$, $past_i[h]$ is set to -1. So, the predicate $(\exists h : past_i[h] > -1)$ indicates that there is a message received in the past checkpoint interval that has been sent from the right side of the recovery line *currently* seen by P_i (case 2 of Section 3.4).

Below the process behavior is shown (the procedures and the message handler are executed in atomic fashion). This implementation assumes that there exists at most one provisional index in each process. So each time two successive provisional indices are detected, the first index is permanently replaced with $< sn_i + 1, 0 >$.

init P_i**:**

$sn_i := 0$; $en_i := 0$; $after\_first\_send_i := FALSE$; $skip_i := FALSE$; $provisional_i := FALSE$;
$\forall h$ $EQ_i[h] := 0$; $\forall h$ $past_i[h] := -1$; $\forall h$ $present_i[h] := -1$;

when m **arrives at** P_i `from` P_j**:**

```
if m.sn > sn_i then                          % P_i is not aware of the recovery line with sequence number m.sn %
begin
  if after_first_send_i then
  begin
    take a checkpoint;                                          % taking a forced checkpoint %
    skip_i := TRUE;
    after_first_send_i := FALSE;
  end;
  sn_i := m.sn; en_i := 0;
  assign the index < sn_i, en_i > to the last taken checkpoint;
  provisional_i := FALSE;                                       % the index is permanent %
  ∀h past_i[h] := −1; ∀h present_i[h] := −1;
```

```
  present_i[j] := m.EQ[j];
  ∀h  EQ_i[h] := m.EQ[h];
end
else if m.sn = sn_i then
     begin
       if present_i[j] < m.EQ[j] then present_i[j] := m.EQ[j];
       ∀h  EQ_i[h] := max(EQ_i[h],  m.EQ[h]);                    % a component-wise maximum is performed %
       ∀h  if past_i[h] < m.EQ[h] then past_i[h] := −1;
     end;
process the message m;
```

when P_i sends *data* **to** P_j:

```
if provisional_i ∧ (∃h : past_i[h] > −1) then          % last checkpoint not equivalent to the previous one %
begin
  sn_i := sn_i + 1;  en_i := 0;
  assign the index < sn_i, en_i > to the last taken checkpoint;
  provisional_i := FALSE;                                                  % the index is permanent %
  ∀h  past_i[h] := −1;  ∀h  present_i[h] := −1;  ∀h  EQ_i[h] := 0;
end;
m.content = data;  m.sn := sn_i;  m.EQ := EQ_i;                            % packet the message %
send (m) to P_j;
after_first_send_i := TRUE;
```

when a basic checkpoint is scheduled from P_i:

```
if skip_i then skip_i := FALSE                          % the basic checkpoint is skipped as in [10]%
else
begin
  if provisional_i then                                          % two successive provisional indices %
    if (∃h : past_i[h] > −1) then                % last checkpoint not equivalent to the previous one %
    begin
      ∀h  past_i[h] := −1;
      sn_i := sn_i + 1;  en_i := 0;
      assign the index < sn_i, en_i > to the last taken checkpoint;          % the index is permanent %
      ∀h  EQ_i[h] := 0;
    end
    else ∀h  past_i[h] := present_i[h];          % last checkpoint is equivalent to the previous one %
  take a checkpoint;                                                % taking a basic checkpoint %
  en_i := en_i + 1;
  EQ_i[i] := en_i;
  assign the index < sn_i, en_i > to the last taken checkpoint;
  provisional_i := TRUE;                                            % the index is provisional %
  ∀h  present_i[h] := −1;
  after_first_send_i := FALSE;
end
```

4.4 Correctness Proof

We want to prove that at any time the set $\cup_{\forall j} C_{j,sn,EQ_i[j]}$ is a recovery line. At this aim, let us introduce the following simple observations and lemmas:

Observation 4.3 *For any checkpoint $C_{i,sn,0}$, there does not exist any message m with $m.sn \geq sn$ such that receive$(m) \in I_{i,sn',en}$ with $sn' < sn$ (this observation derives from rule* `take-forced(BQF)` *when considering $C_{i,sn,0}$ is the first checkpoint with sequence number sn).*

Observation 4.4 *For any message m sent by P_i in $I_{i,sn,en}$ or in a later checkpoint interval, then $m.sn \geq sn$ (this observation derives from the rule* `send-message(BQF)`*).*

Observation 4.5 *Let us consider a causal message chain $\mu = (m_f, \ldots, m_l)$. We have $m_l.sn \geq m_f.sn$ (this observation follows from the rules* `take-forced(BQF)` *and* `send-message(BQF)`*).*

Lemma 4.6 *For any pair of checkpoints $(C_{i,sn,en}, C_{j,sn,0})$ the following predicate holds:*

$$\neg(C_{i,sn,en} \rightarrow_C C_{j,sn,0})$$

Proof Suppose by the way of contradiction, that $C_{i,sn,en} \rightarrow_C C_{j,sn,0}$. In this case, there exits a causal message chain μ starting after $C_{i,sn,0}$ whose message m_l is received by P_j in $I_{j,sn'',en}$ (with $sn'' < sn$). Due to Observation 4.4 and Observation 4.5, $m_l.sn \geq sn$ and this contradicts Observation 4.3 □

Lemma 4.7
Let i, j and k be three integers, at any given time for a pair of checkpoints $(C_{i,sn,EQ_k[i]}, C_{j,sn,EQ_k[j]})$ the following predicate holds:

$$\neg(C_{i,sn,EQ_k[i]} \rightarrow_C C_{j,sn,EQ_k[j]})$$

Proof Suppose by the way of contradiction that there exists a causal message chain μ such that:

$$(P) \qquad C_{i,sn,EQ_k[i]} \rightarrow_C C_{j,sn,EQ_k[j]}$$

Four cases have to be considered:
1) if $i = j$ predicate P contradicts Definition 2.2;
2) if $(k = i) \wedge (i \neq j)$:

- if $EQ_i[j] = 0$, Lemma 4.6 is contradicted;
- if $EQ_i[j] > 0$ then (i) $C_{j,sn,EQ_i[j]}$ is equivalent to $C_{j,sn,EQ_i[j]-1}$ and (ii) there exists a causal message chain μ' which brings to P_i the information of that equivalence in the current checkpoint interval $I_{i,sn,EQ_i[i]}$ (see Section 3.3). From Definition 3.1, $C_{j,sn,EQ_i[j]}$ can be equivalent to $C_{j,sn,EQ_i[j]-1}$ only if $EQ_j[i] > EQ_i[i]$. The latter is a contradiction to the fact that the current equivalence number of P_i is $EQ_i[i]$. This case is shown in Figure 4.a.

3) if $(k = j) \wedge (i \neq j)$:

- if $EQ_j[j] = 0$, Lemma 4.6 is contradicted;
- if $EQ_j[j] > 0$ then $C_{j,sn,EQ_j[j]}$ is equivalent to $C_{j,sn,EQ_j[j]-1}$. Let en_i be the value stored in $EQ_j[i]$. From the rule `send-message(BQF)`, an equivalence number is stored in EQ only when the index is permanent. This means that in the interval between the checkpoint $C_{j,sn,EQ_j[j]}$ and the send of first message m, there must exist a causal message chain μ' starting after a checkpoint $C_{i,sn,en}$ (with $en > en_i$) and ending in $I_{j,sn,EQ_j[j]}$ before the sending of m. In such a case the previous equivalence holds. Due to the rules to update the vector EQ (see Section 3.3), after the receipt of the last message of μ', the value stored in $EQ_j[i]$ is en. This contradicts the fact that the value stored in $EQ_j[i]$ is en_i. This case is shown in Figure 4.b.

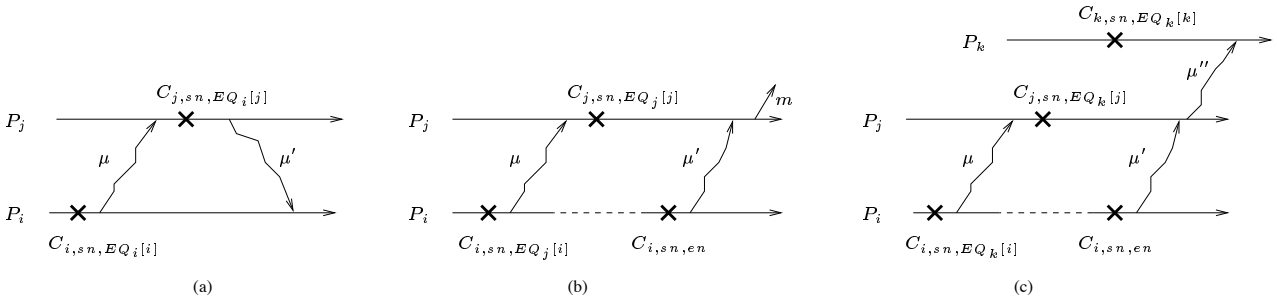

Figure 4: Proof of Lemma 4.7.

4) if $(k \neq i) \wedge (k \neq j) \wedge (i \neq j)$:

- if $EQ_k[j] = 0$, Lemma 4.6 is contradicted;
- if $EQ_k[j] > 0$ then $C_{j,sn,EQ_k[j]}$ is equivalent to $C_{j,sn,EQ_k[j]-1}$. Let en_i be the value stored in $EQ_k[i]$. Due to the initial assumption in order that the equivalence be verified, there must exist (i) a causal message chain μ' starting after a checkpoint $C_{i,sn,en}$ (with $en > en_i$) and ending in $I_{j,sn,EQ_k[j]}$ and (ii) a causal message chain μ'' starting after the receipt of the last message of μ' which brings the information of the equivalence to P_k. Due to the rules to update the vector EQ (see Section 3.3), the value stored in $EQ_k[i]$ is en. This contradicts the fact that the value stored in $EQ_k[i]$ is en_i. This case is shown in Figure 4.c.

In all cases the assumption (P) leads to a contradiction. Then the claim follows. □

Theorem 4.8 *At any given time the set* $S = \cup_{\forall j} C_{j,sn,EQ_i[j]}$ *is a recovery line.*

Proof The proof follows from Lemma 4.7 applied to any distinct pair of checkpoints in S and from the Definition 2.3. □

Remark. Note that each local checkpoint produced by the algorithm belongs to, at least, one recovery line. In particular, $C_{i,sn,en} = next(C_{i,sn',en'})$ with $sn > sn'$ belongs to all recovery lines having sequence number sn'' such that $sn' < sn'' \leq sn$. So, according to the nomenclature introduced by Netzer and Xu in [12], the algorithm does not produce *useless* checkpoints (i.e., checkpoints that cannot be a part of a recovery line).

5 A Performance Study

5.1 The Simulation Model

The simulation compares BCS (see Section 3.1), MS (see Section 4.1) and the proposed algorithm (BQF described in Subsection 4.3) in an *uniform point-to-point* environment in which each process can send a message to any other process and the destination of each message is an uniformly distributed random variable. We assume a system with $n = 8$ processes, each process executes internal, send and receive operations with probability $p_i = 0.8$, $p_s = 0.1$ and $p_r = 0.1$, respectively. The *time to execute an operation* in a process is exponentially distributed with mean value equal to 1 time units. The time for taking a checkpoint, T_{ckpt} is 10 time units. The *the message propagation time* is exponentially distributed with mean value 10 time units for all the algorithms.

We also consider a *bursted point-to-point environment* in which a process with probability $p_b = 0.1$ enters a burst state and then executes only internal and send events (with probability $p_i = 0.8$, $p_s = 0.2$ respectively) for B checkpoint interval (when $B = 0$ we have the uniform point-to-point environment described above).

Basic checkpoints are taken periodically. Let *bcf* (basic checkpoint frequency) be the percentage of the ratio t/T where t is the time elapsed between two successive periodic checkpoints and T is the total execution time. For example, *bcf*= 100% means that only the initial local checkpoint is a basic one, while *bcf*= 0.1% means that each process takes 1000 basic checkpoints.

We also consider a degree of heterogeneity among processes H. For example, $H = 0\%$ (resp. $H = 100\%$) means all processes have the same checkpoint period $t = 100$ (resp. $t = 10$), $H = 25\%$ (resp. $H = 75\%$) means 25% (resp. 75%) of processes have the checkpoint period $t = 10$ while the remaining 75% (resp. 25%) has a checkpoint period $t = 100$.

A first series of simulation experiments were conducted by varying *bcf* from 0.1% to 100% and we measured (a) the ratio *Tot* between the total number of checkpoints taken by an algorithm and the total number of checkpoints taken by BCS and (b) the average number of checkpoints F forced by each basic checkpoint.

In a second series of experiments we varied the degree of heterogeneity H of the processes and then we measured (c) the ratio E between the total number of checkpoints taken by BQF and MS.

Each simulation run contains 8000 message deliveries and for each value of *bcf* and H, we did

several simulation runs with different seeds and the result were within 4% of each other, thus, variance is not reported in the plots.

5.2 Results of the Experiments

5.2.1 Total Number of Forced Checkpoints

Figure 5 shows the ratio *Tot* of MS and BQF in an uniform point-to-point environment. For small values of *bcf* (below 1.0%), there are only few send and receive events in each checkpoint interval, leading to high probability of equivalence between checkpoints. Thus BQF saves from 2% to 10% of checkpoints compared to MS. As the value of *bcf* is higher than 1.0%, MS and BQF takes the same number of checkpoints as the probability that two checkpoints are equivalent tends to zero. An important point lies in the plot of the average number of forced checkpoints per basic one taken by MS and BQF shown in Figure 7. For small value of *bcf*, BQF induces up to 70% less than MS.

The reduction of the total number of checkpoints and of the ratio F is amplified by the bursted environment (Figure 6 and Figure 8) in which the equivalences between checkpoints on processes running in the burst mode are disseminated to the other processes causing other equivalences. In this case, for all values of *bcf*, BQF saves from a 7% to 18% checkpoints compared to MS, and induces up to 77% less than MS.

5.2.2 Heterogeneous Environment

The low values of F shown by BQF suggested that its performance could be particularly good in a heterogeneous environment in which there are some processes with a shorter checkpointing period. These processes would push higher the sequence number leading to a very high checkpointing overhead using either MS or BCS.

In Figure 9, the ratio E as a function of the degree of heterogeneity H of the system is shown in the case of uniform ($B = 0$) and bursted point-to-point environment ($B = 2$). The best performance (about 30% less checkpointing overhead than MS) are obtained when $H = 12.5\%$ (i.e., when only one process has a checkpoint frequency ten times greater than the others) and $B = 2$.

In Figure 10 we show the ratio *Tot* as a function of *bcf* in the case of $B = 2$ and $H = 12.5\%$ which is the environment where BQF got the maximum gain (see Figure 9). Due to the heterogeneity, *bcf* is in the range between 1% and 10% of the slowest processes. We would like to remark that in all the range the checkpointing overhead of BQF is constantly around 30% less than MS.

5.2.3 Rollback Recovery

We measured the average amount of the undone computation UE, in terms of number of events, (i.e., the rollback distance) after the occurrence of a failure of a process. UE is evaluated without

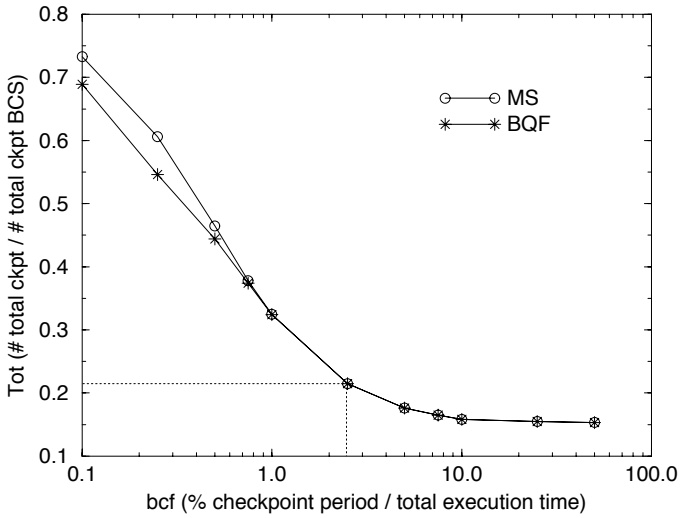

Figure 5: *Tot* vs. *bcf* in the *Uniform Point-to-Point* Environment ($B = 0$).

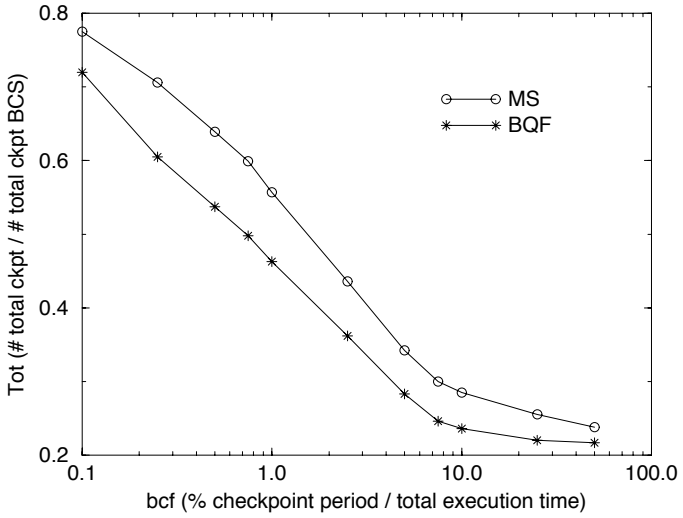

Figure 6: *Tot* vs. *bcf* in the *Bursted Point-to-Point* Environment ($B = 2$).

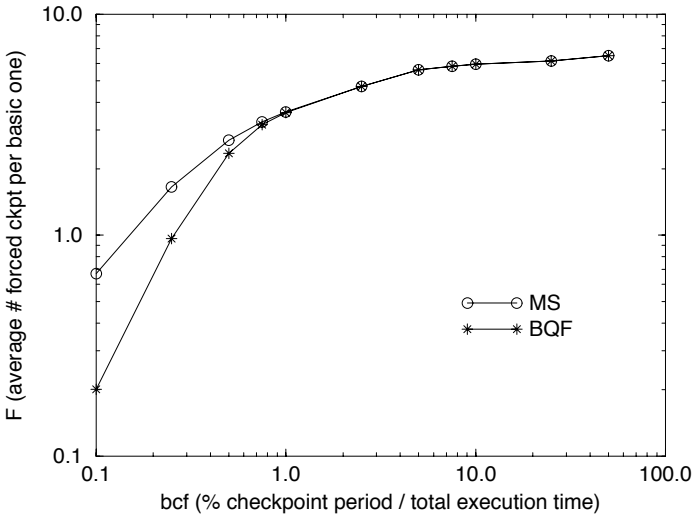

Figure 7: F vs. *bcf* in the *Uniform Point-to-Point* Environment ($B = 0$).

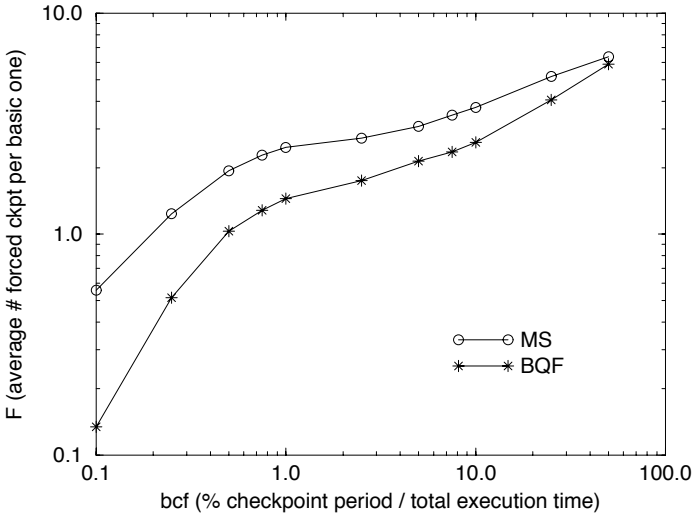

Figure 8: F vs. *bcf* in the *Bursted Point-to-Point* Environment ($B = 2$).

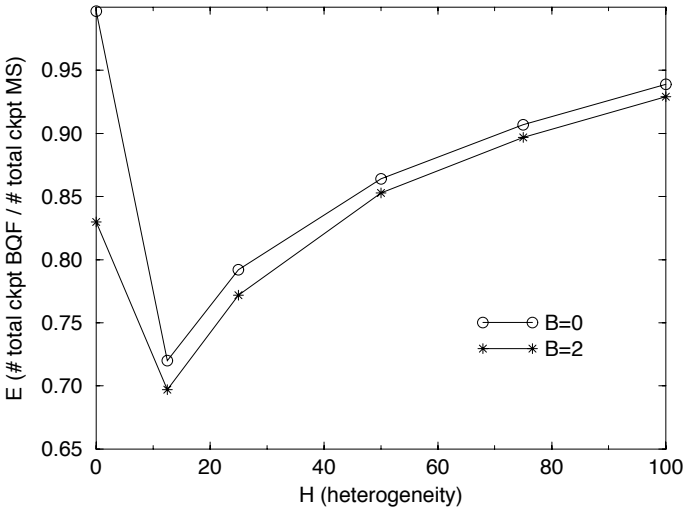

Figure 9: E vs. *Heterogeneity* in both the *Uniform Point-to-Point* Environment ($B = 0$) and the *Bursted Point-to-Point* Environment ($B = 2$).

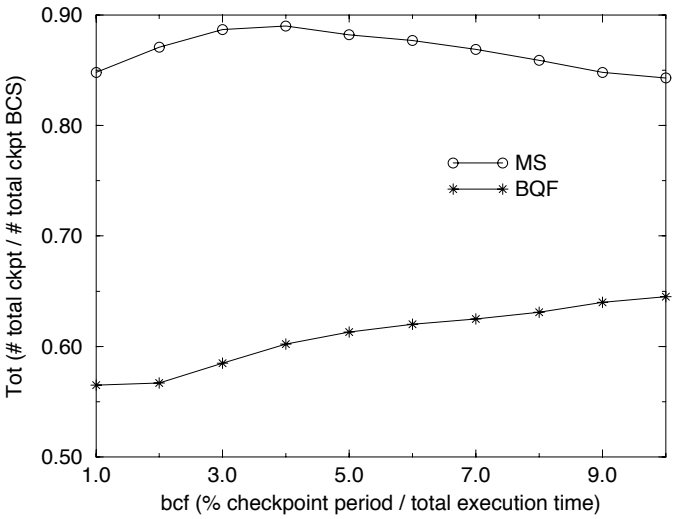

Figure 10: *Tot* vs. *bcf* of the Slowest Processes in a *Bursted Point-to-Point* Environment ($B = 2$) with $H = 12.5\%$.

simulating the rollback phase but considering the amount of undone events as it can be seen by an *omniscient observer* of the system. In particular, each time a process fails, the observer individuates the most recent recovery line of the application and counts the number of events undone to rollback to that recovery line[6].

The closest recovery line to the end of the computation is build as follows: the failed process restarts its computation from its last checkpoint, say A, forcing the other processes to rollback to the recovery line to which A belongs, say $\mathcal{L}_{sn}$.

During the rollback phase, in MS and BCS, if the checkpoint with sequence number sn does not exists a process rolls back to the first checkpoint with sequence number greater than sn, if any, otherwise no rollback action is required for that process.

In BQF, if the index of A is not permanent, the index is replaced with $< sn + 1, 0 >$ and the computation is restarted from the recovery line $\mathcal{L}_{sn+1}$. Otherwise, each process rolls back to the most recent checkpoint with the sequence number sn (i.e., the one with the higher equivalence number). If such a checkpoint does not exists, the process rolls back to the first checkpoint with permanent index $< sn', 0 >$ such that $sn' > sn$.

Simulation experiments were conducted in the uniform point-to-point environment. In Figure 11, UE as a function of bcf is shown. Given the large checkpointing overhead of BCS during failure-free computations (see Figure 5), the recovery line is closest, on the average, to the end of the computation compared to BQF and MS. As an example in the case of $bcf = 2.5\%$ (i.e., 40 basic checkpoints for each process), BQF and MS takes about 80% less forced checkpoints compared to BCS as depicted in Figure 5 while BCS's UE is 70% less than BQF and MS (see Figure 11). This points out an evident tradeoff between UE and the checkpointing overhead in failure free computation.

This behavior is confirmed by plots shown in Figure 12 in an environment whose heterogeneity degree is 12.5% and bcf varies from 1% to 10% of the slowest processes. As an example, if $bcf = 1\%$ then MS's UE is 30% less than BQF while BQF saves about 35% of checkpoints compared to MS (see Figure 10).

5.2.4 Total Overhead Analysis

In this section we introduce a function $OH(N_f)$ which quantifies the total overhead added to the computation by checkpointing and recovery as a function of the number N_f of failures that occur during an execution. We study the behavior of the function OH in BCS, MS and BQF by varying the number of failures of the computation during an execution.

[6]We do not introduce a recovery scheme for our checkpointing algorithm, however, we would like to remark that recovery schemes such as the one presented in [10], can be easily adapted to the BQF algorithm when considering the presence of provisional indices.

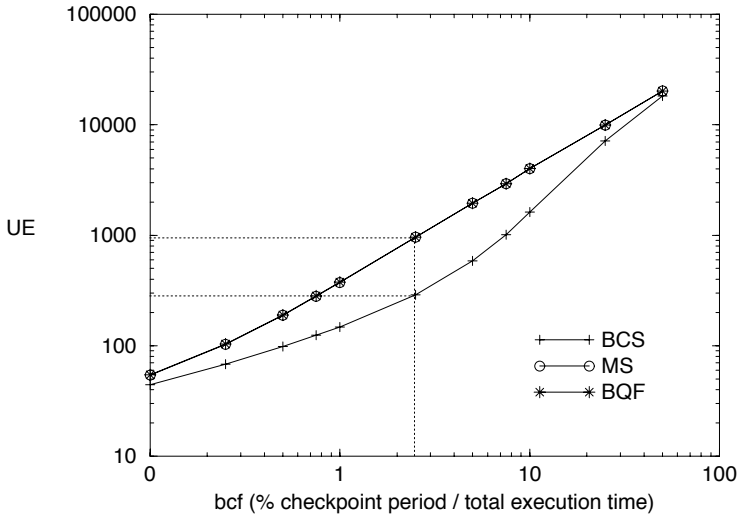

Figure 11: *UE* vs. *bcf* in the *Uniform Point-to-Point* Environment ($B = 0$ and $H = 0\%$).

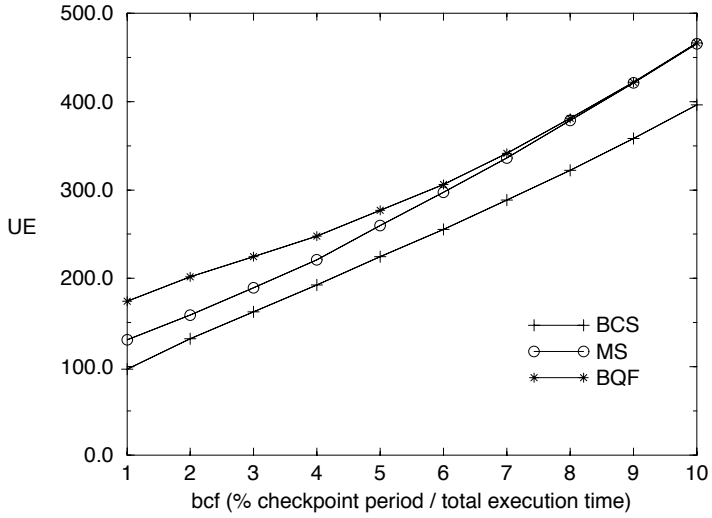

Figure 12: *UE* vs. *bcf* in the *Uniform Point-to-Point* Environment ($B = 0$ and $H = 12.5\%$).

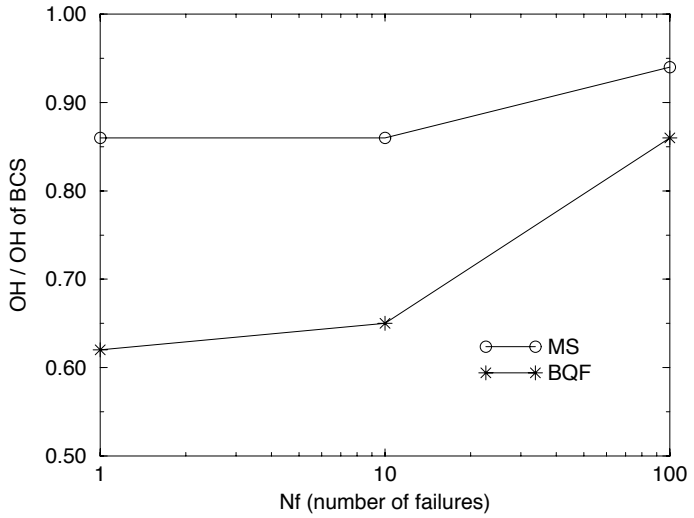

Figure 13: *OH/(OH of BCS)* vs. N_f in the *Uniform Point-to-Point* Environment ($B = 0$ and $H = 12.5\%$).

The total overhead due to checkpointing can be expressed by the product $N_{ckpt}T_{ckpt}$ where N_{ckpt} is the total number of checkpoints taken during a failure free execution and T_{ckpt} is the average time spent in a checkpoint operation.

The average overhead due to a single failure (as it can be seen by the external observer of the system) can be expressed by the sum of two terms. The first term is the product $UC \cdot T_{ckpt}$ where UC is the average number of checkpoints that are undone due to a rollback. The second term is the product $UE \cdot T_{ev}$ where T_{ev} is the average event execution time. We have that the total recovery overhead due to N_f failures is $N_f(UC \cdot T_{ckpt} + UE \cdot T_{ev})$. By combining the checkpointing and the recovery overhead we get:

$$OH(N_f) = N_{ckpt}T_{ckpt} + N_f(UC \cdot T_{ckpt} + UE \cdot T_{ev})$$

Figure 13 shows *OH/OH of BCS* vs. the number of failures imposed during the execution. These plots were obtained in a uniform point-to-point environment with heterogeneity $H = 12.5\%$. A total number of 80000 events were simulated.

The results show that the function OH of BQF is widely less than the one of BCS and MS. The total overhead imposed by the three algorithms becomes comparable only for a very high failure rate (in the order of 10^2 per an execution of 80000 events) which is extremely unlikely in real distributed systems.

6 Conclusion

Among the checkpointing algorithms, the index-based ones ensure the domino-free property to a checkpoint and communication pattern with the smallest number of forced checkpoints. In this paper we presented an index-based checkpointing algorithm, well suited for autonomous distributed systems, that reduces the checkpointing overhead compared to previous index-based solutions. This algorithm lies on an equivalence relation that allows the recovery line to advance without increasing its sequence number.

The algorithm optimistically (and provisionally) assumes that a basic checkpoint C in a process is equivalent to the previous one in the same process by assigning a provisional index. Hence, if at the time of the first send event after C that equivalence is verified, the provisional index becomes permanent. Otherwise the index is increased, as in [2, 10], and this directs forced checkpoints in other processes.

We presented a simulation study which quantifies the saving of checkpoints in different environments compared to previous proposals. The price to pay is each application message piggybacks $n + 1$ integers as control information compared to one integer used by previous algorithms. We also pointed out the effects of the saving of checkpoints on the recovery, and estimated the total overhead due to the checkpointing and the recovery of our algorithm. These results show that the total overhead imposed by our algorithm in an execution is less than that of previous algorithms.

Finally, let us remark that the equivalence relation between checkpoints provides actually a framework that can be used to design efficient checkpoint timestamping mechanisms. Such mechanisms can be embedded in *any* checkpointing algorithm in order to slow down the process of increasing of sequence numbers which is the primary cause of forced checkpoints.

References

[1] R. Baldoni, J.M. Helary, A. Mostefaoui and M. Raynal, A Communication-Induced Checkpointing Protocol that Ensures Rollback-Dependency Trackability, *Proc. IEEE Int. Symposium on Fault Tolerant Computing*, pp. 68-77, 1997.

[2] D. Briatico, A. Ciuffoletti and L. Simoncini, A Distributed Domino-Effect Free Recovery Algorithm, *in Proc. IEEE Int. Symposium on Reliability Distributed Software and Database*, pp. 207-215, 1984.

[3] K.M. Chandy and L. Lamport, Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Trans. on Computer Systems*, vol. 3, no. 1, pp. 63-75, 1985.

[4] F. Cristian and F. Jahanian, A Timestamp-Based Checkpointing Protocol for Long-Lived Distributed Computations, *Proc. IEEE Int. Symposium on Reliable Distributed Systems*, pp. 12-20, 1991.

[5] E.N. Elnozahy, D.B. Johnson and Y.M. Wang, A Survey of Rollback-Recovery Protocols in Message-Passing Systems, *Technical Report No.CMU-CS-96-181, School of Computer Science, Carnegie Mellon University*, 1996.

[6] E.N. Elnozahy and W. Zwaenepoel, Manetho: Transparent Rollback Recovery with Low Overhead, Limited Rollback and Fast Output Commit, *IEEE Trans. on Computers*, vol. 41, no. 5, pp. 526-531, 1992.

[7] R. Koo and S. Toueg, Checkpointing and Rollback-Recovery for Distributed Systems, *IEEE Trans. on Software Engineering*, vol. 13, no. 1, pp. 23-31, 1987.

[8] L. Lamport, Time, Clocks and the Ordering of Events in a Distributed System, *Communications of the ACM*, vol. 21, no. 7, pp. 558-565, 1978.

[9] D. Manivannan, R.H.B. Netzer and M.Singhal, Finding Consistent Global Checkpoints in a Distributed Computation, *IEEE Trans. on Parallel and Distributed Systems*, vol. 8, no. 6, pp. 623-627, 1997.

[10] D. Manivannan and M. Singhal, A Low-Overhead Recovery Technique Using Quasi Synchronous Checkpointing, *Proc. IEEE Int. Conference on Distributed Computing Systems*, pp. 100-107, 1996.

[11] F. Mattern, Virtual Time and Global States of Distributed Systems, *Proc. International Workshop on Parallel and Distributed Algorithms*, pp. 215-226, 1989.

[12] R.H.B. Netzer and J. Xu, Necessary and Sufficient Conditions for Consistent Global Snapshots, *IEEE Trans. on Parallel and Distributed Systems*, vol. 6, no. 2, pp. 165-169, 1995.

[13] B. Randell, System Structure for Software Fault Tolerance, *IEEE Trans. on Software Engineering*, vol. SE1, no. 2, pp. 220-232, 1975.

[14] R.D. Schlichting and F.B. Schneider, Fail-Stop Processors: an Approach to Designing Fault-Tolerant Computing Systems, *ACM Trans. on Computer Systems*, vol. 1, no. 3, pp. 222-238, 1983.

[15] R.E. Strom, D.F. Bacon and S.A.Yemini. Volatile Logging in n-Fault-Tolerant Distributed Systems, *Proc. IEEE Int. Symposium on Fault Tolerant Computing*, pp. 44-49, 1988.

[16] Y.M. Wang, Consistent Global Checkpoints that Contains a Given Set of Local Checkpoints, *IEEE Trans. on Computers*, vol. 46, no. 4, pp. 456-468, 1997.

Authors Biographies

Roberto Baldoni received the laurea in Electronic Engineering in 1990 and the Ph.D degree in Computer Science in 1994 from the University of Rome "La Sapienza". From 1994 to 1995 he holds an appointment as a Computer Science researcher at IRISA/INRIA (France). In 1996 he was visiting assistant professor at the Department of Computer Science of Cornell University. From 1997 to 1998 he was assistant professor in Computer Science at the University of Rome "La Sapienza". Currently he is an associate professor at the same University.

He published more than fifty scientific papers in the fields of fault-tolerant distributed computing and communication protocols. He regularly serves as a referee for many international conferences and journals. He has been invited to serve in the program committees of ICDCS-98 and SRDS-98. He was invited to chair the program committee of the "distributed algorithms" track of the 19th IEEE International Conference on Distributed Computing Systems.

His current research interests include distributed computing, fault-tolerant programming, distributed operating systems, real-time systems, communication protocols and mobile systems.

Francesco Quaglia received the laurea in Electronic Engineering in 1995 from the University of Rome "La Sapienza". Currently he is a Ph.D. student in Computer Engineering at the Dipartimento di Informatica e Sistemistica of the University of Roma "La Sapienza". His research interests include fault-tolerant distributed systems, parallel/distributed simulation and interconnection networks.

Paolo Fornara received the laurea in Electronic Engineering in 1997 from the University of Rome "La Sapienza".