Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XI Ciclo – 1999

Consistent Checkpointing in Distributed Computations: Theoretical Results and Protocols

Francesco Quaglia

Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XI Ciclo - 1999

Francesco Quaglia

Consistent Checkpointing in Distributed Computations: Theoretical Results and Protocols

Thesis Committee		Reviewers
Prof. Bruno Ciciani	(Advisor)	Prof. Jean Michel Hélary
Prof. Giacomo Cioffi		Prof. Mukesh Singhal
Prof. Silvio Salza		

AUTHOR'S ADDRESS:
Francesco Quaglia
Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
E-MAIL: `quaglia@dis.uniroma1.it`
WWW: `http://www.dis.uniroma1.it/~quaglia`

Abstract

This thesis is focused on the study of consistent checkpointing in distributed computations. The model of the computation is asynchronous. The investigated checkpointing approach is known as communication-induced. In this approach, processes of the distributed computation take checkpoints at their own pace (namely basic checkpoints) and some additional checkpoints (namely forced checkpoints) are induced by a lazy coordination scheme, in order to guarantee consistency of global checkpoints. The lazy coordination is realized by piggybacking control information on application messages. Upon the receipt of a message, the recipient process evaluates a predicate basing on the incoming control information and on its local context; if the predicate is evaluated to $TRUE$, a forced checkpoint is taken. The thesis reports both theoretical results on this issue and protocols derived from those results.

Chapter Organization

The first three chapters are devoted to the description of basic concepts and theories on checkpointing.

Original contributions of the thesis starts from Chapter 4 where a taxonomy of communication-induced checkpointing protocols is presented splitting protocols in $\mathcal{VP}$-enforced and $\mathcal{VP}$-accordant ([1]).

Chapter 5 introduces an equivalence relation between checkpoints and presents a $\mathcal{VP}$-enforced communication-induced checkpointing protocol based on such a relation. Its performance is also investigated. The equivalence relation here introduced provides actually a framework that can be used to design efficient checkpoint timestamping mechanisms.

Chapter 6 provides a characterization of the necessary and sufficient condition for the absence of useless checkpoints (i.e., checkpoints that cannot be members of any consistent global checkpoint) in a distributed computation, which was previously an open question. Then, the characterization is used

[1] $\mathcal{VP}$ stands for "Virtual Precedence property".

as a basis for the design of $\mathcal{VP}$-accordant checkpointing protocols ensuring no useless checkpoint. Applications of the proposed protocols are also discussed.

Finally, in Chapter 7, a necessary and sufficient condition for the consistency of global checkpoints of distributed databases is provided by extending results taken from the context of distributed computations. Non-intrusive transaction-induced checkpointing protocols are also presented.

Most of this work can be found in the following papers:

R. Baldoni, F. Quaglia and P. Fornara, An Index-Based Checkpointing Algorithm for Autonomous Distributed Systems, *Proc. 16th IEEE Int. Symposium on Reliable Distributed Systems*, 1997, pp. 27-34 (an expanded version appeared on *IEEE Transactions on Parallel and Distributed Systems*, vol.10, no.2, February 1999).

R. Baldoni, F. Quaglia and B. Ciciani, A $\mathcal{VP}$-Accordant Checkpointing Protocol Preventing Useless Checkpoints, *Proc. 17th IEEE Int. Symposium on Reliable Distributed Systems*, 1998, pp. 61-67.

R. Baldoni, F. Quaglia and M. Raynal, Consistent Checkpointing in Distributed Databases: Towards a Formal Approach, Tech. Rep. 27-97, Dipartimento di Informatica e Sistemistica, Universita' di Roma "La Sapienza", July 1997 (submitted paper).

F. Quaglia, R. Baldoni and B. Ciciani, A Low-Overhead Z-Cycle-Free Checkpointing Algorithm for Distributed Systems, *Proc. European Research Seminar on Advances in Distributed Systems*, 1997, pp. 198-203.

F. Quaglia, B. Ciciani and R. Baldoni, A Checkpointing-Recovery Scheme for Distributed Systems, *in Dimiter R. Avresky, David R. Kaeli, editors, "Fault Tolerant Parallel and Distributed Systems" (Chapter 5)*, Kluwer Academic Publishers, 1998.

F. Quaglia, R. Baldoni and B. Ciciani, On the No-Z-Cycle Property in Distributed Executions, Tech. Rep. 01-99, Dipartimento di Informatica e Sistemistica, Universita' di Roma "La Sapienza", January 1999 (submitted paper).

F. Quaglia, R. Baldoni and B. Ciciani, Characterizing the "No-Z-Cycle" Property in Distributed Computations, submitted paper.

F. Quaglia, B. Ciciani and R. Baldoni, Checkpointing Protocols in Distributed

Systems with Mobile Hosts: a Performance Analysis, *Proc. 3rd Workshop on Fault Tolerant Parallel and Distributed Systems*, LNCS 1388, 1998, pp.742-755.

Acknowledgments

My deepest debt of gratitude goes to my advisor Bruno Ciciani, for his help during the whole Ph.D. course. He suggested me to study problems related to checkpointing. I started to study such problems together with him in the context of optimistic synchronization protocols, then we moved to the context of distributed computations. Bruno always inspired and encouraged me during all troubled times.

I'm also deeply indebted with Roberto Baldoni, with whom most of the results of this thesis have been obtained. He showed me how to tackle the checkpointing problem from a theoretical point of view, and also how theoretical results can provide a basis for protocols with practical impact.

I would like to thank Michel Raynal, with whom results reported in Chapter 7 have been obtained. It was a honour to discuss with him issues related to checkpointing.

I express my gratitude to Jean Michel Hélary for comments he gave me on a preliminary version of a technical report dealing with the characterization introduced in Chapter 6, and for his help in the construction of the proof of Theorem 7.4.3 in Chapter 7. I thank him also for having accepted to be an external referee for this thesis.

I would like to thank Ravi Prakash, who gave me interesting criticisms on the characterization theorem.

Special thank goes to Paolo Fornara and Luca De Santis. They helped me to develop simulation code and to collect simulation data reported in Chapter 5 and in Chapter 6.

I also thank Giacomo Cioffi and Silvio Salza for being internal referees, Prof. Mukesh Singhal as external referee, Luigia Carlucci Aiello and Giorgio Ausiello as presidents of the Ph.D. Committee of the Dipartimento di Informatica e Sistemistica.

Finally, I thank my parents Angela and Gennaro, my sisters Maria Paola and Rossella, and my girlfriend Cristina, for strength and invaluable help they give me.

Francesco

Contents

Chapter 1

Introduction

A global state of a distributed computation is a set of individual process states, also called *local states*, one for each process. Each local state represents a snapshot of the process at a given point of the computation. A local checkpoint, or simply checkpoint, is a local state saved onto stable storage. A set of checkpoints, one for each process, is a *global checkpoint* of the distributed computation.

Global checkpoints have application in several problems of distributed computing such as hardware/software fault tolerance [19, 49], distributed debugging [16, 23], the determination of distributed breakpoints [22, 39] and of shared global states [24], the evaluation of global predicates [34], protocol specification [25] and others [63, 64]. However, the application of global checkpoints to previous problems may result ineffective, or even useless, if the problem of *consistency* is not tackled.

A global checkpoint is consistent if no checkpoint in the global checkpoint depends on another one. Informally, consistency means that there does not exist any message whose receive event is recorded in the global checkpoint whereas the corresponding send event is not recorded. If this happens, the global checkpoint represents a snapshot of the computation recording a dependence which is not yet generated. Such a dependence is the source of the inconsistency of the global checkpoint.

As an example of drawbacks due to inconsistency, in the context of fault tolerance through checkpoint-based rollback, the absence of consistent global checkpoints requires, in case of failure, the distributed computation to be restarted from its initial state (this unbounded rollback extent is known as *domino effect* [49]). This is a highly undesirable phenomenon implying that all checkpoints taken (i.e., the overhead imposed to the computation for taking them) result to be useless for protecting against the loss of all useful work performed until the occurrence of the failure.

Depending on the way checkpoints are taken by processes, checkpointing protocols can be split into three classes: *uncoordinated*, *coordinated* and *communication-induced*.

This thesis focuses on the communication-induced class and considers a particular model of the distributed computation usually termed in the literature as *asynchronous* model with non-FIFO communication between processes. Such a model is presented in Section 1.1 of this chapter. Both theoretical and practical aspects of communication-induced checkpointing are investigated. Starting from theoretical results, communication-induced checkpointing protocols are derived with the aim at improving system performance compared to previous solutions.

Note that when a checkpointing protocol runs at processes, the outcoming distributed computation is modeled not only as a partial order of events, but also as a set of relations among checkpoints. Thus, in Section 1.2 of this chapter, the notion of *checkpoint and communication pattern* of a distributed computation is presented. Then, in Section 1.3 the formal definition of consistent global checkpoint is provided.

Although this thesis presents results for communication-induced checkpointing, Section 1.4 of this chapter is devoted to the description of features of protocols in each class in order to outline basic differences among distinct classes. We give details while describing protocols of the uncoordinated and coordinated class; instead, less details are given about protocols of the communication-induced class as they will be extensively described in Chapter 3. As it will be shown, some of the coordinated checkpointing protocols require more strict constraints on the computational model investigated in this thesis (for example FIFO communication). Whenever one of these protocols is described, the imposed constraints are explicitly mentioned.

A set of preliminary definitions and notations forming a basis for any future reasoning or description are also presented somewhere in this chapter. Additional definitions/notations are introduced whenever they are needed.

1.1 Model of the Distributed Computation

A distributed computation consists of a set P of n processes $\{P_1, P_2, \ldots, P_n\}$. Processes do not share memory and do not share a common clock value; furthermore, no private information of any process (such as clock drift, clock granularity or clock precision) is known by other processes. They communicate only by exchanging messages. Each pair of processes is connected by an asynchronous, directed logical channel. Transmission delays over channels are unpredictable but finite.

Processes of the distributed computation are *sequential*. A process produces a sequence of *events*; each event moves the process from one local state to another. The x-th event in process P_i is denoted as $e_{i,x}$. We assume events are produced by the execution of internal, send and receive statements. The send and receive events of a message m are denoted respectively by $send(m)$ and $receive(m)$.

Definition 1.1.1
In process P_i an event $e_{i,x}$ precedes an event $e_{i,y}$, denoted $e_{i,x} \prec_P e_{i,y}$, iff $x < y$.

Definition 1.1.2
An event $e_{i,x}$ of process P_i precedes an event $e_{j,y}$ of process P_j due to message m, denoted $e_{i,x} \prec_m e_{j,y}$, iff:

$$(e_{i,x} = send(m)) \wedge (e_{j,y} = receive(m))$$

Lamport's *Happened-Before* relation [33], denoted as $\stackrel{e}{\rightarrow}$, is the transitive closure of the union of relations $\prec_P$ and $\prec_m$. Let $\mathcal{H}$ be the set of all events produced by a distributed computation, the computation can be modeled by the partial order $\widehat{\mathcal{H}} = (\mathcal{H}, \stackrel{e}{\rightarrow})$. The relation $\stackrel{e}{\rightarrow}$ expresses causal dependences between events. If $e_{i,x} \stackrel{e}{\rightarrow} e_{j,y}$, then $e_{j,y}$ is causally dependent on $e_{i,x}$.

Let us now introduce some graphical notations. In any picture, horizontal lines extending towards the right end side represent process execution; arrows between processes represent messages. As an example, in Figure 1.1 we have a computation consisting of three processes and two messages. Process P_2 sends a message m to P_1 and then receives message m' sent by P_3.

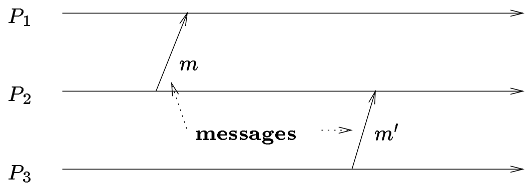


Figure 1.1: An Example of Distributed Computation.

1.2 Checkpoint and Communication Patterns of Distributed Computations

A local state of a process saved on stable storage is called a *checkpoint* of the process. A local state is not necessarily recorded as a local checkpoint, so the set of local checkpoints is a subset of the set of local states.

The x-th checkpoint of process P_i is denoted as $C_{i,x}$ where x is called the *rank* of the checkpoint. The rank of checkpoints of a process increases monotonically: each time a checkpoint is taken the rank is increased by one. It is assumed that each process P_i takes an initial checkpoint $C_{i,1}$ (corresponding to the initial state of the process) and that after each event a checkpoint will eventually be taken. Hence the execution of a process always terminates with a checkpoint. A checkpoint interval $I_{i,x}$ is the set of events between $C_{i,x}$ and $C_{i,x+1}$.

Let us finally introduce the concept of checkpoint and communication pattern related to a distributed computation:

Definition 1.2.1
A checkpoint and communication pattern of a distributed computation is a pair $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ *where* $\widehat{\mathcal{H}}$ *is a distributed computation and* $\mathcal{C}_{\widehat{\mathcal{H}}}$ *is a set of local checkpoints defined on* $\widehat{\mathcal{H}}$.

From a graphical point of view, the action of taking a checkpoint at a given point of the execution is pictured as a rectangular box placed on the line representing the process execution. As an example, in Figure 1.2 we have a computation with three processes and four checkpoints. Checkpoints $C_{1,1}$, $C_{2,1}$ and $C_{3,1}$ correspond to the initial states of the processes. The checkpoint interval $I_{2,1}$ (corresponding to events occurring in P_2 between $C_{2,1}$ and $C_{2,2}$) is marked in the picture. Note that the termination of the computation is not shown (otherwise a checkpoint should be placed at the end of each horizontal line).

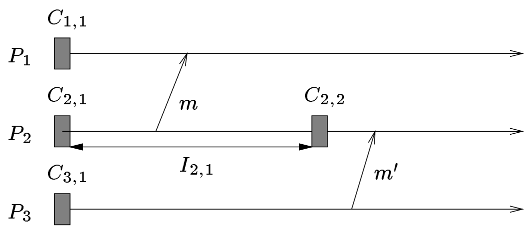


Figure 1.2: An Example of Distributed Computation with Checkpoints.

1.3 Consistent Global Checkpoints

A global checkpoint of a distributed computation is a set of local checkpoints $\{C_{1,x_1}, \ldots, C_{n,x_n}\}$, one for each process. The notion of consistent global checkpoint [14] can be easily formalized by using the following precedence relation between checkpoints:

Definition 1.3.1
A checkpoint $C_{i,x}$ of process P_i precedes checkpoint $C_{j,y}$ of process P_j, denoted $C_{i,x} \prec_{ckpt} C_{j,y}$, if there exists a message m such that:

$$((send(m) \in I_{i,x'}) \wedge (x' \geq x)) \wedge ((receive(m) \in I_{j,y'}) \wedge (y' < y))$$

In other words, Definition 1.3.1 states that $C_{i,x}$ precedes $C_{j,y}$ if there exists a message m which is sent by P_i after $C_{i,x}$ was taken and is received by P_j before taking $C_{j,y}$. In the literature, such a message is said to be *orphan* with respect to the ordered pair $(C_{i,x}, C_{j,y})$ [14]. As an example, in Figure 1.3 a computation with two processes and an orphan message m with respect to the ordered pair $(C_{1,1}, C_{2,2})$ is shown. Due to m, checkpoint $C_{1,1}$ precedes $C_{2,2}$ through the $\prec_{ckpt}$ relation.

Definition 1.3.2
A global checkpoint $\{C_{1,x_1}, \ldots, C_{n,x_n}\}$ is consistent iff for any pair of checkpoints (C_{i,x_i}, C_{j,x_j}) in it:

$$(\neg(C_{i,x_i} \prec_{ckpt} C_{j,x_j})) \wedge (\neg(C_{j,x_j} \prec_{ckpt} C_{i,x_i}))$$

Intuitively, from Definition 1.3.2 a global checkpoint is consistent if for any message m whose receive event is recorded in the global checkpoint then also the corresponding send event is recorded in the global checkpoint. As an example, the global checkpoint $\{C_{1,1}, C_{2,2}\}$ in Figure 1.3 is not consistent because the send event of message m is not recorded in it. In the context of rollback recovery based on checkpointing, the inconsistency of the ordered pair $(C_{1,1}, C_{2,2})$ means that, in case of failure, the application cannot be rolled back to the global checkpoint $\{C_{1,1}, C_{2,2}\}$. If P_1 rolls back to $C_{1,1}$ then it undoes all the events produced after taking that checkpoint, including the send event of message m. If P_2 rolls back to $C_{2,2}$ then the receive of m is not undone. In such a case, there exists a message which is not sent but has been received, hence the global checkpoint records a causal dependence between P_1 and P_2 which is not yet generated.

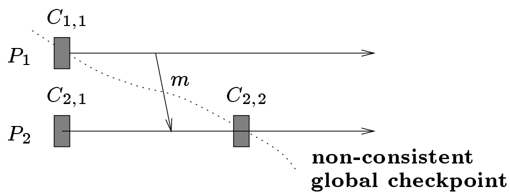


Figure 1.3: An Example of Precedence Between Checkpoints.

1.4 Checkpointing Protocols

In this section a description is given of checkpointing protocols in the uncoordinated and coordinated classes. Furthermore, basic concepts about communication-induced checkpointing are also presented (communication-induced protocols are, instead, extensively discussed in Chapter 3).

1.4.1 Uncoordinated Protocols

Uncoordinated (or independent) checkpointing protocols allow each process to decide independently when to take checkpoints. The main advantage is the low overhead imposed to the computation because no coordination among processes is necessary. Autonomy in taking checkpoints also allows each process to select appropriate checkpoint positions in order to further reduce the overhead: (i) by saving smaller amounts of state information (this may happen in the case of processes having dynamic state size) or (ii) by checkpointing during idle CPU periods.

The main disadvantage of the uncoordinated approach is the possibility that no consistent global checkpoint can ever be formed. As already outlined, this can lead to an unbounded rollback extent in the case of fault tolerance realized through checkpoint-based rollback.

The dependences between checkpoints caused by message exchanges need to be recorded in order to reconstruct a consistent global checkpoint whenever it is reclaimed. To this purpose, a direct dependency tracking technique [9, 57, 59] is commonly adopted. It works as follows: whenever a process P_i executing at its checkpoint interval $I_{i,x}$ sends a message m to P_j, the pair (i,x) is piggybacked on m. If P_j receives m in its checkpoint interval $I_{j,y}$, the dependence between $C_{i,x}$ and $C_{j,y+1}$ is recorded when $C_{j,y+1}$ is taken. Whenever a consistent global checkpoint is reclaimed by a process P_k, the latter broadcasts a *dependency_request* message for collecting the dependency information from the other processes. Upon the receipt of the message, process P_h replies to P_k with the dependency information. The consistent global checkpoint is then calculated by P_k basing on the collected information. Such a calculation is realized building and analyzing either a rollback-dependency-graph [9, 13, 63] or a checkpoint-graph [57, 62].

Basically, in a rollback-dependency-graph each node corresponds to a checkpoint and an edge exists between $C_{i,x}$ to $C_{j,y}$ if: (1) $i \neq j$ and a message m is sent by P_i in the checkpoint interval $I_{i,x-1}$ and is received by P_j in $I_{j,y-1}$, or (2) $i = j$ and $y = x+1$. The name rollback-dependency-graph comes from the context of fault tolerance and indicates that if there exists an edge between $C_{i,x}$ and $C_{j,y}$, and the interval $I_{i,x-1}$ is rolled back on P_i, then the interval $I_{j,y-1}$ must be rolled back as well (because $C_{j,y}$ depends on $C_{i,x-1}$). To calculate a consistent global checkpoint containing $C_{i,x}$ the following algorithm is used [9, 63]: the node corresponding to $C_{i,x+1}$ is marked; then all the nodes reachable by the initially marked node are marked as well (i.e., a reachability analysis is performed on the graph); the last unmarked node for each process corresponds to a checkpoint which is a member of the consistent global checkpoint. Note that it is not guaranteed that the identified global checkpoint actually contains $C_{i,x}$ (i.e., the corresponding node could be marked during the analysis).

The checkpoint-graph is quite similar to the rollback-dependency graph, with the difference that an edge exists between nodes corresponding to $C_{i,x}$ and $C_{j,y}$ if there exists a message m which is sent in $I_{i,x}$ and is received in $I_{j,y}$. Also in this case, reachability analysis is used for identifying consistent global checkpoints [57, 60].

1.4.2 Coordinated Protocols

In coordinated checkpointing protocols, processes coordinate their checkpointing actions in order to ensure consistency of a global checkpoint. In the context of checkpoint-based rollback recovery, coordinated protocols allow computations which are free from the domino effect as, after the occurrence of a failure, the computation can be always resumed from the last taken global checkpoint (being it consistent). The main disadvantages are: (i) the sacrifice of process autonomy and (ii) the message overhead due to the coordination.

A simple approach to coordinate checkpointing actions is to block interprocess communication until the end of the execution of the checkpointing protocol [18, 55]. This can be done through a simple two-phase based protocol structured as follows. The initiator process broadcasts a *checkpoint_request* message and takes its checkpoint; upon the receipt of that message, any process, other than the initiator, takes a checkpoint, stops sending application messages and replies to the initiator with a *local_checkpoint_done* message. After having received the *local_checkpoint_done* message from all the other processes, the coordinator starts the second phase by broadcasting a *global_checkpoint_done* message. Upon the receipt of the latter message, any process resumes normal execution.

An alternative to the blocking technique is non-blocking coordination. In this type of coordination, processes other than the initiator do not block sending application messages when the *checkpoint_request* message is received. The problem incurred is that a process P_j can receive an application message m sent by P_i after the latter received the *checkpoint_request* message from the initiator. Such a situation is depicted in Figure 1.4.a. If P_j receives and processes the message m before the receipt and the processing of the *checkpoint_request* message then checkpoints $C_{i,x}$ and $C_{j,y}$ cannot be part of a consistent global checkpoint due to the presence of m which establishes the following relation $C_{i,x} \prec_{ckpt} C_{j,y}$. As a result, the outcoming global checkpoint is not consistent. In the case of FIFO communication channels, Chandy and Lamport provide a solution to this problem [14] by forcing process P_i to send a *checkpoint_request* message to P_j before the sending of m and imposing to each process to take a checkpoint upon the receipt of the first *checkpoint_request* message. In such a case process P_j takes the checkpoint before the receipt of m (see Figure 1.4.b), thus avoiding inconsistency of the global checkpoint. A modification of the Chandy-Lamport scheme for the case of non-FIFO communication channels has been presented in [32]. Such solution avoids the sending of the *checkpoint_request* message from P_i to P_j, instead, the checkpoint request is piggybacked on m. Upon the receipt of m piggybacking the request, a checkpoint is taken by P_j before processing the message. In the case of non-FIFO communication channels, it is possible that a checkpoint request with index ind_1 is received when a checkpoint request with index $ind_2 > ind_1$ was already processed. In such a case the checkpoint request with index ind_1 is discarded.

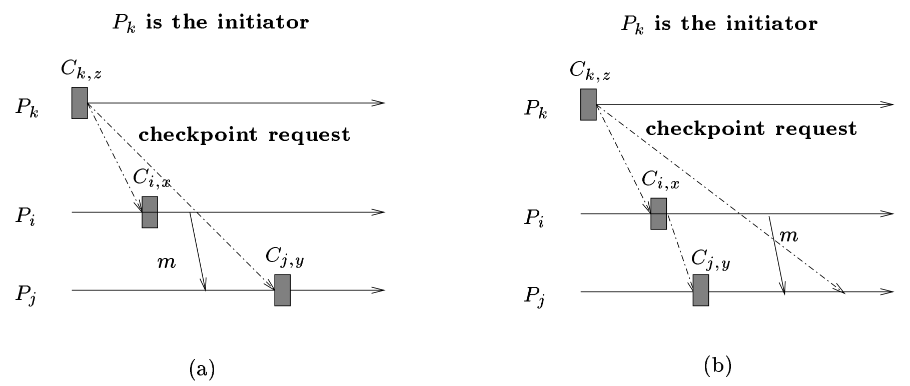


Figure 1.4: A Global Checkpoint which is not Consistent (a); a Consistent Global Checkpoint (b).

A way to reduce the impact of coordination on the execution is to force coordination itself only among processes that really need to coordinate (i.e., processes that have communicated with the initiator since the last taken checkpoint) [8, 31]. In the scheme presented by Koo and Tueg [31] a two-phase approach is adopted with the following characteristics. In the first phase the initiator identifies all the processes that communicated with it since its last checkpoint and sends a *checkpoint_request* message to all of them. Upon the receipt of that message, a process behaves in a similar way (i.e., it identifies its set of communicating processes since the last checkpoint and sends them the request). When all processes are identified, the second phase is started, in which the checkpointing actions are performed. This scheme requires blocking coordination.

A rather different way to reduce the overhead due to coordination messages is the usage of synchronous, or *quasi* synchronous, checkpointing clocks [17, 48, 56]. Note that synchronous clocks imply the computational model to be more strict than the general one described in Section 1.1. Furthermore, an additional restriction is that checkpoints can be triggered only on a periodic basis. If processes take local checkpoints approximately at the same time then the need for broadcasting a checkpoint request message is avoided. To guarantee consistency in the presence of drift between clocks, either the sending of messages is blocked for a given amount of time (related to the maximum deviation between clocks) or checkpoint requests are piggybacked on application messages. In the latter case, if upon the receipt of a message m piggybacking the request the recipient process has not yet taken the checkpoint (due to drift between clocks) then it takes the checkpoint prior to processing the message.

1.4.3 Communication-Induced Protocols

In communication-induced checkpointing, the coordination between checkpointing actions at distinct processes is realized in a lazy fashion by piggybacking control information on application messages. Upon the receipt of an application message, the recipient process examines the information prior to processing the message. If a given predicate $\mathcal{P}$ is evaluated to $TRUE$ then a checkpoint is taken before processing the message. Such a checkpoint is called *forced* checkpoint. Protocols in this class differ by the amount of control information piggybacked on the application messages and by the predicate $\mathcal{P}$ triggering checkpoints upon the receipt of a message. Note that the control information incoming with application messages is commonly used to update local control information, namely local context, proper of the recipient process.

In this kind of approach we can distinguish between two types of checkpoints: (i) basic checkpoints, that are taken by a process according to its own local strategy (an example of local strategy is periodic checkpointing), and (ii) forced checkpoints, which are triggered by the lazy coordination scheme.

In contrast with coordinated checkpointing, no coordination message is exchanged among processes, hence the only information available to the checkpointing protocol at the receive event of an application message is the one encoded by the control information piggybacked on that message plus the local context of the process. This information is related to the causal past of that event which is captured by the $\stackrel{e}{\rightarrow}$ relation. The following constraints commonly identify the communication-induced class:

C1. The usable knowledge at an event e is the knowledge of the restriction of

$(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ to e's *causal past*;

C2. Upon the arrival of a message m at process P_i, the checkpointing protocol has to evaluate the predicate $\mathcal{P}$ *on-the-fly* (i.e., without additional delays). If it is evaluated to $TRUE$, a forced checkpoint has to be taken before processing m;

C3. The evaluation of the predicate is based on the usable knowledge available at that event (i.e., the local context of the process plus the control information piggybacked on the application message). In other words, no control message is allowed;

C4. The content of an application message cannot be interpreted by the checkpointing protocol;

C5. Information about other processes (such as clock speed, clock drift, etc.) and about the network's characteristics (such as the maximum message transmission delay) are not known by any process.

The structure of the predicate $\mathcal{P}$ determines the property ensured by the outcoming checkpoint and communication pattern of the distributed computation. Two main properties are of interest for most applications: the no-Z-cycle property and the rollback-dependency-trackability property. Informally, the no-Z-cycle property stipulates that each local checkpoint belongs to at least one consistent global checkpoint. Both previous properties will be presented in Chapter 2. Finally, an overview of communication-induced protocols ensuring either one or the other property will be reported in Chapter 3.

Chapter 2

Consistent Checkpointing

The definition of consistency of a global checkpoint relies on the notion of causality, as consistency means that no checkpoint in the global checkpoint depends on another checkpoint in the global checkpoint through the $\prec_{ckpt}$ relation - see Definition 1.3.2 - (such a relation captures causal dependences between checkpoints due to the exchange of a single message). However, reasoning by causality has been for long time the major cause preventing the answers to the following fundamental questions:

- $\mathcal{Q}(C_{i,x}, C_{j,y})$: given a pair $(C_{i,x}, C_{j,y})$ of checkpoints of distinct processes, which is the necessary and sufficient condition for these checkpoints to be members of a same consistent global checkpoint?
- $\mathcal{Q}(C_{i,x})$: given a checkpoint $C_{i,x}$ of process P_i, which is the necessary and sufficient condition ensuring that checkpoint $C_{i,x}$ can be member of at least one consistent global checkpoint?

The precedence relation $\prec_{ckpt}$ between checkpoints, and, more generally, the concept of causality have been shown by Netzer and Xu to be not enough powerful to form a basis for providing answers to previous questions. Netzer and Xu provided those answers in a recent past [40] by starting from a notion of dependence superseding the causal one. Actually they provided the answer to the following question $\mathcal{Q}(S)$, which includes both $\mathcal{Q}(C_{i,x}, C_{j,y})$ and $\mathcal{Q}(C_{i,x})$:

- $\mathcal{Q}(S)$: given a set S of checkpoints of distinct processes, including at least one checkpoint and at most one checkpoint for each process, which is the necessary and sufficient condition for these checkpoints to be members of a same consistent global checkpoint?

Their results, which are described in this chapter, are of interest not only from a theoretical point of view but also from a practical one as they gave a strong shot to research in the field of design of communication-induced checkpointing protocols.

2.1 Netzer and Xu Theory

2.1.1 Z-Paths

Netzer and Xu generalized the notion of causal dependence through the introduction of the concept of *zigzag path* (*Z-path* for short) [40]. Z-paths are particular kind of dependences between checkpoints which include both causality and non-causality.

Informally, a Z-path between a checkpoint $C_{i,x}$ and a checkpoint $C_{j,y}$ is a particular sequence of messages $[m_1, \ldots, m_q]$ such that the sending of a message m_i belongs on a process to the same, or to a successive, checkpoint interval of the receive of the message m_{i-1}. Formally, a Z-path from $C_{i,x}$ to $C_{j,y}$ is defined as follows:

Definition 2.1.1
A Z-path exists from checkpoint $C_{i,x}$ to checkpoint $C_{j,y}$ iff there exists a sequence of messages $[m_1, m_2, \ldots, m_q]$ such that:

(1) $(send(m_1) \in I_{i,x'}) \wedge (x' \geq x)$
(i.e., m_1 is sent by process P_i after taking $C_{i,x}$);

(2) $\forall p \ : \ 1 \leq p < q \Rightarrow$ *if* $receive(m_p) \in I_{k,z}$ *then* $(send(m_{p+1}) \in I_{k,z'}) \wedge (z' \geq z)$
(i.e., if m_p $(1 \leq p < q)$ is received by process P_k in the checkpoint interval $I_{k,z}$, then m_{p+1} is sent by P_k in the same or in a later checkpoint interval, although m_{p+1} may be sent before or after m_p is received);

(3) $(receive(m_q) \in I_{j,y'}) \wedge (y' < y)$
(i.e., m_q is received by process P_j before taking $C_{j,y}$).

Figure 2.1.a and Figure 2.1.b show two examples of Z-path between $C_{i,x}$ and $C_{j,y}$ formed by messages $[m_1, m_2]$.

The following fundamental theorem has been proved by Netzer and Xu [40]:

Theorem 2.1.1
A set of checkpoints S, where each is from a different process, can belong to the same consistent global checkpoint iff no checkpoint in S has a Z-path to any checkpoint in S.

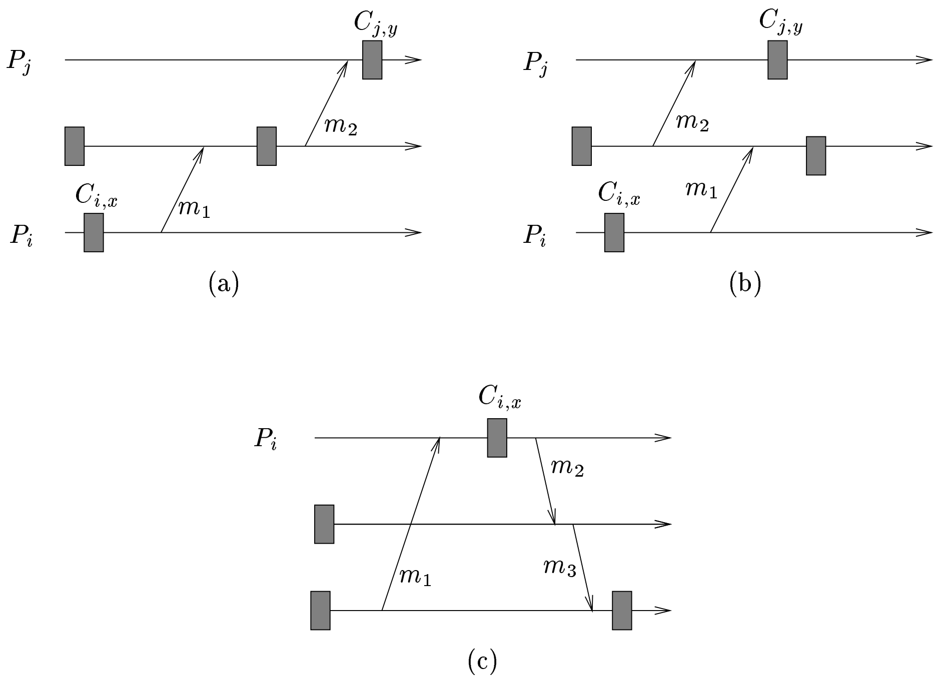


Figure 2.1: A causal Z-path from $C_{i,x}$ to $C_{j,y}$ (a), a non-causal Z-path from $C_{i,x}$ to $C_{j,y}$ (b), a Z-cycle involving $C_{i,x}$ (c).

Basing on Theorem 2.1.1, we have that, though $C_{i,x}$ and $C_{j,y}$ in Figure 2.1.a and in Figure 2.1.b do not depend on each other through the relation $\prec_{ckpt}$, they cannot be members of a same consistent global checkpoint as there exists a Z-path between them.

Z-paths can be split in two families: the *causal* Z-paths which are actually casual paths of messages and the *non-causal* Z-paths in which there exists at least one message m_p whose send precedes the receive of m_{p-1} in the same checkpoint interval. Formally:

Definition 2.1.2
A Z-path from $C_{i,x}$ to $C_{j,y}$ formed by messages $[m_1, \ldots, m_q]$ is causal if

$$\forall p \;:\; 1 \leq p < q \Rightarrow \mathit{receive}(m_p) \prec_P \mathit{send}(m_{p+1})$$

Otherwise the Z-path is non-causal.

As an example, the Z-path from the checkpoint $C_{i,x}$ to $C_{j,y}$ formed by $[m_1, m_2]$ shown in Figure 2.1.a is a causal one. Instead, the Z-path from $C_{i,x}$ to $C_{j,y}$ formed by $[m_1, m_2]$ shown in Figure 2.1.b is a non-causal one.

2.1.2 Z-Cycles

Due to the presence of non-causal Z-paths, it is possible for a sequence of messages to establish a relation between a checkpoint $C_{i,x}$ and itself. Such a relation has been formalized by Netzer and Xu with the name *zigzag cycle* (*Z-cycle* for short) [40]. Therefore, a Z-cycle involving $C_{i,x}$ is a Z-path from $C_{i,x}$ to itself.

As an example, the sequence of messages $[m_2, m_3, m_1]$ shown in Figure 2.1.c involves checkpoint $C_{i,x}$ in a Z-cycle. Using the notion of Z-cycle, the following fundamental Corollary has been derived from Theorem 2.1.1:

Corollary 2.1.2
A checkpoint $C_{i,x}$ of process P_i can belong to at least one consistent global checkpoint iff $C_{i,x}$ is involved in no Z-cycle.

2.2 Properties of Checkpoint and Communication Patterns

Starting from the notions of Z-path and Z-cycle, two fundamental properties of checkpoint and communication patterns of distributed computations have been studied. These properties are described in this section.

2.2.1 The No-Z-Cycle Property

Given a checkpoint $C_{i,x}$ belonging to a checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ of a distributed computation, then, by Corollary 2.1.2, $C_{i,x}$ can be part of at least one consistent global checkpoint iff it is involved in no Z-cycle. If the property of being involved in no Z-cycle holds for any checkpoint in $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$, then $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ is said to satisfy the No-Z-Cycle ($\mathcal{NZC}$) property. More formally:

Property 2.2.1
A checkpoint and communication pattern of a distributed computation $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ satisfies the No-Z-Cycle property ($\mathcal{NZC}$) iff no Z-cycle exists in $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$.

$\mathcal{NZC}$ is a highly desirable property in the context of many applications. In particular, in a checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ of a distributed computation satisfying $\mathcal{NZC}$ the progress of the global consistent checkpoint is guaranteed (because each time a local checkpoint is taken then there exists at least a global consistent checkpoint including it). In the context of rollback recovery, ensuring the $\mathcal{NZC}$ property means rollback without the risk of the domino-effect.

2.2.2 The Rollback-Dependency-Trackability Property

Sometimes applications relying on checkpointing also involve other problems. For example, rollback recovery involves problems as *recovery line identification*, *garbage collection* and *output commit*. Identifying a recovery line of a distributed computation means determining the global consistent checkpoint more close to the end of the computation (the computation is then rolled back to that recovery line in order to minimize the amount of lost work). All the checkpoints preceding the recovery line can be garbage collected for recovering storage. Instead, the output commit problem appears whenever there exist interactions with external entities (for example an external client) which cannot be required to rollback.

Efficient solutions to previous problems can be found in a simple way if processes can calculate efficiently the minimum and maximum consistent global checkpoint containing a given local checkpoint([1]). For example, the efficient calculation of the maximum consistent global checkpoint is the basis for an efficient rollback minimizing the amount of lost work. Also, the efficient calculation of the minimum consistent global checkpoint recording all the outputs is the basis for efficient solutions to the output commit problem.

Wang has shown [64] that if all dependences between checkpoints due to Z-paths are *trackable* on-the-fly (i.e., at the time a checkpoint is taken) then the individuation of the minimum and maximum consistent global checkpoints associated to a specified set of checkpoints is quite straightforward. Dependences between checkpoints due to Z-paths are trackable iff they can be revealed by causality.

A dependence between two checkpoints $C_{i,x}$ and $C_{j,y}$ due to a non-causal Z-path from $C_{i,x}$ to $C_{j,y}$ cannot be revealed by causality. An example of this type of dependence is the one between $C_{i,x}$ and $C_{j,y}$ due to messages $[m_1, m_2]$ in Figure 2.1.b. However, if given a dependence due to a non-causal Z-path, the same dependence is also established by a causal Z-path then such a dependence can be tracked on-the-fly. Whenever a dependence between checkpoints established by a non-causal Z-path is also established by a causal one, then the original Z-path is said to be *causally doubled*.

As an example, in Figure 2.2 the dependence between $C_{i,x}$ and $C_{j,y}$ due to the non-causal Z-path formed by $[m_1, m_2]$ is also established by the causal Z-path formed by $[m_1, m_3]$, hence process P_j is able to track such dependence involving $C_{j,y}$ on-the-fly by exploiting causality (i.e., the non-causal Z-path is causally doubled).

The ability for each process to track on-the-fly all dependences due to Z-paths and involving its checkpoints, deriving from the fact that all non-causal

[1]The minimum (resp. maximum) consistent global checkpoint containing $C_{i,x}$ corresponds to the earliest (resp. latest) consistent global checkpoint containing $C_{i,x}$ [35, 63, 64].

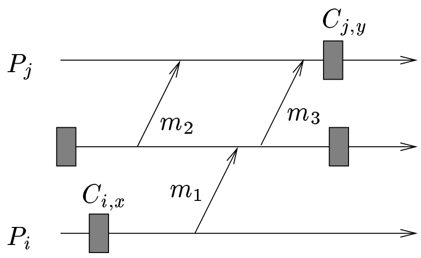


Figure 2.2: A non-Causal Z-path from $C_{i,x}$ to $C_{j,y}$ Formed by $[m_1, m_2]$ which is Causally Doubled by the Causal Z-path Formed by $[m_1, m_3]$.

Z-paths are causally doubled, is a property of the checkpoint and communication pattern of the distributed computation known as Rollback-Dependency-Trackability ($\mathcal{RDT}$) [3, 64]. Formally:

Property 2.2.2
A checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ of a distributed computation satisfies the Rollback-Dependency-Trackability property ($\mathcal{RDT}$) iff all its Z-paths are causally doubled.

As shown by Wang [64], in a checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ satisfying the $\mathcal{RDT}$ property all dependences due to Z-paths can be tracked on-the-fly by a transitive dependency tracking mechanism. Details about such a mechanism will be discussed in Chapter 3 while describing checkpointing protocols ensuring the $\mathcal{RDT}$ property.

2.2.3 Relation Between Properties

Note that a Z-path from a checkpoint $C_{i,x}$ to itself (i.e., a Z-cycle involving that checkpoint) is a particular non-causal Z-path that cannot be doubled by any causal Z-path (such a doubling would lead to a cycle in the Happened-Before relation which is acyclic). This observation straightforwardly implies the following result: if all the non-causal Z-paths are doubled in a checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ of a distributed computation, then no Z-cycle exists in $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$. In terms of properties we get:

$$\mathcal{RDT} \Rightarrow \mathcal{NZC}$$

In other words, if $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ satisfies the $\mathcal{RDT}$ property, then it also satisfies the $\mathcal{NZC}$ property. Therefore, none of the local checkpoints of a checkpoint and communication pattern satisfying $\mathcal{RDT}$ is useless (as no Z-cycle exists in $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ due to the implication between properties). Furthermore, to each checkpoint is on-the-fly associable the set of checkpoints on which it depends on due to Z-paths. The latter feature is not guaranteed in a checkpoint and communication pattern of a distributed computation satisfying $\mathcal{NZC}$ but not $\mathcal{RDT}$.

Chapter 3

Communication-Induced Protocols: Overview

This chapter is devoted to the description of communication-induced checkpointing protocols existing in the literature and ensuring either $\mathcal{NZC}$ or $\mathcal{RDT}$. Recall that this type of protocols induce a separation of checkpoints into basic ones and forced ones (forced checkpoints are triggered whenever, upon the receipt of a message, a predicate $\mathcal{P}$ proper of the protocol is evaluated to $TRUE$).

3.1 Protocols Ensuring the No-Z-Cycle Property

As formally stated, the $\mathcal{NZC}$ property stipulates that no checkpoint of a checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ of a distributed computation is involved in any Z-cycle. Equivalently, $\mathcal{NZC}$ is ensured whenever, each checkpoint belongs to at least one consistent global checkpoint.

As shown in Chapter 1, a simple way to guarantee that each checkpoint belongs to at least one consistent global checkpoint is to start a explicit coordination protocol each time a local checkpoint $C_{i,x}$ is taken by process P_i. Such a coordination will determine a consistent global checkpoint including $C_{i,x}$.

Briatico et al. [12] argued that previous coordination can be realized, in the context of communication-induced checkpointing, by introducing the concept of *sequence number* of a consistent global checkpoint and by piggybacking as control information on the application messages the value of the sequence number.

More technically, each process P_i is endowed with a sequence number sn_i, which is initialized to zero at the beginning of the execution. When a checkpoint $C_{i,x}$ is taken, the current value of sn_i is recorded onto stable storage together with the checkpoint. Hence, to each checkpoint $C_{i,x}$ is associated a sequence number denoted $C_{i,x}.sn$, with $C_{i,1}.sn = 0$. Each time a basic checkpoint is scheduled by P_i then sn_i is increased by one prior to taking the checkpoint. Each time an application message m is sent by P_i to any other process, the value of sn_i is attached to m as control information, denoted as $m.sn$.

As the aim of the protocol by Briatico et al. [12] is to force consistency (i.e., independence) between checkpoints having the same value of the sequence number, then the following behavior characterizes the handling of the receipt of any message: when P_i receives in the checkpoint interval $I_{i,x-1}$ a message m piggybacking a sequence number greater than the local one, then the local sequence number is set to $m.sn$ and a forced checkpoint $C_{i,x}$ is taken prior to processing m (hence, $C_{i,x}.sn = m.sn$).

As an example, in Figure 3.1 process P_1 sends a message m_2 to P_2 after taking checkpoint $C_{1,2}$ whose sequence number is equal to 1 (therefore $m_2.sn = 1$). Upon the receipt of m_2, P_2 takes the forced checkpoint $C_{2,2}$ prior to processing the message and assigns to that checkpoint the sequence number 1 (i.e., the sequence number received with the message m_2).

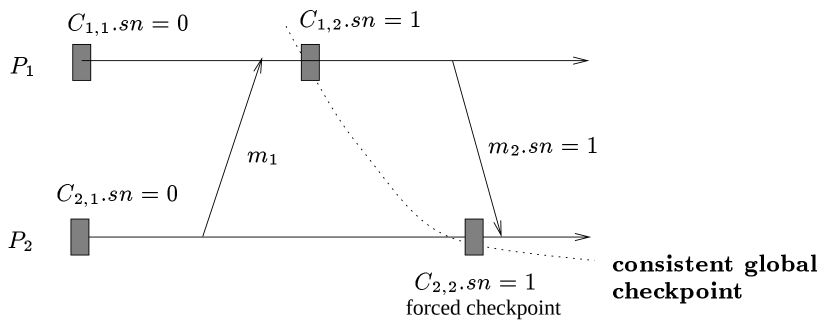


Figure 3.1: An Example of Applying of the Briatico et Al. Protocol.

The $\mathcal{NZC}$ property is guaranteed since the following features are ensured by the protocol to the resulting checkpoint and communication pattern of the computation:

(A) $C_{i,x-1}.sn < C_{i,x}.sn$;

(B) if a message m is sent by P_i after taking $C_{i,x}$ (i.e., $send(m) \in I_{i,x+\epsilon}$ with $\epsilon \geq 0$), then $m.sn \geq C_{i,x}.sn$;

(C) along any Z-path $[m_1, \ldots, m_q]$ from $C_{i,x}$ to $C_{j,y}$ then $\forall p\ :\ 1 \leq p < q \Rightarrow m_{p+1}.sn \geq m_p.sn$;

(D) if a message m is received by P_k, it is received in a checkpoint interval $I_{i,x}$ such that $C_{i,x}.sn \geq m.sn$.

The combination of constraints (A), (B), (C) and (D) guarantees the absence of Z-cycles. As a sketch proof (for a complete proof the reader can refer to [37], where a theoretical framework for classifying communication-induced protocols is presented), let us assume the existence of a Z-cycle involving $C_{i,x}$. It implies the existence of at least a Z-path from $C_{i,x}$ to itself formed by a sequence of messages $[m_1, \ldots, m_q]$; note that m_q is received by P_i before $C_{i,x}$ is taken (i.e., in a checkpoint interval $I_{i,x-\epsilon}$ with $\epsilon > 0$). Due to constraint (B) then $m_1.sn \geq C_{i,x}.sn$; due to constraint (C) $m_q.sn \geq m_1.sn$, hence $m_q.sn \geq C_{i,x}.sn$. As m_q is received by P_i before $C_{i,x}$ is taken, it is received in a checkpoint interval $I_{i,x-\epsilon}$, with $\epsilon > 0$, such that $C_{i,x-\epsilon}.sn < C_{i,x}.sn$ due to constraint (A). By the combination of previous results, $m_q.sn > C_{i,x+\epsilon}.sn$ thus violating constraint (D).

This protocol guarantees that checkpoints with the same sequence number are members of a consistent global checkpoint as, due to previous constraints, no Z-path exists among them (note that, due to the updating rule of the sequence number upon the receipt of a message m, there could be some gap in the sequence numbers assigned to checkpoints by a process; Briatico et al. [12] proved that if a process has not assigned the sequence number num, the first local checkpoint of the process with sequence number num', such that $num' > num$, can be included in the consistent global checkpoint formed by local checkpoints with sequence number num).

From the point of view of the checkpointing overhead, the taking of forced checkpoints pushes the sequence number at some processes higher which may cause more forced checkpoints to be taken. At worst 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. This denotes that the behavior of the protocol in terms of checkpointing overhead may be strongly dependent on the correlation among the policies adopted for taking basic checkpoints at distinct processes. Such an observation is confirmed by simulation results reported in [7].

Furthermore, whenever a consistent global checkpoint associated to a given sequence number is reclaimed, there is no guarantee that the obtained global checkpoint is the closest one to the end of the computation. An example of this drawback is shown in Figure 3.2. If process P_2 reclaims the global consistent checkpoint with sequence number 1 at some point X of its execution, then the global checkpoint $\{C_{1,2}, C_{2,2}, C_{3,2}\}$ is identified which is not the closest one to the end of the computation (the closest one is $\{C_{1,3}, C_{2,2}, C_{3,3}\}$). Such a drawback can lead, in the context of rollback recovery, to rollback extents which are larger than what actually needed to resume the computation from a con-

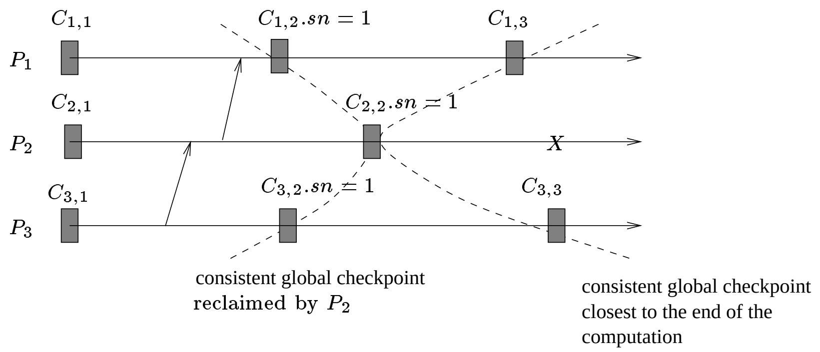


Figure 3.2: P_2 Reclaims a Global Checkpoint which is not the Closest one to the End of the Computation.

sistent global checkpoint. On the other hand, the guarantee that checkpoints with the same sequence number are members of the same consistent global checkpoint allows easy and efficient calculation of a global checkpoint including a given local checkpoint (such a calculation does not require exchange of information among processes related to dependences between checkpoints). As an example of exploitation of this feature in the context of rollback recovery, an efficient asynchronous distributed protocol to rollback the computation to a consistent global checkpoint formed by checkpoints with a given sequence number has been presented by Manivannan and Singhal [36]. In their scheme, the sequence number characterizing the global consistent checkpoint to which the computation must be rolled back is identified by the failed process P_i when resuming the execution (such number corresponds to the sequence number of the last taken checkpoint of P_i). Then the rollback is realized by broadcasting to all the other processes a *rollback* message, carrying the sequence number identified by P_i.

Manivannan and Singhal also presented a quasi synchronous protocol [36] for reducing the checkpointing overhead of the protocol in [12]. In their protocol, each process P_i is endowed with both a sequence number sn_i, and a $next\_to\_be\_assigned_i$ integer variable recording the sequence number to be assigned to the next to be taken checkpoint. The value of the sequence number is piggybacked on any outgoing application message. As in the protocol in [12], upon the receipt of a message m at P_i with $m.sn > sn_i$, sn_i is updated from $m.sn$ and a forced checkpoint is taken.

The assumption underlying the protocol is that every process increments its next_to_be_assigned sequence number at the same regular time interval corresponding to the smallest of the checkpoint time intervals of all the processes (note that such assumption requires processes to have a common clock, hence the computational model is more strict compared to the general one described in Section 1.1 of Chapter 1). This is done in order to keep sequence numbers to be assigned to checkpoints of distinct processes close to each other. Upon the scheduling of a basic checkpoint, the checkpoint is skipped if a forced checkpoint was taken with sequence number equal to the next_to_be_assigned one. Since the sequence numbers of the latest checkpoints of the processes are close to each other, the global consistent checkpoint associated to the sequence number of the last taken checkpoint of P_i results close to the global checkpoint which is the closest one to the end of the computation.

If common clock is not guaranteed, the technique of skipping basic checkpoints still remains a good way to reduce the checkpointing overhead. In particular, a version of the original protocol well suited to computations without common clock among processes is based on the following observation: there is no reason to take a basic checkpoint if at least one forced checkpoint has been taken during the current checkpoint period. So, assuming each 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 is set to $TRUE$ each time a forced checkpoint is taken), then, when P_i schedules a basic checkpoint $C_{i,x}$, such checkpoint is taken only if $skip_i = FALSE$, otherwise it is skipped. Note, however, that the skipping of basic checkpoints sometimes may not be applicable. This may happen, for example, whenever basic checkpoints are scheduled on a non-periodic basis.

Another improvement of the protocol in [12] aiming at reducing the number of forced checkpoints per basic one has been presented by Hélary et al. [28]. The protocol exploits the information spread by causality about values of the sequence numbers of the processes in order to ensure that if there exists a Z-path from $C_{i,x}$ to $C_{j,y}$ then $C_{i,x}.sn < C_{j,y}.sn$. This is a guarantee that no Z-cycle can even be formed. More technically, if the protocol would allow the formation of a Z-cycle involving $C_{i,x}$, then there should exist a Z-path from $C_{i,x}$ to itself; in such a case the inequality $C_{i,x}.sn < C_{i,x}.sn$ should be verified, which is, obviously, an absurd.

In the presented protocol, the sequence number sn_i of P_i becomes the *local clock* lc_i. Each process P_i piggybacks on any application message m the following data structures: $clock_i$ ($m.clock$), $ckpt_i$ ($m.ckpt_i$) and $taken_i$ ($m.taken$). The explanation of the data structures is as follows:

- $clock_i$
  is a vector of n integers with the following meaning: $clock_i[j]$ represents, to the knowledge of P_i the highest value of the local clock of P_j (i.e., lc_j); upon the receipt of a message m, P_i updates $clock_i$ from $m.clock$ by taking a component-wise maximum;

- $ckpt_i$
  is a vector of n integers with the following meaning: $ckpt_i[j]$ represents,

to the knowledge of P_i the highest value of the rank of checkpoints of P_j (i.e., it counts how many checkpoints have been taken by P_j to the knowledge of P_i); upon the receipt of a message m, P_i updates $ckpt_i$ from $m.ckpt$ by taking a component-wise maximum;

- $taken_i$
  is a vector of n booleans with the following meaning: $taken_i[j]$ is equal to $TRUE$ if there exists a causal Z-path between the last checkpoint of P_j seen by P_i through causality and the next checkpoint of P_i, and the causal Z-path includes a checkpoint (the updating rule of this vector is out of the scope of this description).

Furthermore, process P_i has the following local data structures:

- $send\_to_i$
  which is a vector of n booleans; $send\_to_i[j]$ is equal to $TRUE$ iff P_i sent a message to P_j in its current checkpoint interval;

- min_to_i
  which is a vector of n integers; $min\_to_i[j]$ records the local clock of P_i which has been piggybacked on the first message sent by P_i to P_j in its current checkpoint interval.

Basing on previous data structures, the authors introduce a protocol guaranteeing that no Z-Path exists between any pair of checkpoints $(C_{i,x}, C_{j,y})$ such that $C_{i,x}.sn = C_{j,y}.sn$. Therefore no Z-path exists from a checkpoint to itself, implying the absence of Z-cycles. In the protocol, a forced checkpoint is taken by P_i upon the receipt of a message m sent by P_j if the following predicate holds:

$$
\begin{aligned}
\mathcal{P} \;\equiv\; \exists k \;:\; & send\_to_i[k] \wedge \\
& (m.clock[j] > min\_to_i[k]) \wedge \\
& ((m.clock[j] > max(clock_i[k], m.clock[k])) \vee \\
& \;\; (m.ckpt[i] = ckpt_i[i] \wedge m.taken[i]))
\end{aligned}
$$

Basically predicate $\mathcal{P}$ means that there exists a process P_k such that P_i sent a message to P_k in its current checkpoint interval and: the local clock of P_j piggybacked on m is larger that the local clock of P_k known by P_i through causality, or, there exists a causal Z-path between the last checkpoint of P_i and the next to be taken checkpoint of P_i which includes a checkpoint of a process.

The authors proved that other protocols [12, 36] trigger the forced checkpoint according to a predicate $\mathcal{P}'$ such that $\mathcal{P} \Rightarrow \mathcal{P}'$. However, the potential reduction of the checkpointing overhead due to the reduction of the number of forced checkpoints per basic one compared to the other protocols is not quantified. Recall that the inclusion between predicates means that the protocol by Hélary et al. takes a forced checkpoint whenever the other protocols do it only under the same causal past. As there is no guaranty that the computation evolves at the same way under different checkpointing protocols, performance of the protocol by Hélary et al., in terms of forced checkpoints per basic one, is not guaranteed to be better than that of the other protocols. This is the reason why we use the term "potential reduction".

All previous protocols ensure that checkpoints with the same sequence number are members of the same consistent global checkpoint. However, not all dependences between checkpoints in $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ due to Z-paths can be known by a process as these protocols allow non-causal Z-paths to be not causally doubled (i.e., $\mathcal{RDT}$ is not guaranteed). Let us consider the example with three processes shown in Figure 3.3 where CGC_0 (resp. CGC_1) represents the consistent global checkpoint formed by checkpoints with sequence number equal to 0 (resp. 1). There exists a non causal Z-path from checkpoint $C_{3,1}$ to checkpoint $C_{1,2}$ due to messages $[m_1, m_2]$ which is not causally doubled.

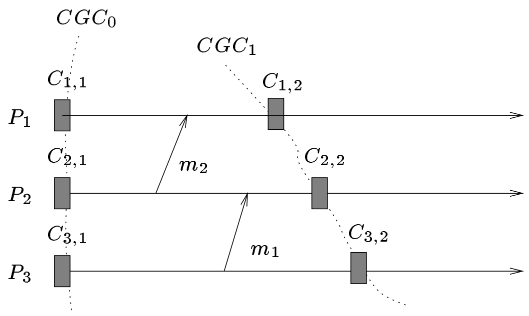


Figure 3.3: An Example of Z-path which is not Causally Doubled.

Partial Absence of Z-cycles

Wang and Fuchs [61] modified the protocol in [12] by introducing the notion of *laziness*. The latter is a positive integer Z such that only checkpoints with sequence number which is a multiple of Z are mutually consistent (i.e., no Z-paths exists among them). Therefore, only the global checkpoints consisting of local checkpoints with sequence number which is a multiple of Z are guaranteed to be consistent. This protocol shows the advantage of a reduction of the number of forced checkpoints (as the lazy coordination acts less frequently) but has the disadvantage to not guarantee $\mathcal{NZC}$ (as only local checkpoints with sequence number which is a multiple of Z are guaranteed to belong to a consistent global checkpoint). In the context of rollback recovery, this protocol allows the possibility to reduce the checkpointing overhead at the expense of a potentially larger rollback extent. If the laziness parameter Z is set to one, the protocol boils down to the Briatico et al. one.

A different approach to the partial absence of Z-cycles has been presented by Xu and Netzer in [67]. They introduced a checkpointing protocol which prevents the formation of a particular type of Z-cycles. The particular type of Z-cycle, that for the sake of clarity is below referred to as XN-cycle, is defined as follows:

Definition 3.1.1
A Z-cycle due to $[m_1, \ldots, m_q]$ involving checkpoint $C_{i,x}$ is an XN-cycle iff:

$$\mathcal{R} \equiv \forall p \ : \ 2 \leq p < q \Rightarrow receive(m_p) \prec_p send(m_{p+1})$$

In other words, an XN-cycle is a Z-cycle in which message m_1 is the only one that is received after the successive message in the sequence is sent (i.e., the sequence of messages $[m_2, \ldots, m_q]$ constitutes a causal path).

Their protocol induces the recipient process of message m_1 to take a forced checkpoint upon the receipt of such message. As an example, in Figure 3.4.a we have a Z-cycle involving $C_{1,2}$ formed by messages $[m_1, m_2, m_3]$. In this Z-cycle, predicate $\mathcal{R}$ holds as only for message m_1 we have $\neg(receive(m_1) \prec_p send(m_2))$. This Z-cycle is prevented by the protocol through a forced checkpoint $C_{3,2}$ taken upon the receipt of m_1 (see Figure 3.4.b). In Figure 3.5 a Z-cycle which in not prevented by the protocol is shown (in this Z-cycle, predicate $\mathcal{R}$ does not hold as there are two messages, m_1 and m_2, for which $\neg(receive(m_1) \prec_p send(m_2))$ and $\neg(receive(m_2) \prec_p send(m_3)))$.

In their protocol, each process P_i maintains a dependency vector DV_i of n integers. The i-th entry records the rank of the last checkpoint taken by P_i. The j-th entry records the rank of the last checkpoint taken by P_j known by P_i through causality. Causal information is spread among processes by piggybacking on each outgoing message m the current value of DV ($m.DV$). Upon the receipt of a message m by P_i, the vector DV_i is updated from $m.DV$ by taking a component-wise maximum.

When a checkpoint is taken, the value of DV_i is copied into a vector ZV_i. $ZV_i[j] = y$ means that there exists a causal path from the y-th checkpoint of process P_j to the $DV_i[i]$-th checkpoint of process P_i. Each message m sent by P_i to P_j piggybacks, together with the current value of DV_i, the integer $ZV_i[j]$ ($m.Zid$). Upon the receipt of m, P_j takes a forced checkpoint if $m.Zid = DV_j[j]$.

Although both protocols in [61] and [67] allows the presence of Z-cycles, they are completely different. The protocol in [61] guarantees that, at some point of the computation, a global checkpoint will exist which is distinct from the initial one (unless $Z = \infty$). On the contrary, the protocol in [67] does not guarantee that feature, as the absence of XN-cycles does not imply that a global checkpoint will ever be formed.

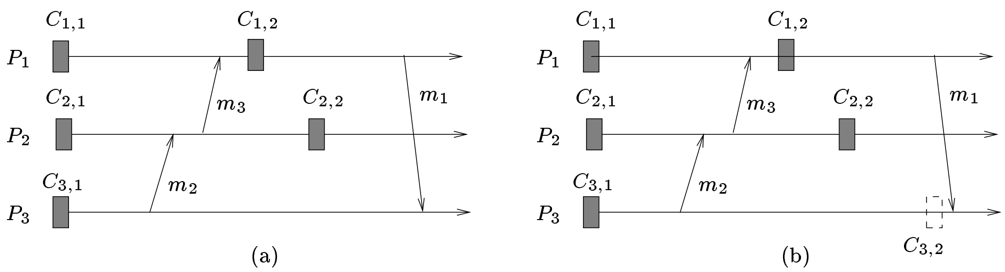


Figure 3.4: A Z-cycle in which Predicate $\mathcal{R}$ Holds (a); the Z-cycle is Prevented by the Xu-Netzer Protocol Through the Forced Checkpoint $C_{3,2}$ (b).

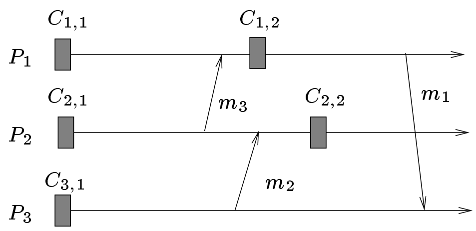


Figure 3.5: Z-cycle which is not Prevented by the Xu-Netzer Protocol.

3.2 Protocols Ensuring the Rollback-Dependency-Trackability Property

The $\mathcal{RDT}$ property has been introduced by Wang [64]. He also designed a checkpointing protocol ensuring that property which is a generalization of several previous existing protocols ensuring the same property. A protocol which cannot be considered as deriving from Wang's protocol has been presented in [4] (a preliminary version also appeared in [5]). Such a protocol will be discussed in this section as last one.

Wang's protocol ensures $\mathcal{RDT}$ by exploiting the Fixed-Dependency-After-Send (FDAS) model. This model can be easily explained by looking at Figure 3.6 showing an example involving three processes. In the computation in

Figure 3.6.a, there exists a Z-path from $C_{3,1}$ to $C_{1,2}$, due to messages $[m_1, m_2]$, which is not causally doubled. In this case the FDAS model pushes P_2 to take a forced checkpoint before the receipt of m_2 in order to prevent the formation of that non-causal Z-path. The resulting checkpoint and communication pattern of the computation does not contain non-causal Z-paths which are not causally doubled. On the other hand, if there exists the message m_3, as shown in Figure 3.6.b, then there is no need to take the forced checkpoint as the Z-path from $C_{3,1}$ to $C_{1,2}$ due to $[m_1, m_2]$ is doubled by the causal Z-path due to $[m_3, m_2]$. The substantial difference between the two scenarios is as follows. In Figure 3.6.b, at the time of sending message m_2 establishing a dependence involving $C_{1,2}$, process P_2 already tracked by causality the existence of $C_{3,1}$, hence, always by causality, process P_1 can track the dependence of $C_{1,2}$ on $C_{3,1}$ due to the Z-path formed by $[m_3, m_2]$. In Figure 3.6.a, P_2 tracks the existence of $C_{3,1}$ only upon the receipt of m_1 when the dependence due to m_2 was already generated. As a consequence P_1 is prevented to track the dependence due to $[m_1, m_2]$. The insertion of the forced checkpoint $C_{2,2}$ in the scenario in Figure 3.6.a prevents the formation of the non-trackable dependence between $C_{3,1}$ and $C_{1,2}$.

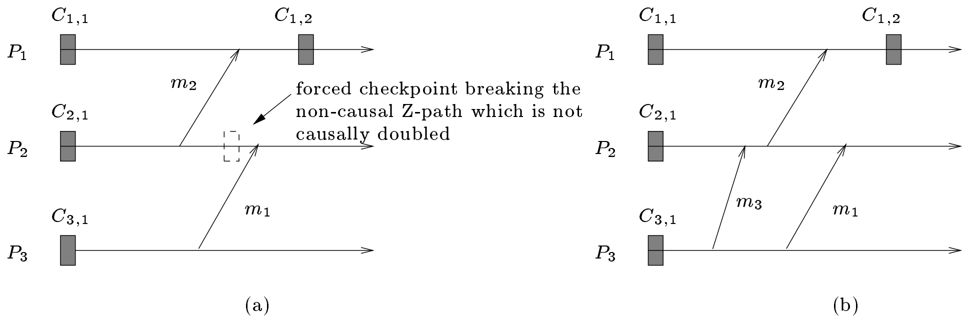


Figure 3.6: The FDAS model.

Wang has shown [64] that FDAS can be implemented by endowing each process P_i with a transitive dependency vector D_i of n integers and a boolean variable $after\_first\_send_i$ indicating if there has been at least a send event in the current checkpoint interval of P_i.

$D_i[i]$ represents the rank of the last checkpoint taken by P_i. Such a vector is piggybacked as control information on any message m ($m.D$). Upon the receipt of a message m, D_i is updated from $m.D$ by taking a component-wise maximum. Hence, the j-th entry represents the maximum rank of checkpoints of P_j known by P_i through causality. Upon the receipt of a message m in the checkpoint interval $I_{i,x}$ which is about to chance at least one entry of D_i (i.e., P_i becomes aware of the existence of at least one new checkpoint) then a forced checkpoint is taken by P_i if $after\_first\_send_i = TRUE$.

In the FDAS protocol, all dependences due to Z-paths involving checkpoints of P_i are tracked by causality (as the outcoming checkpoint and communication pattern of the distributed computation satisfies $\mathcal{RDT}$) and are recorded in the vector D_i. More technically, if at the time $C_{i,x}$ is taken, $D_i[j] = y$, then there exists a dependence between the y-th checkpoint of P_j and $C_{i,x}$ due to a Z-path. P_i learns that no dependence due to a Z-path will ever exist between $C_{j,y+1}$ and $C_{i,x}$, therefore they can be members of a consistent global checkpoint. Let $D_{i,x}$ be a vector of n integers associated to $C_{i,x}$ and obtained, at the time $C_{i,x}$ is taken, as follow:

- $D_{i,x}[i] = x$;
- $\forall j \ : \ (1 \leq j \leq n) \wedge (j \neq i) \rightarrow D_{i,x}[j] = D_i[j] + 1$;

then, Wang proved the following theorem:

Theorem 3.2.1
Given a checkpoint $C_{i,x}$ of a checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ of a distributed computation satisfying $\mathcal{RDT}$, the minimum consistent global checkpoint containing $C_{i,x}$ can be computed as:

$$\cup_{1 \leq j \leq n} C_{j, D_{i,x}[j]}$$

Theorem 3.2.1 implicitly states that the calculation of the minimum consistent global checkpoint containing a given checkpoint $C_{i,x}$ can be done locally by P_i without the need for exchanging dependency information with other processes. Protocols for minimum and/or maximum consistent global checkpoints collection can be found in [1, 29, 35, 63, 64].

Other communication-induced checkpointing protocols ensuring $\mathcal{RDT}$ to the outcoming checkpoint and communication pattern of the distributed computation are discussed below.

The Fixed-Dependency-Interval (FDI) protocol [58] is a derivation of FDAS. Upon the receipt of a message m, FDI induces P_i to take a forced checkpoint if at least one entry of D_i is about to be chanced (irrespective whether there have been send events in the current checkpoint interval).

Another protocol deriving from FDAS is No-Receive-After-Send (NRAS) in which a checkpoint is taken by P_i upon the receipt of m if $after\_first\_send_i = TRUE$ (irrespective whether the D_i vector is going to be updated). Such a protocol is equivalent to Russell's MRS protocol [50] where M stands for "take a checkpoint", S stands for "send" and R stands for "receive". NRAS (and therefore also MRS) generates a checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ of a distributed computation in which in any checkpoint interval there does not exist a send event preceding a receive one (i.e., all Z-paths in $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$

are causal). For the context of rollback recovery, a modification of Russell's protocol has been presented in [15]. The protocol does not allow a send event to precede a receive one in a checkpoint interval; furthermore, a checkpoint is taken after α consecutive receive events in order to reduce the amount of lost work in case of failure (α is selected in function of failure probability and other system parameters).

Other deriving protocols are: Checkpoint-Before-Receive (CBR) in which a forced checkpoint is taken before the receipt of any message; Checkpoint-After-Send (CAS) in which a forced checkpoint is taken after the send of any message and Checkpoint-After-Send-Before-Receive (CASBR) in which a checkpoint is taken both before the receipt and after the send of any message. Also for CBR, CAS and CASBR no send event precedes a receive one in any checkpoint interval.

Note that NRAS, CBR, CAS and CASBR actually may work without the need for piggybacking control information. Indeed, the piggybacked dependency vector is used only to track dependences between checkpoints by not to determine the insertion of forced checkpoints (i.e., the predicate that triggers forced checkpoints is evaluated by using only the local context of a process related to events occurred in the current checkpoint interval).

All previously described protocols are based on the removal of some Z-paths in order to ensure the absence of non-causal Z-paths which are not causally doubled. Baldoni et al. [3] gave a characterization of $\mathcal{RDT}$ by founding a small subset of Z-paths to be causally doubled in order to ensure that all Z-paths of the checkpoint and communication pattern of the computation are causally doubled. They termed these Z-paths as Elementary-Prime-Simple-Causal-Message-Z-paths (EPSCM-paths), and introduced a protocol which breaks only those EPSCM-paths which, upon their formation, are perceived as not causally doubled (a technical description of the protocol, which appeared in [4], will be given in Chapter 6 where a comparison with a checkpointing protocol presented in the same chapter is performed). Simulation results have shown that their protocol, compared to all the other protocols ensuring $\mathcal{RDT}$, achieves a reduction of the number of forced checkpoints in any environment (e.g., client-server, master-slave etc.).

As final point of this section we recall some concepts of the classification of protocols ensuring $\mathcal{RDT}$ presented by Manivannan and Singhal in [37]. These protocols are splitted into two classes: Strictly Z-path Free (SZpF) and Z-path Free (ZpF). The classification is based on the degree to which non-causal Z-paths are allowed by the protocol in the checkpoint and communication pattern of the computation.

A communication-induced checkpointing protocol is said to be SZpF iff it generates a checkpoint and communication pattern of a distributed computation containing no non-causal Z-path. NRAS, CAS, CBR, and CASBR are examples of SZpF protocols as they do not allow the formation of non-causal Z-paths. The disadvantage of an SZpF protocol is the potentially unacceptable checkpointing overhead needed for the prevention of all non-causal Z-paths.

A communication-induced checkpointing protocol is said to be ZpF iff it generates a checkpoint and communication pattern of a distributed computation in which all non-causal Z-paths are causally doubled. A ZpF protocol has the same advantages of an SZpF one concerning the possibility to use information related to causality for determining consistent global checkpoints (as under both type of protocols dependences due to causal Z-paths are representative of all dependences between checkpoints). Furthermore, a ZpF protocol, compared to an SZpF one, shows a potential reduction of the checkpointing overhead as checkpoints are taken only to prevent the formation of non-causal Z-path which are not causally doubled (i.e., not all the non-causal Z-paths are prevented).

Chapter 4

A Taxonomy of Protocols

This chapter is devoted to the introduction of a taxonomy of communication-induced checkpointing protocols ensuring either $\mathcal{NZC}$ or $\mathcal{RDT}$. The taxonomy relies on the Virtual-Precedence ($\mathcal{VP}$) property introduced by Hélary et al. [27].

We show that, although protocols ensuring $\mathcal{NZC}$ (or $\mathcal{RDT}$) also ensure the $\mathcal{VP}$ property to the outcoming checkpoint and communication pattern of the distributed computation, a taxonomy of protocols can be made basing on the way the $\mathcal{VP}$ property is used in the design of the protocol. The proposed taxonomy splits protocols in: $\mathcal{VP}$*-enforced* and $\mathcal{VP}$*-accordant.*

4.1 The Virtual Precedence Property

4.1.1 Description

As shown in Chapter 1, a distributed computation can be modeled as a partially ordered set of events. A higher level *abstraction* of the computation has been introduced in [27] by considering the execution of each process as a sequence of *intervals*. Each interval consists of a set of consecutive events produced by the process. The proposed abstraction is such that

- every event belongs to a single interval;
- every interval contains at least one event.

The i-th interval of process P_i in the abstraction is denoted as $I_{i,x}$.

To the abstraction of the computation a directed graph is associated, namely *Abstraction-graph* (*A-graph*), structured as follows:

- each vertex corresponds to an interval $I_{i,x}$;

- there exists an edge from $I_{j,y}$ to $I_{i,x}$ if:
  - $j = i$ and $y = x - 1$ (*local* edge); or
  - there exists a message m such that $(send(m) \in I_{j,y}) \wedge (receive(m) \in I_{i,x})$ (*communication* edge).

Note that abstractions of different computations can produce the same A-graph. Furthermore, depending on the abstraction, the A-graph may have cycles.

Let consider each message m and each interval $I_{i,x}$ to be marked with a *timestamp*. Informally, an abstraction of a distributed computation satisfies the $\mathcal{VP}$ property if it is possible to timestamp messages and intervals in a way that:

F1 : for any pair of messages m and m' such that $receive(m) \in I_{i,x}$ and $send(m') \in I_{i,x}$ then the timestamp of m is smaller than or equal to the timestamp of m';

F2 : the timestamp of $I_{i,x}$ is larger than or equal to the timestamp of all messages received in $I_{i,x}$ and is smaller than or equal to the timestamp of all messages sent in $I_{i,x}$.

This means that, in the logical time (timestamp), communications can be seen as causal in each interval. That is, communication events can be reordered in any interval making all the receive events to precede all the send events and timestamp does not decrease following causal paths. An example of this is shown in Figure 4.1.

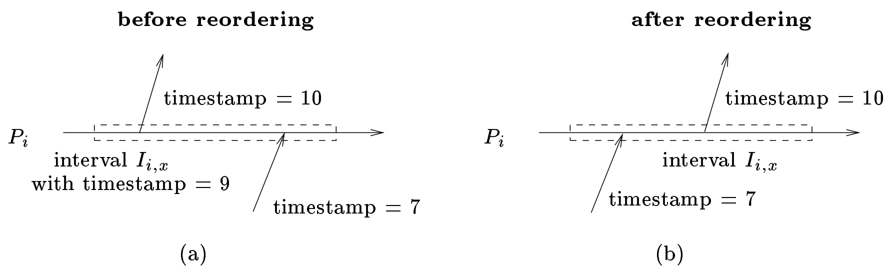


Figure 4.1: The Virtual Precedence Property.

In other words, an interval-based abstraction of a distributed computation satisfies $\mathcal{VP}$ if, and only if, it is possible to associate a *timestamping function* within intervals with the following characteristics: (i) intervals which are connected by a message must be timestamped in a non-decreasing way (safety part) and (ii) the timestamp of a process must increase after communication (liveness part). It is easy to see that if we consider each interval $I_{i,x}$ formed by a single event, then the timestamping function boils down to the Lamport's scalar clock [33] or the Fidge-Mattern's vector time [21, 38]. Hélary et al. ([27]) proved the following theorem:

Theorem 4.1.1
An abstraction of a distributed computation $\widehat{\mathcal{H}}$ satisfies the $\mathcal{VP}$ property iff the corresponding A-graph has no cycle including a local edge.

4.1.2 Equivalence Between the No-Z-Cycle Property and the Virtual Precedence Property

In the particular context of the checkpointing problem, intervals of the abstraction correspond to checkpoint intervals. Then, the abstraction of the distributed computation corresponds to a checkpoint and communication pattern.

Given a checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ of a distributed computation $\widehat{\mathcal{H}}$, the following properties hold [27]:

(A) if there exists a Z-cycle in $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ then the A-graph corresponding to the abstraction of the computation contains at least one cycle involving a local edge;

(B) if no Z-cycle exists in $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ then the A-graph corresponding to the abstraction of the computation contains no cycle involving a local edge.

Property (A) means $\mathcal{VP} \Rightarrow \mathcal{NZC}$. Property (B) means $\mathcal{VP} \Leftarrow \mathcal{NZC}$. Therefore, Theorem 4.1.1 can be reformulated as:

Theorem 4.1.2
A checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ of a distributed computation satisfies the $\mathcal{VP}$ property iff it satisfies the $\mathcal{NZC}$ property (i.e., $\mathcal{VP} \Leftrightarrow \mathcal{NZC}$).

4.2 A Taxonomy of Protocols Based on the Virtual Precedence Property

All communication-induced checkpointing protocols generating checkpoint and communication patterns $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ which satisfy $\mathcal{NZC}$, make these checkpoint and communication patterns to satisfy $\mathcal{VP}$ as well due to Theorem 4.1.2. This means the $\mathcal{VP}$ property constitutes a common basis for all such protocols.

However, the design of a checkpointing protocol not necessarily relies on such a common basis. In the following sections we exploit latter concept by introducing the notions of $\mathcal{VP}$-enforced protocol and $\mathcal{VP}$-accordant protocol.

4.2.1 $\mathcal{VP}$-Enforced Protocols

Let a timestamping function be assumed to timestamp messages and checkpoint intervals consistently with rules F1 and F2 described in Section 4.1.1. Then a checkpointing protocol ensuring $\mathcal{VP}$ to the outcoming checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ of the computation can be derived as follows. Timestamps are piggybacked on any sent application message. Then, upon the arrival of a message m at P_i in the checkpoint interval $I_{i,x}$, the communication-induced checkpointing protocol pushes P_i to take a forced checkpoint $C_{i,x+1}$ before receiving m whenever one of the rules F1 or F2 would be violated by that receive event. The new created checkpoint interval $I_{i,x+1}$ is then timestamped by the protocol according to the chosen timestamping function. In this approach we have:

- the timestamp assigned to the checkpoint interval $I_{i,x+1}$ depends on the chosen timestamping function;
- the timestamps assigned to messages sent in $I_{i,x+1}$ depend on the timestamping function (note that such timestamps cannot be smaller than the timestamp assigned to $I_{i,x+1}$ due to rule ii); they can assume the same value of the timestamp assigned to the interval $I_{i,x+1}$).

We name any protocol designed starting, has done above, by an *a priori* assumed timestamping function as a $\mathcal{VP}$-enforced protocol.

The goodness of a $\mathcal{VP}$-enforced protocol, evaluated in terms of induced checkpoints per basic checkpoint, depends on the goodness of the a priori assumed timestamping function. In particular, the timestamping function is considered as "good" if the deriving checkpointing protocol produces checkpoint and communication patterns with a low number of forced checkpoints per basic one (note that low number of forced checkpoints per basic one implicitly means low probability that upon the receipt of a message either rule F1 or rule F2 is violated).

Basically, the predicate that triggers the taking of the forced checkpoint $C_{i,x+1}$ is evaluated to $TRUE$ whenever the message m arriving at P_i in $I_{i,x}$ piggybacks a timestamp larger than the timestamp assigned to $I_{i,x}$. Hence, less forced checkpoints are taken whenever timestamps of incoming messages do not exceed timestamps of local checkpoint intervals. Two main approaches can be envisaged for achieving this:

(1) let the timestamps of checkpoint intervals to increase at the same speed at distinct processes;

(2) let the timestamps of checkpoint intervals to increase as slowly as possible.

Approach (1), envisaged for example in the protocol by Manivannan and Singhal [36] requires a kind of synchronization of checkpointing clocks at distinct processes, thus imposing a constraint on the computational model described in Chapter 1. In particular, if basic checkpoints are taken at the same physical time and newly created intervals are timestamped with the same timestamp value, then no forced checkpoint is ever taken.

Approach (2) consists of refining the a priori assumed timestamping function as much as possible in order to slow down the rate for the increasing of the timestamp at each process. In latter context both the protocol by Briatico et al. [12] and the protocol by Hélary et al. [28] can be seen as generated by an a priori assumed timestamping function (the function is such that the timestamp does not decrease along any Z-path), and the timestamping function of the latter protocol can be considered as a refinement of the timestamping function of the former one.

4.2.2 $\mathcal{VP}$-Accordant Checkpointing Protocols

We name $\mathcal{VP}$-accordant any protocol which is designed without a priori assuming a timestamping function consistent with rules i) and ii) of Section 4.1.1. Instead, it relies on the study of the structure of sub-patterns (i.e., portions) of a checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ of a distributed computation.

Sometimes it is possible to prove that if a checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ does not contain sub-patterns with a given structure, namely STR, then $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ satisfies $\mathcal{NZC}$ or $\mathcal{RDT}$. An example of this is the study presented by Baldoni et al. in [3] where, as outlined in Section 3.2 of Chapter 3, it is shown that if all EPSCM-paths are causally doubled then the checkpoint and communication pattern of the distributed computation satisfies $\mathcal{RDT}$ (in such a case, the structure STR to be avoided is that of an EPSCM-path which is not causally doubled).

The absence of sub-patterns with structure STR is, therefore, a sufficient condition guaranteeing $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ satisfies either $\mathcal{NZC}$ or $\mathcal{RDT}$. Thus the design of a $\mathcal{VP}$-accordant protocol starts by the identification of the structure STR, whose formation has to be prevented by the protocol itself.

In this approach the predicate that triggers the action to take a forced checkpoint $C_{i,x+1}$ at P_i upon the receipt of a message m depends upon the structure of checkpoint and communication sub-patterns that are going to be formed if the message would be received by P_i in $I_{i,x}$. Thus, if the predicate is evaluated to $TRUE$, at least one "bad" checkpoint and communication sub-pattern (i.e., one having structure STR) is going to be formed. Then the protocol takes a forced checkpoint to prevent the formation of that pattern.

As $\mathcal{VP} \Leftrightarrow \mathcal{NZC}$, also for a $\mathcal{VP}$-accordant protocol there will exist a timestamping function that could be used to timestamp checkpoint intervals of the computation produced by the protocol consistently with rules i) and ii). However such a function is not used while designing the protocol.

4.3 Applying the Taxonomy to Existing Protocols

In this section the proposed taxonomy is applied to classify existing communication-induced checkpointing protocols discussed in Chapter 3.

All communication-induced checkpointing protocols ensuring $\mathcal{NZC}$ described in Section 3.1 of Chapter 3 are in the $\mathcal{VP}$-enforced class. This is because they are designed by assuming a timestamping function which prevents timestamp from decreasing along any Z-path.

Also the FDI protocol ensuring $\mathcal{RDT}$ [58] described in Section 3.2 of Chapter 3 is in the $\mathcal{VP}$-enforced class. This is because it timestamps messages and intervals with a dependency vector and takes forced checkpoints whenever, upon the receipt if a message at least one entry of the local dependency vector is about to be changed (irrespective of the sub-patterns that are going to be formed due to that receive event). Then, if we consider the following relation among two different timestamps T_1 and T_2:

$$T_1 \leq T_2 \Leftrightarrow \forall j \ : \ 1 \leq j \leq n \Rightarrow T_1[j] \leq T_2[j]$$

the protocol imposes that the timestamp of messages does not decrease along any Z-path. Hélary et al. [27] defined a *meta* timestamping function and showed that all above mentioned protocols derive from instantiations of the meta function (i.e., instantiations of a meta protocol).

All the other protocols ensuring $\mathcal{RDT}$, described in Section 3.2 of Chapter 3, are in the $\mathcal{VP}$-accordant class, as they aim at preventing sub-patterns with a given structure.

A graphical representation of the application of the taxonomy is shown in Table 4.3.

Note that, to the best of our knowledge, there does not exist any communication-induced checkpointing protocol belonging to the $\mathcal{VP}$-accordant class which ensures $\mathcal{NZC}$ but not $\mathcal{RDT}$. We will design protocols with this feature in Chapter 6, by preliminary studying sub-patterns of a checkpoint and communication pattern of a distributed computation. In particular, properties on Z-cycles will be studied and a particular type of Z-cycle, namely *core Z-cycle*, is identified such that, given a checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ of a distributed computation, then no Z-cycles exists in it iff no core Z-cycles exists. The designed protocols prevent the formation of core Z-cycles ensuring that the outcoming checkpoint and communication pattern of the distributed computation satisfies $\mathcal{NZC}$.

ensured property	$\mathcal{VP}$-enforced protocols	$\mathcal{VP}$-accordant protocols
$\mathcal{NZC}$	Briatico et al. Manivannan-Singhal Hélary et al.	
$\mathcal{RDT}$	FDI	FDAS NRAS MRS CAS CBR CASBR Baldoni et al.

Table 4.1: Application of the Taxonomy to Existing Protocols.

Chapter 5

A Virtual Precedence Enforced Protocol

This chapter is devoted to the design of a $\mathcal{VP}$-enforced checkpointing protocol ensuring $\mathcal{NZC}$. The protocol is designed starting from a notion of *equivalence* between local checkpoints of a process, here introduced. Such a notion allows to slow down the rate at which timestamps grow at distinct processes, thus reducing the probability of forced checkpoints. Therefore, such protocol exploits approach (2) envisaged in Section 4.2.1 of Chapter 4.

The usefulness of the proposed protocols is demonstrated by simulation results of a case study in the context of rollback recovery.

Note that the equivalence relation here defined provides actually a framework that can form a basis for the design of other communication-induced protocols in the $\mathcal{VP}$-enforced class. Furthermore, the presented protocol does not represent an instantiation of the meta protocol by Hélary et al. [27] as it uses the notion of *provisional* timestamp (that will be referred to as provisional index) which is not considered in the meta protocol.

5.1 Relation of Equivalence Between Checkpoints

Let consider two successive checkpoints $C_{i,x}$ and $C_{i,x+1}$ of process P_i. We define the following equivalence relation among them:

Definition 5.1.1
Two local checkpoints $C_{i,x}$ and $C_{i,x+1}$ of process P_i are equivalent with respect to a consistent global checkpoint CGC, denoted $C_{i,x} \overset{CGC}{\equiv} C_{i,x+1}$, if:

(i) $C_{i,x} \in CGC$; *and*

(ii) $\forall C_{j,y} \in CGC \ : \ j \neq i \Rightarrow \neg(C_{j,y} \prec_{ckpt} C_{i,x+1})$.

In other words, if $C_{i,x}$ belongs to the consistent global checkpoint CGC then $C_{i,x+1}$ is equivalent to $C_{i,x}$ with respect to CGC if it does not depend, through the relation $\prec_{ckpt}$, on any checkpoint in CGC. As an example of equivalence, in Figure 5.1 a scenario with three processes is shown. There exists a global consistent checkpoint $CGC = \{C_{1,x_1}, C_{2,x_2}, C_{3,x_3}\}$. As C_{2,x_2+1} does not depend through the $\prec_{ckpt}$ relation on both C_{1,x_1} and C_{2,x_2} then $C_{2,x_2} \stackrel{CGC}{\equiv} C_{2,x_2+1}$.

From a graphical point of view, we can distinguish a right end side of the computation with respect to CGC and a left end side. The right end side consists of events produced by any process P_j after taking the checkpoint C_{j,x_j} belonging to CGC. Instead, the left end side consists of events produced by any process P_j before C_{j,x_j}, belonging to CGC, is taken. Going back to the example in Figure 5.1, C_{2,x_2+1} does not depend, through the $\prec_{ckpt}$ relation, on any checkpoint in CGC means there does not exist any message m which has been sent from the right end side of CGC and is received by P_2 before C_{2,x_2+1} is taken.

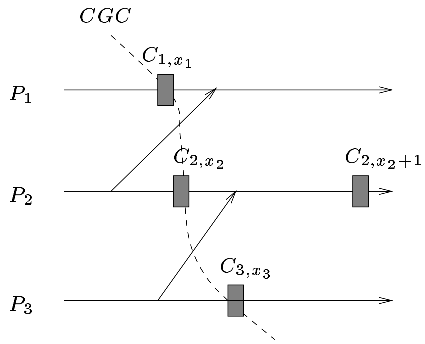


Figure 5.1: An Example of Pairs of Equivalent Checkpoints.

The notion of equivalence between local checkpoints is important because of the following lemma:

Lemma 5.1.1
If $C_{i,x_i} \stackrel{CGC}{\equiv} C_{i,x_i+1}$ *then the set of checkpoints* $CGC - \{C_{i,x_i}\} \cup \{C_{i,x_i+1}\}$ *is a consistent global checkpoint.*

Proof
$C_{i,x_i} \stackrel{CGC}{\equiv} C_{i,x_i+1}$ implies CGC is a consistent global checkpoint including C_{i,x_i}. Therefore, by Definition 5.1.1

$$\forall C_{j,x_j} \in CGC \ : \ j \neq i \Rightarrow \neg(C_{j,x_j} \prec_{ckpt} C_{i,x_i+1})$$

As CGC is consistent, then, by Definition 1.3.2

$$\forall C_{j,x_j} \in CGC \ : \ j \neq i \Rightarrow \neg(C_{i,x_i+1} \prec_{ckpt} C_{j,x_j})$$

thus $CGC - \{C_{i,x_i}\} \cup \{C_{i,x_i+1}\}$ is a consistent global checkpoint. $Q.E.D.$

Having two successive checkpoints C_{i,x_i} and C_{i,x_i+1} of process P_i equivalent with respect to a given consistent global checkpoint CGC implies:

(i) the first checkpoint, namely C_{i,x_i} belongs to CGC (i.e., it is involved in no Z-cycle);

(ii) by Lemma 5.1.1, the second checkpoint, namely C_{i,x_i+1} automatically belongs to the consistent global checkpoint CGC' obtained by substituting C_{i,x_i+1} to C_{i,x_i} in CGC (therefore also C_{i,x_i+1} is involved in no Z-cycle).

An example of non-equivalent checkpoints C_{2,x_2} and C_{2,x_2+1} with respect to CGC is shown in Figure 5.2. The non-equivalence is due to the presence of message m which establishes the following relation: $C_{3,x_3} \prec_{ckpt} C_{2,x_2+1}$ (i.e., there exists a message m which has been sent from the right end side of CGC and has been received by P_2 before C_{2,x_2+1} is taken). Note that in this case, if we substitute C_{2,x_2+1} to C_{2,x_2} in CGC we obtain a global checkpoint which is not consistent.

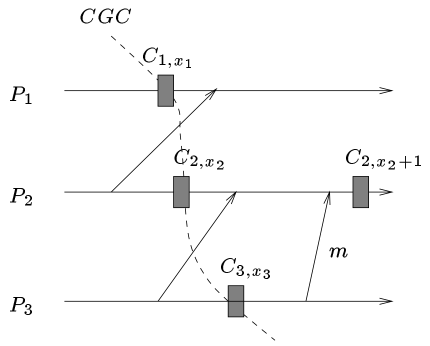


Figure 5.2: C_{2,x_2} is not Equivalent to C_{2,x_2+1} with Respect to CGC due to m.

From the point of view of communication-induced checkpointing, when a checkpoint is taken which is equivalent to the previous one with respect to some consistent global checkpoint, then no lazy coordination must be started in order to guarantee that checkpoint to be not involved in any Z-cycle (as this is automatically verified). Furthermore, the consistent global checkpoint is automatically advanced by including the taken checkpoint.

From an operational point of view, the equivalence between checkpoints can be detected exploiting dependences between checkpoints established by message exchange.

In the following sections we show a solution for detecting the equivalence on-the-fly which requires to piggyback on any application message one integer plus a vector of n integers (i.e. the asymptotic space-complexity of the control information is $O(n)$). Such information also represents the timestamp associated to a message. The proposed solution is based on the notion of sequence number, as well as on the introduction of the notion of *equivalence number* of a checkpoint.

5.2 Sequence and Equivalence Numbers of a Consistent Global Checkpoint

We suppose process P_i owns two local variables: sn_i and en_i. The variable sn_i stores the sequence number of the current consistent global checkpoint. The variable en_i represents the number of equivalent local checkpoints with respect to the current global checkpoint number (both sn_i and en_i are initialized to zero).

Whenever a checkpoint $C_{i,x}$ is taken, together with the checkpoint two integers are recorded onto stable storage, namely $C_{i,x}.sn$ and $C_{i,x}.en$, which represent, respectively, the consistent global checkpoint number and the equivalence number of P_i at the time $C_{i,x}$ is taken. The pair of integers $< sn_i, en_i >$ is also called local *index* of P_i, hence, to each checkpoint $C_{i,x}$ is associated the index $< C_{i,x}.sn, C_{i,x}.en >$.

In the remainder of this chapter, the notation $C_{i,x}(< sn, en >)$ is used whenever the checkpoint $C_{i,x}$ of P_i whose index is $< sn, en >$ has to be identified. Therefore, $C_{i,x}(< sn_i, en_i >)$ always identifies the last checkpoint taken by P_i. Furthermore, the notation $I_{i,x}(< sn, en >)$ identifies the checkpoint interval $I_{i,x}$ starting after $C_{i,x}(< sn, en >)$ is taken. To the first checkpoint $C_{i,1}$ of process P_i the index $< 0, 0 >$ is assigned.

Similarly to the classical sequence number based approach [12, 36], forced checkpoints are taken as follows. Each application message sent by P_i piggybacks the current sn_i value. Whenever a message m arrives at P_i in $I_{i,x}$ such that $m.sn > sn_i$ then the local index of P_i is set to $< m.sn, 0 >$ and a forced checkpoint $C_{i,x+1}$ is taken with index $< m.sn, 0 >$. The updating rule of the local index is such that whenever the sequence number sn_i is increased the equivalence number en_i is set to zero. For any pair of checkpoints $C_{i,x}(< sn, 0 >)$ and $C_{j,y}(< sn, 0 >)$ the following relation holds:

$$(\neg(C_{i,x} \prec_{ckpt} C_{j,y})) \wedge (\neg(C_{i,x} \prec_{ckpt} C_{j,y}))$$

hence, checkpoints with the same sequence number and equivalence number equal to zero are members of a consistent global checkpoint.

Each time a basic checkpoint $C_{i,x+1}$ is taken which is equivalent to its predecessor $C_{i,x}$ with respect to some consistent global checkpoint then, by Lemma 5.1.1 such checkpoint is not involved in any Z-cycle. Therefore, the lazy coordination for determining a consistent global checkpoint containing $C_{i,x+1}$ must be started only if the equivalence is not verified.

Denoting with $CGC(C_{i,x})$ the global consistent checkpoint containing $C_{i,x}$, then, upon the scheduling of a basic checkpoint $C_{i,x+1}$, the local index is updated according to the following rule:

if $C_{i,x} \stackrel{CGC(C_{i,x})}{\equiv} C_{i,x+1}$
then $en_i \leftarrow en_i + 1$
else $sn_i \leftarrow sn_i + 1;\ en_i \leftarrow 0;$

In other words, if $C_{i,x}$ is equivalent to $C_{i,x+1}$ with respect to the consistent global checkpoint $CGC(C_{i,x})$, then the same sequence number of $C_{i,x}$ and an equivalence number increased by one are assigned as index to $C_{i,x+1}$. Otherwise, the index of $C_{i,x+1}$ becomes $< C_{i,x}.sn + 1, 0 >$. Note that whenever the sequence number is not increased, no lazy coordination inducing forced checkpoints in other processes in started (as forced checkpoints are triggered basing only on the comparison between the sequence number piggybacked on an arriving message and the local sequence number of the recipient process).

The increasing of the sequence numbers is the basis for the communication-induced coordination (as in the protocols in [12, 36]), whereas the increasing of equivalence numbers is used to spread the knowledge on the automatic advancement of the consistent global checkpoint due to the occurrence of equivalences between local checkpoints of a process. Whenever a process learns that the consistent global checkpoint has moved, then it may track new equivalences with respect to the advanced global checkpoint. The next section is devoted to the explanation of latter concept and to the description of a mechanism to track the dynamically created equivalences on-the-fly.

5.2.1 Tracking Equivalent Checkpoints

Let consider the three processes scenario in Figure 5.3.a. There exists a consistent global checkpoint $CGC = \{C_{1,x_1}, C_{2,x_2}, C_{3,x_3}\}$ formed by checkpoints with index $< sn, 0 >$. Checkpoint C_{2,x_2+1} is equivalent to C_{2,x_2} with respect to CGC as there does not exist any message m which has been sent from the right end side of CGC and has been received by P_2 before C_{2,x_2+1} is taken. This equivalence generates a new consistent global checkpoint $CGC' = \{C_{1,x_1}, C_{2,x_2+1}, C_{3,x_3}\}$. Figure 5.3.b shows that in the progress of the execution process P_1 takes the checkpoint C_{1,x_1+1}. Such a checkpoint is not equivalent to C_{1,x_1} with respect to CGC due to the presence of the message m which establishes the following relation $C_{2,x_2} \prec_{ckpt} C_{1,x_1+1}$. However, C_{1,x_1+1} is equivalent to C_{1,x_1} with respect to CGC'. This means that the equivalence between C_{2,x_2} and C_{2,x_2+1}, allowing the consistent global checkpoint to advance from CGC to CGC', also permits the equivalence to exist between C_{1,x_1+1} and C_{1,x_1} with respect to CGC', allowing thus to advance the consistent global checkpoint from CGC' to CGC''. In what follows it is shown how equivalence numbers can be used in order to let P_2 track the equivalence between C_{2,x_2} and C_{2,x_2+1}, and then let P_1 track the advancement of the consistent global checkpoint from CGC to CGC'.

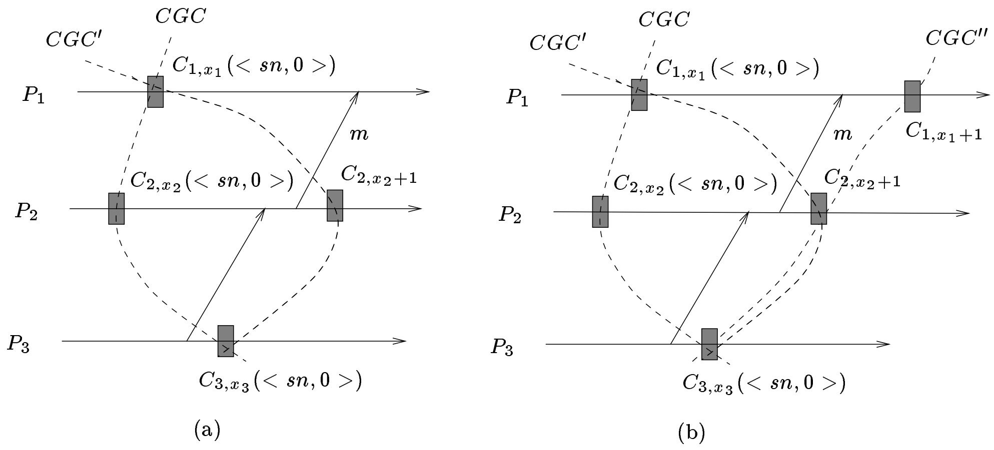


Figure 5.3: An Example of Equivalence Between Checkpoints Generated by the Advancement of the Consistent Global Checkpoint.

Let process P_i be endowed with 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 rules:

- 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, 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, respectively.

Let us remark that the set $\cup_{\forall j} C_{j,x_j}(< sn, EQ_i[j] >)$ is a consistent global checkpoint (a formal proof of this property is given in Theorem 6.2.4). So, to the knowledge of P_i, the vector EQ_i actually identifies the most recent consistent global checkpoint with sequence number sn_i.

Upon the arrival of a message m at P_i in the checkpoint interval $I_{i,x}(< sn_i, en_i >)$ one of the following three cases is true:

(1) $(m.sn < sn_i)$ or $((m.sn = sn_i)$ and $(\forall j\ m.EQ[j] < EQ_i[j]))$;
in this case m has been sent from the left side of the consistent global checkpoint $\cup_{\forall j} C_{j,x_j}(< sn, EQ_i[j] >)$;

(2) $(m.sn = sn_i)$ and $(\exists j :\ m.EQ[j] \geq EQ_i[j])$;
in this case, m has been sent from the right side of the consistent global checkpoint $\cup_{\forall j} C_{j,x_j}(< sn, EQ_i[j] >)$;

(3) $(m.sn > sn_i)$;
in this case m has been sent from the right side of a consistent global checkpoint whose sequence number is unknown by P_i (i.e., P_i is not aware of that consistent global checkpoint).

As explained in previous section, a message m falling in case (3) directs P_i to take a forced checkpoint $C_{i,x+1}$ with index $< m.sn, 0 >$ (note that after taking the forced checkpoint, message m falls in case (2) with respect to the checkpoint interval $I_{i,x+1}$).

When a forced checkpoint is taken upon the receipt of a message m, process P_i has no possibility to select an index for that checkpoint as the index $< m.sn, 0 >$ must be assigned to it. Therefore, the only interesting cases for tracking the equivalence, and thus increasing the equivalence number, are (1) and (2).

When the basic checkpoint $C_{i,x+1}$ is scheduled, P_i falls in one of the following two alternatives:

(i) If no message is received in $I_{i,x}(< sn, en >)$ that falls in case (2), then $C_{i,x} \overset{\cup_{\forall j} C_{j,x_j}(<sn, EQ_i[j]>)}{\equiv} C_{i,x+1}$. This equivalence can be tracked by a process using its local context at the time the checkpoint $C_{i,x+1}$ is scheduled. Thus $C_{i,x+1}.sn \leftarrow C_{i,x}.sn$ and $C_{i,x+1}.en \leftarrow C_{i,x}.en + 1$. The equivalence $C_{2,x_2} \overset{CGC(C_{2,x_2})}{\equiv} C_{2,x_2+1}$, shown in Figure 5.4, is an example of such a case;

(ii) If there exists at least a message m received in $I_{i,x}(< sn, en >)$ which falls in case (2), one checkpoint belonging to the consistent global checkpoint $\cup_{\forall j} C_{j,x_j}(< sn, EQ_i[j] >)$ precedes $C_{i,x+1}$ through the $\prec_{ckpt}$ relation. Such a situation is shown in Figure 5.4 where $\cup_{\forall j} C_{j,x_j}(< sn, EQ_1[j] >) = \{C_{1,x_1}, C_{2,x_2}, C_{3,x_3}\}$, and due to m, $C_{2,x_2} \prec_{ckpt} C_{1,x_1+1}$. The consequence is that process P_i cannot determine, at the time the checkpoint

$C_{i,x+1}$ is scheduled, if $C_{i,x}$ is equivalent to $C_{i,x+1}$ with respect to some consistent global checkpoint.

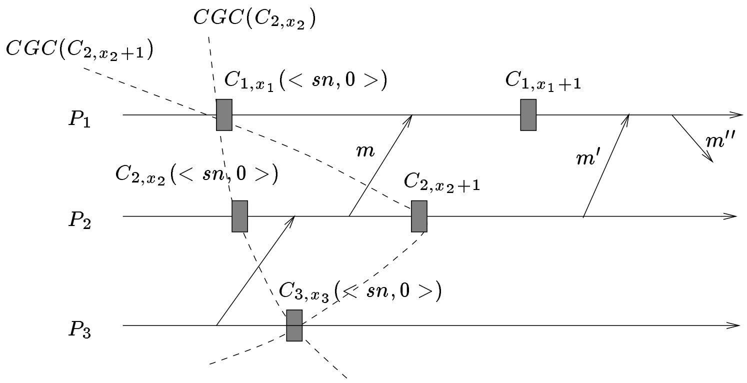


Figure 5.4: Upon the Receipt of m', P_1 Detects $C_{1,x_1} \overset{CGC(C_{2,x_2})}{\equiv} C_{1,x_1+1}$.

To solve the problem raised in point (ii), two approaches can be envisaged. If, at the time the basic checkpoint $C_{i,x+1}$ is scheduled, the equivalence between $C_{i,x}$ and $C_{i,x+1}$ is undetermined (case (ii) discussed above) then:

Pessimistic Approach.
Process P_i pessimistically assumes the two checkpoints are not equivalent with respect to any consistent global checkpoint even though this determination could be revealed wrong in the future of the computation. In such a case, upon the taking of $C_{i,x+1}$ the local index is updated as follows $sn_i \leftarrow C_{i,x}.sn + 1$ and $en_i \leftarrow 0$. Figure 5.4 shows a case in which message m' brings the information (encoded in $m'.EQ$) to P_1 that $C_{2,x_2} \overset{CGC(C_{2,x_2})}{\equiv} C_{2,x_2+1}$ and that the consistent global checkpoint was advanced including C_{2,x_2}. In such a case, P_1 can determine C_{1,x_1} is equivalent to C_{1,x_1+1} with respect to $CGC(C_{2,x_2+1})$ corresponding to the set $\{C_{1,x_1}, C_{2,x_2+1}, C_{3,x_3}\}$. 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 $I_{i,x}$ starts and $equiv_i$ is set to $FALSE$ whenever a message m such that $m.sn = sn$ is received in $I_{i,x}$. Upon scheduling $C_{i,x+1}$, if $\neg(equiv_i)$ then the index $< sn + 1, 0 >$ is assigned to $C_{i,x+1}$. This implementation [44, 45] does not require to piggyback the vector EQ.

Optimistic Approach.
Process P_i assumes *optimistically* (and *provisionally*) that $C_{i,x}$ is equivalent to $C_{i,x+1}$. So the index of $C_{i,x+1}$ becomes $< C_{i,x}.sn, C_{i,x}.en + 1 >$.

As provisional indices cannot be propagated in the system (this would lead to a non consistent view of processes regarding information on other processes spread through causality), if at the time of the first send event occurring after $C_{i,x+1}$ is taken the equivalence is still undetermined, then the index of $C_{i,x+1}$ is re-updated as $< C_{i,x}.sn + 1, 0 >$ (thus, $sn_i \leftarrow sn_i + 1$, $en_i \leftarrow 0$, and $\forall j : EQ_i[j] \leftarrow 0$). Otherwise, the provisional index becomes permanent.

Figure 5.4 shows a case in which $C_{1,x_1} \overset{CGC(C_{2,x_2})}{\equiv} C_{1,x_1+1}$ and this is detected by P_i before sending m''. In this case the index of C_{i,x_i+1} becomes permanent upon the send of m''.

In the next section a communication-induced checkpointing protocol is described which follows the optimistic approach.

5.2.2 Sequence and Equivalence Number Based Protocol (SENBP)

In this section a Sequence and Equivalence Number Based Protocol (SENBP) following the optimistic approach in the detection of equivalent checkpoints is presented. The protocol can be sketched by three rules: `take-basic`, `take-forced` and `send-message`.

Take-Basic Rule.
Whenever a basic checkpoint is scheduled, the local sequence number is not updated by optimistically assuming that each basic checkpoint is equivalent to the previous one. Hence, each process P_i is endowed with 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` :
When a basic checkpoint is scheduled:
$en_i \leftarrow en_i + 1$;
Take a checkpoint with a provisional index $< sn_i, en_i >$;
$provisional_i \leftarrow TRUE$;

Send-Message Rule.
Due to the presence of provisional indices caused by the existence of non resolved equivalences, the protocol needs a 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` are the following:

`send-message` :
    Before sending a message m in $I_{i,x}$:
        if $\neg(after\_first\_send_i)$ and $provisional_i$
        then
            if $\neg(C_{i,x-1} \overset{\cup_{\forall j} C_{j,x_j}(<sn, EQ_i[j]>)}{\equiv} C_{i,x})$
            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 Rule.
The last rule of the protocol `take-forced` refines the corresponding rules in protocols in [12, 36] by using a simple observation.

Observation 5.2.1
Upon the receipt of a message m in $I_{i,x}(< sn_i, en_i >)$ such that $m.sn > sn_i$, there is no reason to take a forced checkpoint if there has been no send event in $I_{i,x}(< sn_i, en_i >)$.

Indeed, no $\prec_{ckpt}$ relation can be established between the last checkpoint $C_{i,x}(< sn_i, en_i >)$ and any checkpoint with sequence number $m.sn$ and, thus, the index of $C_{i,x}(< sn_i, en_i >)$ can be replaced permanently with the index $< m.sn, 0 >$. As discussed in Section 3.2 of Chapter 3, Observation 5.2.1 has been used for the first time by Wang in [64] to develop the Fixed-Dependency-After-Send protocol. The `take-forced` rule is as follows:

`take-forced` :
    Upon the receipt of a message m in $I_{i,x}(< sn_i, en_i >)$:
        case
            $sn_i < m.sn$ **and** $after\_first\_send_i \rightarrow$ /* part (a) */
                $sn_i \leftarrow m.sn$; $en_i \leftarrow 0$;
                a forced checkpoint $C_{i,x+1}(< m.sn, 0 >)$ is taken
                and its index is permanent;
                $provisional_i \leftarrow FALSE$;
                $\forall j\ EQ_i[j] \leftarrow m.EQ[j]$;
            $sn_i < m.sn$ **and** $\neg(after\_first\_send_i) \rightarrow$ /* part (b) */
                $sn_i \leftarrow m.sn$; $en_i \leftarrow 0$;

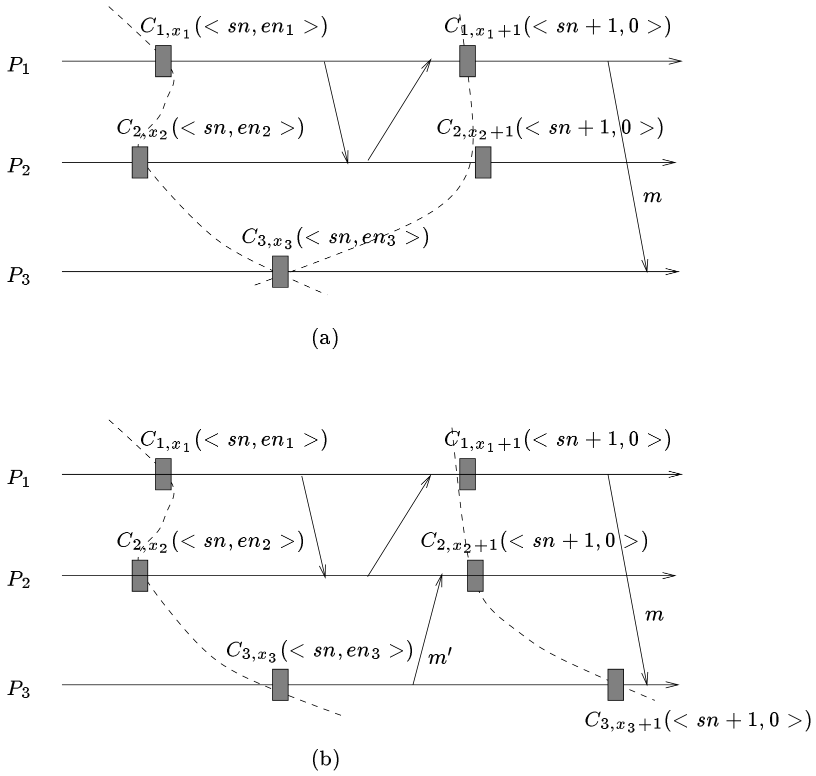


Figure 5.5: Upon the Receipt of m, C_{3,sn,en_3} can be Part of a Consistent Global Checkpoint with Sequence Number $sn+1$ (a); C_{3,sn,en_3} cannot belong to a Consistent Global Checkpoint with Sequence Number $sn+1$ (b).

the index of the last checkpoint $C_{i,x}$ is replaced
permanently with $< m.sn, 0 >$;
$provisional_i \leftarrow FALSE$;
$\forall j \ EQ_i[j] \leftarrow m.EQ[j]$;
$sn_i = m.sn \rightarrow$ /* 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 5.5.a, the local checkpoint C_{3,x_3} can belong to the consistent global checkpoint with sequence number $sn+1$ and formed by $\{C_{i,x_1+1}, C_{2,x_2+1}, C_{3,x_3}\}$ (so the index $< sn, en_3 >$ can be replaced with $< sn+1, 0 >$). On the contrary, due to the send event of message m' in $I_{3,x_3}(< sn, en_3 >)$ depicted in Figure 5.5.b, a forced checkpoint C_{3,x_3+1} with index $< sn+1, 0 >$ has to be taken upon the receipt of message m.

Part (b) of `take-forced` decreases the number of forced checkpoints compared to the protocols in [12, 36]. The **then** alternative of `send-message` 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.

5.2.3 A Modification of SENBP (M-SENBP) for the Case of Periodic Basic Checkpoints

Performance of the SENBP protocol, in terms of checkpointing overhead imposed to the computation can be improved in the case basic checkpoints are scheduled on a periodic basis by including in the protocol the technique of skipping basic checkpoints presented by Manivannan and Singhal [36]. They have shown that 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 endows 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 the `take-basic` rule including the skipping technique is as follows:

`take-basic` :
    When a basic checkpoint is scheduled:
      if $skip_i$
      then $skip_i \leftarrow FALSE$
      else $en_i \leftarrow en_i + 1$;
          Take a checkpoint with a provisional index $< sn_i, en_i >$;
          $provisional_i \leftarrow TRUE$;

The checkpointing protocol embedding the skipping technique will be referred to has Modified SENBP (M-SENBP). An implementation of M-SENBP is described below.

5.2.4 An Implementation of M-SENBP

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**.

$Present_i[j]$ represents the maximum equivalence number en_j sent by P_j and received in the current checkpoint interval by P_i, and piggybacked on a message that falls in the case 2 of Section 5.2.1. 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 consistent global checkpoint (case 2 of Section 5.2.1) *currently* seen by P_i.

In Figure 5.6 and in Figure 5.7 the behavior of process P_i is shown (the procedures and the message handler are executed in atomic fashion). The shown implementation assumes that there exist at most one provisional index in each process. So each time two successive provisional indices are detected, the first index is permanently replaced with $< sn_i + 1, 0 >$.

5.2.5 Correctness Proof

In what follows, a formal proof is given that at any time under M-SENBP the set $\cup_{\forall j} C_{j,x_j}(< sn, EQ_i[j] >)$ is a consistent global checkpoint (note that the proof holds also in the case the skipping technique of basic checkpoints is removed by the protocol). At this aim, let us introduce the following simple observations and lemmas:

Observation 5.2.2
For any checkpoint $C_{i,,x}(< sn, 0 >)$, there does not exist any message m with $m.sn \geq sn$ such that $receive(m) \in I_{i,x-\epsilon}$ with $\epsilon > 0$. This observation derives from rule `take-forced` *of M-SENBP when considering $C_{i,x}(< sn, 0 >)$ is the first checkpoint with sequence number sn.*

Observation 5.2.3
For any message m sent by P_i in $I_{i,x}(< sn, en >)$ or in a later checkpoint interval, then $m.sn \geq sn$. This observation derives from the rule `send-message` *of M-SENBP.*

Lemma 5.2.4
For any pair of checkpoints $(C_{i,x}(< sn, en >), C_{j,y}(< sn, 0 >))$ the following predicate holds:

$$\neg(C_{i,x} \prec_{ckpt} C_{j,y})$$

```
init P_i:
sn_i := 0;  en_i := 0;
after_first_send_i := FALSE;  skip_i := FALSE;  provisional_i := FALSE;
∀h EQ_i[h] := 0;  ∀h Past_i[h] := −1;  ∀h Present_i[h] := −1;

when message m arrives at P_i from P_j:
if m.sn > sn_i then                      % P_i is not aware of the sequence number m.sn %
begin
  if after_first_send_i then
  begin
    take a checkpoint;                                  % taking a forced checkpoint %
    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 %
       ∀h if Past_i[h] < m.EQ[h] then Past_i[h] := −1;
     end;
process the message m;
```

Figure 5.6: M-SENBP - Part A.

Proof (By Contradiction)
Suppose by the way of contradiction, that $C_{i,x} \prec_{ckpt} C_{j,y}$. In this case, there exits a message m sent by P_i after $C_{i,x}$ is taken and received by P_j before taking $C_{j,y}$. Due to Observation 5.2.3 $m.sn \geq sn$, therefore, due to Observation 5.2.2, it cannot be received by P_j before $C_{j,y}(< sn, 0 >)$. Thus the assumption is contradicted and the claim follows. *Q.E.D.*

Lemma 5.2.5
Let i, j and k be three integers. At any given time for a pair of checkpoints $(C_{i,x}(< sn, EQ_k[i] >), C_{j,y}(< sn, EQ_k[j] >))$ the following predicate holds:

$$\neg(C_{i,x} \prec_{ckpt} C_{j,y})$$

Proof (By Contradiction)
Suppose by the way of contradiction that $R \equiv C_{i,x} \prec_{ckpt} C_{j,y}$ holds due to a message m. Four cases have to be considered:

```
when P_i sends data to P_j:
if provisional_i ∧ (∃h : Past_i[h] > −1)   % last ckpt not equivalent to previous one %
then
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 provisional_i then                        % two successive provisional indices %
  if (∃h : Past_i[h] > −1)                 % last ckpt not equivalent to previous one %
then
  begin
    ∀h past_i[h] := −1;
    sn_i := sn_i + 1;  en_i := 0;
    assign the index < sn_i, en_i > to the last checkpoint;      % permanent index %
    ∀h EQ_i[h] := 0;
  end
  else ∀h Past_i[h] := Present_i[h];        % last ckpt is equivalent to 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 checkpoint;
provisional_i := TRUE;                                  % the index is provisional %
∀h Present_i[h] := −1;
after_first_send_i := FALSE;
```

Figure 5.7: M-SENBP - Part B.

1) if $i = j$ predicate R contradicts Definition 1.3.1;

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

   - if $EQ_i[j] = 0$, Lemma 6.2.2 is contradicted;
   - if $EQ_i[j] > 0$ then: (i) $C_{j,y}(< sn, EQ_i[j] >)$ is equivalent to $C_{j,y-1}(< sn, EQ_i[j] - 1 >)$ and (ii) there exists a causal path of messages which brings to P_i the information of that equivalence in the current checkpoint interval $I_{i,x}(< sn, EQ_i[i] >)$.

     From Definition 5.1, $C_{j,y}(< sn, EQ_i[j] >)$ can be equivalent to $C_{j,y-1}(< sn, EQ_i[j] - 1 >)$ 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 5.8.a.

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

- if $EQ_j[j] = 0$, Lemma 6.2.2 is contradicted;
- if $EQ_j[j] > 0$ then $C_{j,y}(< sn, EQ_j[j] >)$ is equivalent to $C_{j,y-1}(< sn, EQ_j[j] - 1 >)$. Let en_i be the value stored in $EQ_j[i]$. From the rule `send-message` of M-SENBP, an equivalence number is stored in EQ only when the index is permanent. This means that in the interval of events between the checkpoint $C_{j,y}(< sn, EQ_j[j] >)$ and the first send event of a message m', there must exist a causal path of messages starting after a checkpoint $C_{i,x+\epsilon}(< sn, en >)$ (with $en > en_i$) and ending in $I_{j,y}(< sn, EQ_j[j] >)$ before the sending of m'. In such a case the previous equivalence holds. Due to the rules to update the vector EQ, after the receipt of the last message of that causal path, 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 5.8.b.

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

- if $EQ_k[j] = 0$, Lemma 6.2.2 is contradicted;
- if $EQ_k[j] > 0$ then $C_{j,y}(< sn, EQ_k[j] >)$ is equivalent to $C_{j,y-1}(< sn, EQ_k[j]-1 >)$. Let en_i be the value stored in $EQ_k[i]$. Due to the initial assumption, in order to ensure that the equivalence is verified there must exist (i) a causal path of messages μ' starting after a checkpoint $C_{i,x+\epsilon}(< sn, en >)$ (with $en > en_i$) and ending in $I_{j,y}(< sn, EQ_k[j] >)$ and (ii) a causal path of messages 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 5.2), 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 5.8.c.

In all cases the assumption that the predicate $\mathcal{R}$ holds leads to a contradiction. Then the claim follows. *Q.E.D.*

Theorem 5.2.6
At any given time the set $S = \cup_{\forall j} C_{j,x_j,}(< sn, EQ_i[j] >)$ is a consistent global checkpoint.

Proof
The proof follows from Lemma 6.2.3 applied to any distinct pair of checkpoints in S and from the Definition 1.3.2. *Q.E.D.*

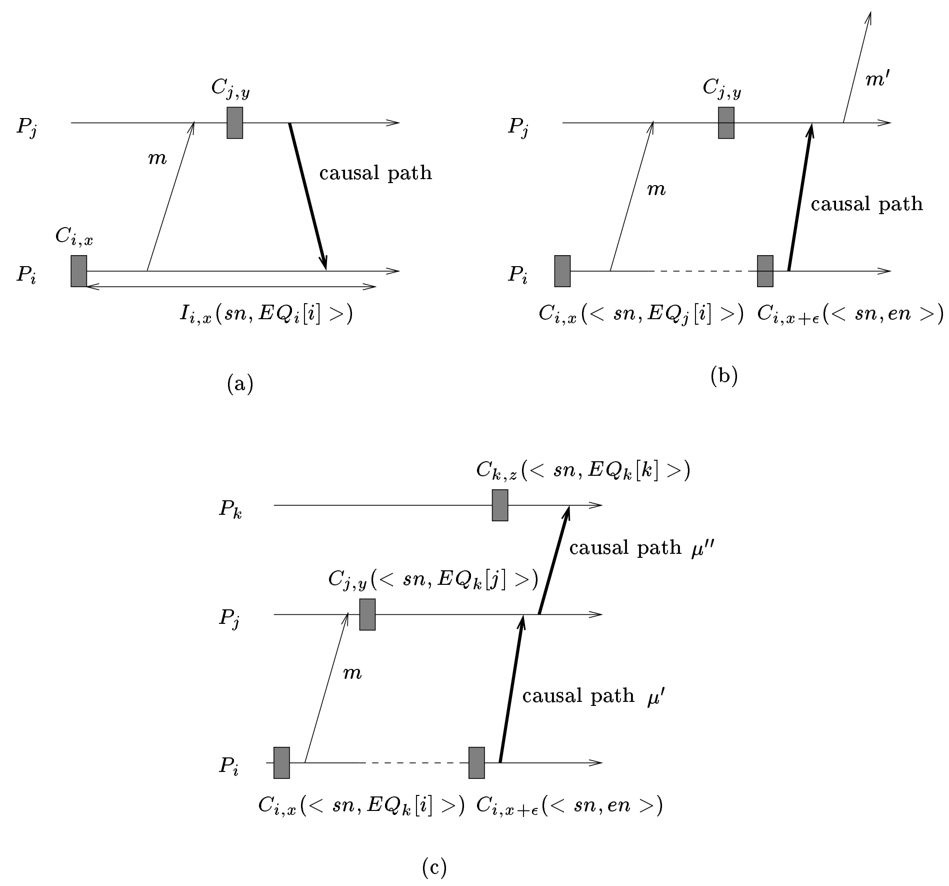


Figure 5.8: Proof of Lemma 6.2.3

Note that each local checkpoint produced by the protocol belongs to, at least, one consistent global checkpoint. In particular, $C_{i,x_i}(< sn, en >)$ belongs to all consistent global checkpoints having sequence number sn' such that $C_{i,x_i-1}.sn < sn' \leq sn$.

5.3 Performance Measures: a Case Study in the Context of Rollback Recovery

In this section, a performance comparison between M-SENBP and previous protocols is presented in the context of rollback recovery. Performance data are obtained through simulation. Performance measures are related to the overhead imposed by the protocols during failure free computation and to the extent of rollback in case of failure.

5.3.1 The Simulation Model

The simulation compares the protocol in [12] (hereafter BCS), the protocol in [36] (hereafter MS) and M-SENBP ([1]) in an *uniform point-to-point* environment in which each process can send a message to any other 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 protocols.

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 intervals (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 schedules 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 a protocol 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 M-SENBP and MS.

Each simulation run contains 8000 message receives 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.

[1]Simulation results of the protocol in [28] are not reported as for the considered environment they are quite similar to those of BCS.

5.3.2 Results of the Experiments

Total Number of Checkpoints

Figure 5.9 shows the ratio *Tot* of MS and M-SENBP 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 M-SENBP saves from 2% to 10% of checkpoints compared to MS. As the value of *bcf* is higher than 1.0%, MS and M-SENBP 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 M-SENBP shown in Figure 5.11. For small values of *bcf*, M-SENBP 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 5.10 and Figure 5.12) 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*, M-SENBP saves from 7% to 18% checkpoints compared to MS, and induces up to 77% less than MS.

Heterogeneous Environment

The low values of F shown by M-SENBP 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 very high checkpointing overhead using either MS or BCS.

In Figure 5.13, 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 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 5.14 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 M-SENBP got the maximum gain (see Figure 5.13). 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 M-SENBP is constantly around 30% less than that of MS.

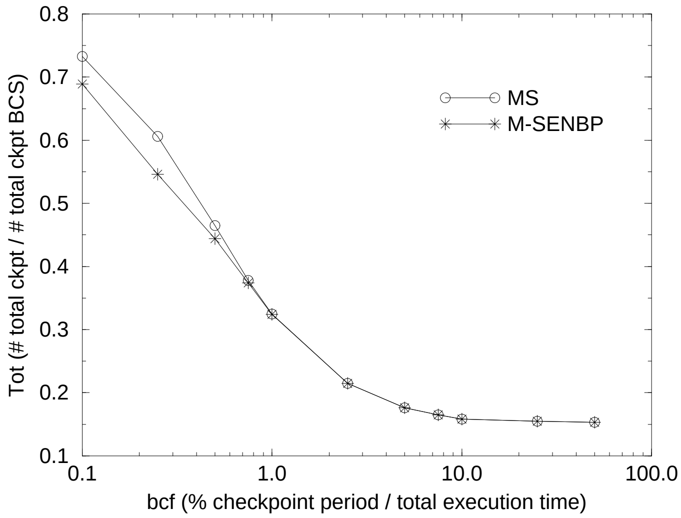


Figure 5.9: *Tot* vs. *bcf* in the *Uniform Point-to-Point* Environment ($B = 0$).

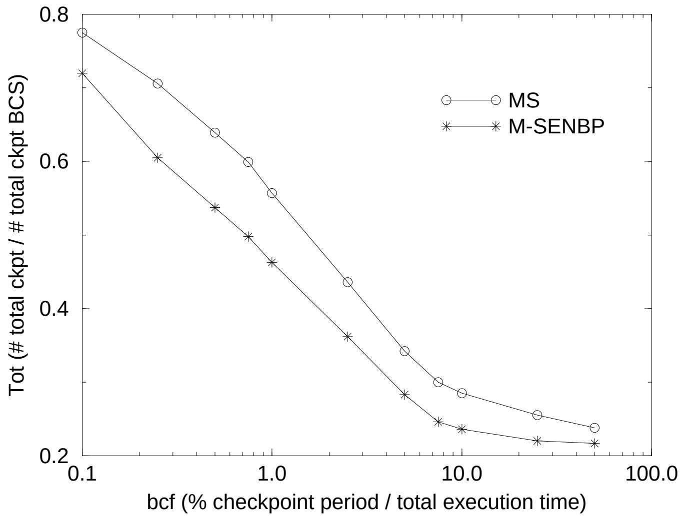


Figure 5.10: *Tot* vs. *bcf* in the *Bursted Point-to-Point* Environment ($B = 2$).

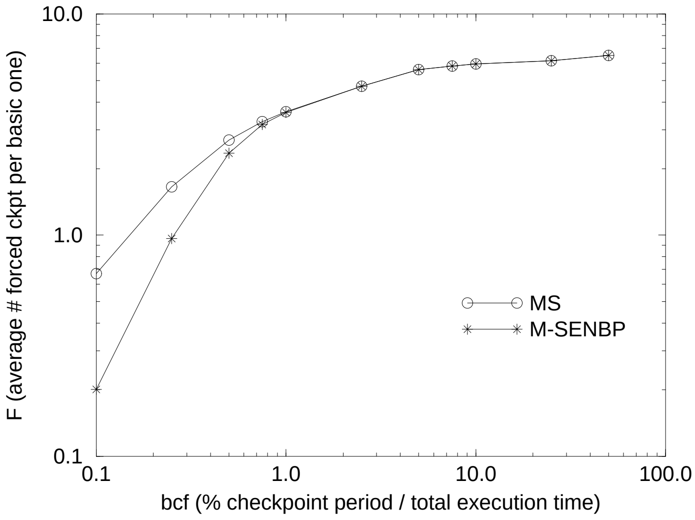


Figure 5.11: F vs. *bcf* in the *Uniform Point-to-Point* Environment ($B = 0$).

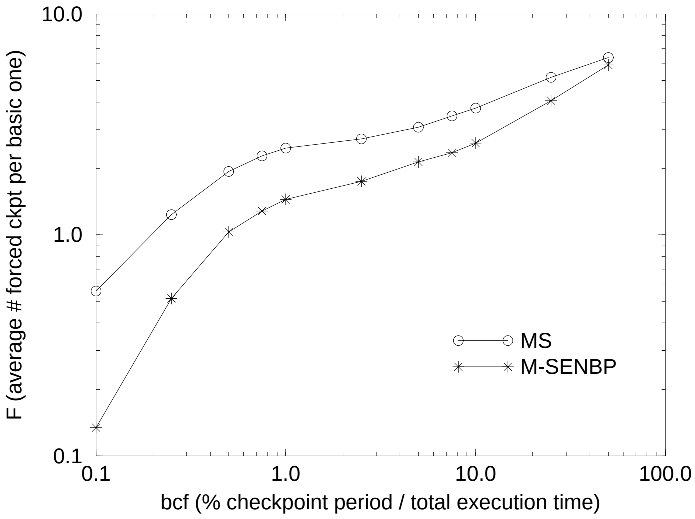


Figure 5.12: F vs. *bcf* in the *Bursted Point-to-Point* Environment ($B = 2$).

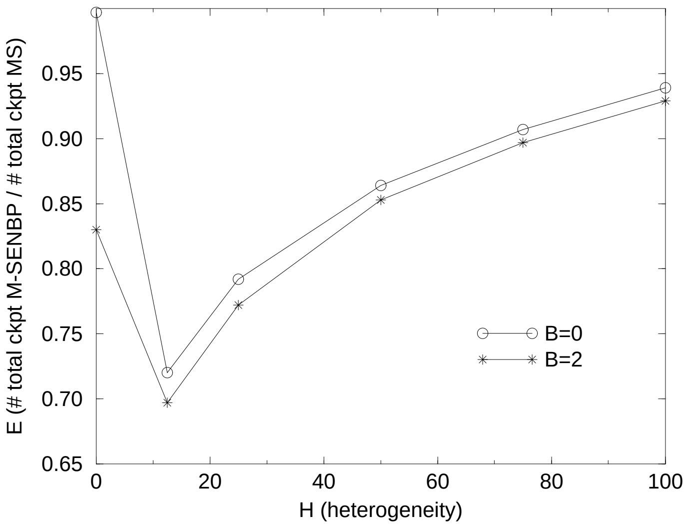


Figure 5.13: E vs. *Heterogeneity* in both the *Uniform Point-to-Point* Environment ($B = 0$) and the *Bursted Point-to-Point* Environment ($B = 2$).

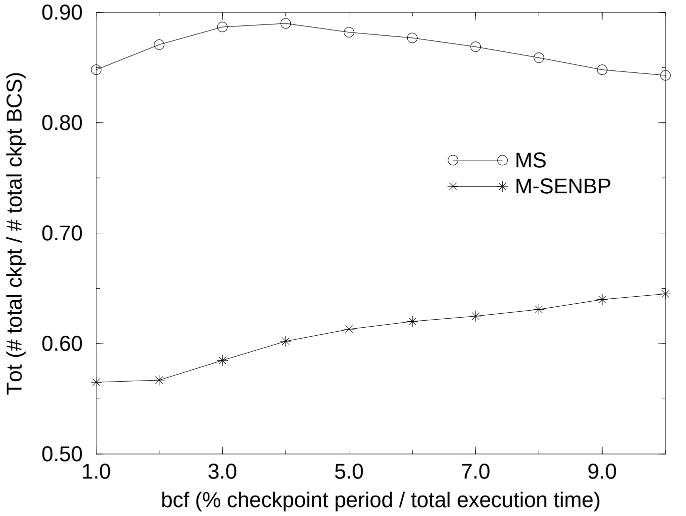


Figure 5.14: *Tot* vs. *bcf* of the Slowest Processes in a *Bursted Point-to-Point* Environment ($B = 2$) with $H = 12.5\%$

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 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 consistent global checkpoint of the application associated to the sequence number of the last taken checkpoint of the failed process and counts the number of events undone to rollback to that global checkpoint.

The consistent global checkpoint to which the application should be rolled back is build as follows: the failed process restarts its computation from its last checkpoint, say A, forcing the other processes to rollback to the global checkpoint to which A belongs, say $CGC(A)$.

During the rollback phase, in MS and M-SENBP, if the checkpoint with sequence number $A.sn$ 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 M-SENBP, if the index of A is not permanent, the index is replaced with $< sn + 1, 0 >$ prior the rollback. Otherwise, each process rolls back to the most recent checkpoint with sequence number sn (i.e., the one with the higher equivalence number).

Simulation experiments were conducted in the uniform point-to-point environment. In Figure 5.15, UE as a function of bcf is shown. Given the large checkpointing overhead of BCS during failure-free computations (see Figure 5.9), the consistent global checkpoint to which the application is rolled back is closest, on the average, to the end of the computation compared to M-SENBP and MS. As an example in the case of $bcf = 2.5\%$ (i.e., 40 basic checkpoints for each process), M-SENBP and MS takes about 80% less checkpoints compared to BCS as depicted in Figure 5.9 while BCS's UE is 70% less than M-SENBP and MS (see Figure 5.15). 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 5.16 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 M-SENBP, while M-SENBP saves about 35% of forced checkpoints compared to MS (see Figure 5.14).

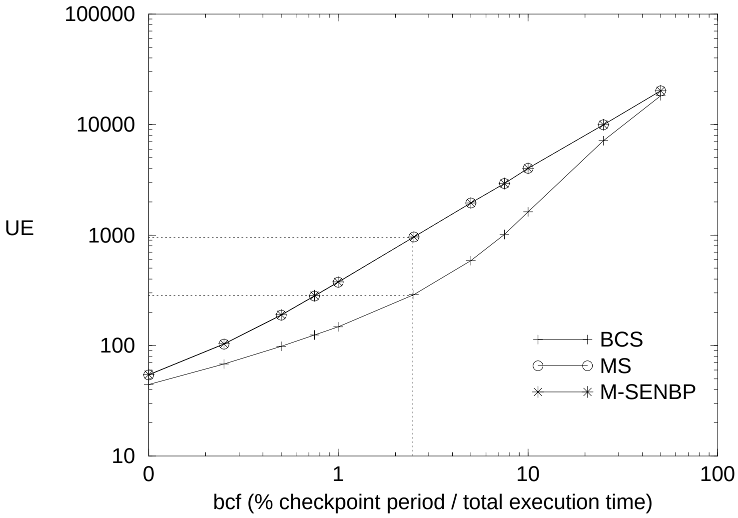


Figure 5.15: *UE* vs. *bcf* in the *Uniform Point-to-Point* Environment ($B = 0$ and $H = 0\%$).

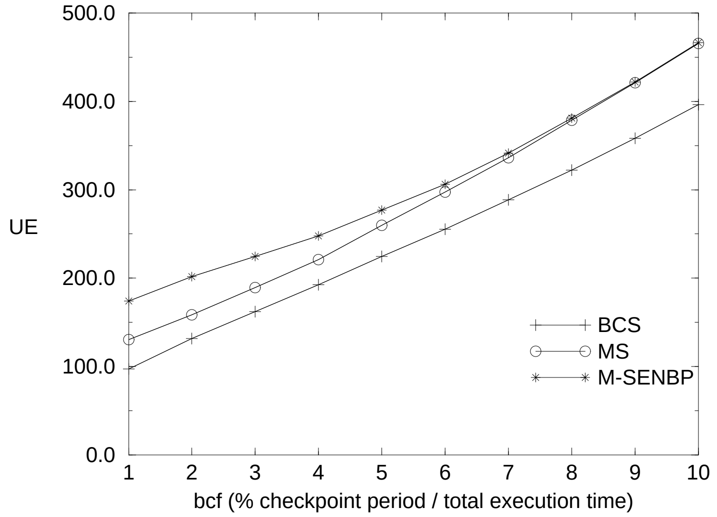


Figure 5.16: *UE* vs. *bcf* in the *Uniform Point-to-Point* Environment (B=0 and $H = 12.5\%$).

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. We study the behavior of the function OH in BCS, MS and M-SENBP by varying the number of failures occurring in the computation.

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 computation and T_{ckpt} is the average time spent for 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 5.17 shows *OH/(OH of BCS)* vs. the number of failures. These plots were obtained in an 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 M-SENBP is widely less than the one of BCS and MS. The total overhead imposed by the three protocols becomes comparable only for a very high failure rate (in the order of 10^2 failures per an execution of 80000 events) which is extremely unlikely in real distributed systems.

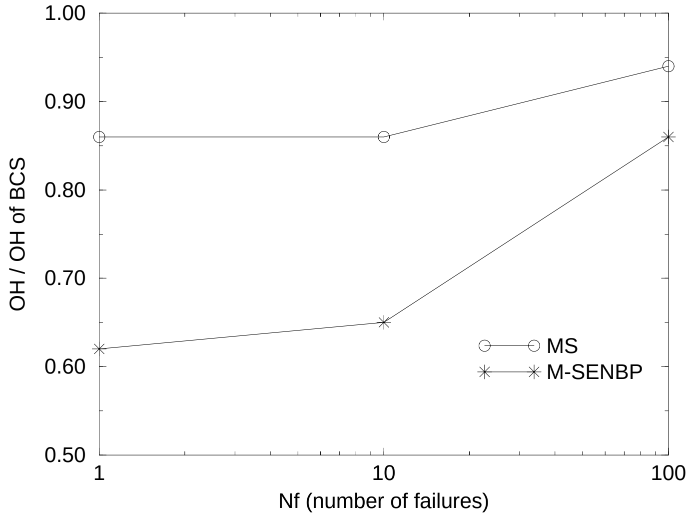


Figure 5.17: *OH/(OH of BCS)* vs. N_f in the *Uniform Point-to-Point* Environment (B=0 and $H = 12.5\%$).

Chapter 6

Virtual Precedence Accordant Protocols

The aim of this chapter is to study the structure of a checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ of a distributed computation in order to identify particular sub-patterns whose absence implies (and is implied by) the absence of Z-cycles. More technically, a characterization of the $\mathcal{NZC}$ property is introduced, which was previously an open problem.

The particular sub-pattern identified in this study has been named *Core Z-Cycle* (CZC)[1]. The derived characterization is based on a property which stipulates that there is no core Z-cycle in the computation ($\mathcal{NCZC}$ property). A Core Z-cycle is a Z-cycle with several constraints on its structure.

More precisely, the following result is proved:

- $\mathcal{NZC} \Leftrightarrow \mathcal{NCZC}$ (i.e., the characterization theorem). This is obtained by introducing successive embedded subsets of Z-cycles, namely, *elementary Z-cycles*, *prime Z-cycles* and *core Z-cycles*, whose members satisfy progressively stronger constraints on their checkpoint and communication pattern structure.

This result has been obtained thanks to the introduction of concatenation relations on message chains and checkpoints that allow to express, in an easy way, the basic structure of checkpoint and communication patterns.

The introduced characterization is important not only from a theoretical point of view but also from a practical one as communication-induced checkpointing protocols ensuring the $\mathcal{NZC}$ property can be derived.

[1]In the rest of the chapter capitalized words denote a *specific* checkpoint and communication pattern, bold capitalized words denote a *set* of checkpoint and communication patterns of the same type and calligraphic style denotes *properties* related to checkpoint and communication patterns.

In particular, members of **CZC** cannot be tracked on-the-fly, however, a particular checkpoint and communication pattern, namely *Suspect Core Z-Cycle* (SCZC) is identified, which represents the causal part of *any* CZC. As it is causal, it is on-the-fly trackable by a communication-induced protocol.

A first communication-induced protocol, namely P1, preventing the formation of SCZCs is introduced. The protocol pushes processes to take forced checkpoints basing on a predicate $\mathcal{P}_1$, and has control information with space-complexity $O(n^2)$. Then, a second protocol, namely P2 is derived. It is based on a predicate $\mathcal{P}_2$ weaker than $\mathcal{P}_1$, and has control information with space-complexity $O(n)$.

These protocols are, to the best of our knowledge, the first $\mathcal{VP}$-accordant protocols explicitly designed to ensure the $\mathcal{NZC}$ property, but not $\mathcal{RDT}$. Performance of the proposed protocols are compared to that of previous ones both through a theoretical analysis and through simulation results. Finally, a distributed protocol for consistent global checkpoint collection is presented. Applications of the checkpointing and global checkpoint collection protocols are finally discussed.

6.1 Preliminary Definitions

This section introduces a formal definition of causal and non-causal message chains (the notion of message chain has already been used in previous chapters, for example under the name of "sequence of messages forming a Z-path", but without a formal definition, which becomes now mandatory) and two concatenation relations on checkpoints and/or chains of messages. These relations express both causal and non-causal ways for checkpoints and/or chains of messages to be combined, and allow synthetic expressions for checkpoint and communication sub-patterns of a checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ of a distributed computation. Finally, the concept of Z-cycle is reformulated using the concatenation relations.

6.1.1 Message Chains

Definition 6.1.1
A message chain is a sequence of messages $\zeta = [m_1, m_2, \ldots, m_\ell]$ *such that*

$$\forall k : 1 \leq k \leq \ell - 1 \Rightarrow (receive(m_k) \in I_{i,x}) \wedge (send(m_{k+1}) \in I_{i,y}) \wedge (x \leq y)$$

In other words, a message chain corresponds to the sequence of messages which establishes a Z-path between two checkpoints. As an example, in Figure 6.1.b we have a message chain formed by messages $[m_1, m_2, m_3]$. A particular case of message chain is the *causal* message chain, in which the receive of a

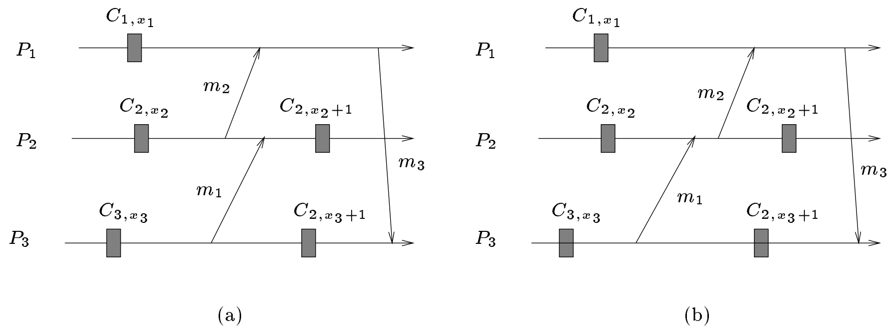


Figure 6.1: (a) a Message Chain Formed by Messages $[m_1, m_2, m_3]$; (b) a Causal Message Chain Formed by Messages $[m_1, m_2, m_3]$.

message always precedes on a process the send of the successive message of the chain. More formally we have:

Definition 6.1.2
A message chain $\zeta = [m_1, m_2, \ldots, m_\ell]$ is causal if

$$\forall k : 1 \leq k \leq \ell - 1 \Rightarrow receive(m_k) \prec_P send(m_{k+1})$$

otherwise, the chain is non causal.

In other words, a causal message chain corresponds to the sequence of messages which establishes a causal Z-path between two checkpoints. It also corresponds to a formal definition of the notion of causal path of messages used in previous chapters. An example of causal message chain is the one formed by messages $[m_1, m_2, m_3]$ in Figure 6.1.b. Recall that a chain with only one message is always causal.

For the sake of clarity, the Greek letter μ indicates a causal message chain. Furthermore we denote with $\zeta.first$ (resp. $\zeta.last$) the first (resp. last) message of a message chain ζ.

$|\zeta|$ denotes the number of messages forming the chain ζ (i.e., the dimension of ζ). In particular, $|\zeta| = \ell$ means that the chain ζ consists of ℓ messages. The operator *minus* is used to denote the removal of a subchain from a chain; for example $\zeta - \zeta.last$ (resp. $\zeta - \zeta.first$) denotes a chain obtained from ζ by removing its last (resp. first) message; $\zeta - \widehat{\zeta}$ denotes a chain obtained by removing the subchain $\widehat{\zeta}$ from ζ where $\widehat{\zeta}$ can be either the initial or the final part of ζ.

Let us finally give the concept of sequence of checkpoint intervals related to a message chain. To each message chain $\zeta = [m_1, m_2, \ldots, m_\ell]$ is associated a sequence of checkpoint intervals $S(\zeta) = (I_{j_1,z_1}, I_{j_2,z_2}, \ldots, I_{j_\ell,z_\ell})$ such that $send(m_i) \in I_{j_i,z_i}$ with $(1 \leq i \leq \ell)$.

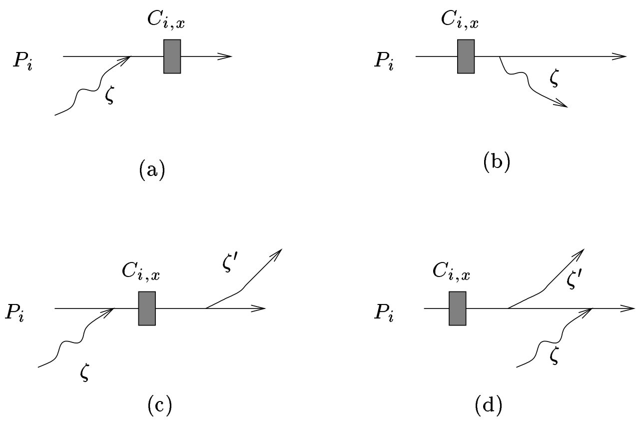


Figure 6.2: Examples of Applying of the Concatenation Relations.

6.1.2 Concatenation Relations

In this section two concatenation relations are introduced, which are used to express and combine, in an easy way, checkpoint and communication patterns.

Causal Concatenation

The causal concatenation, denoted by the symbol $\circ$, can be applied to express the causal combination of two objects (an object can be either a checkpoint or a message chain). Such a concatenation is defined as follows:

Definition 6.1.3
An object $\mathtt{a}$ *is causally concatenated to an object* $\mathtt{b}$, *denoted* $\mathtt{a} \circ \mathtt{b}$, *iff:*

1. $\mathtt{a} = C_{i,x}$, $\mathtt{b} = \zeta$ *and* $\exists v \geq 0 \;\; : \;\; send(\zeta.first) \in I_{i,x+v}$; *or*
2. $\mathtt{a} = \zeta$, $\mathtt{b} = C_{i,x}$ *and* $\exists v > 0 \;\; : \;\; receive(\zeta.last) \in I_{i,x-v}$; *or*
3. $\mathtt{a} = \zeta$, $\mathtt{b} = \zeta'$ *and* $receive(\zeta.last) \prec_P send(\zeta'.first)$.

Examples of causal concatenation are shown in Figures 6.2.a ($\zeta \circ C_{i,x}$), 6.2.b ($C_{i,x} \circ \zeta$) and 6.2.c ($\zeta \circ \zeta'$).

Non-Causal Concatenation

The non-causal concatenation, denoted by the symbol $\bullet$, can be applied to express the non-causal combination of message chains. Such a concatenation is defined as follows:

Definition 6.1.4
A message chain ζ is non-causally concatenated to a message chain ζ' in the checkpoint interval $I_{k,y}$, denoted $\zeta \overset{k,y}{\bullet} \zeta'$, iff the following predicate holds:

$$\begin{aligned} \mathcal{NCC} \;\equiv\; & (receive(\zeta.last) \in I_{k,y}) \wedge \\ & (send(\zeta'.first) \in I_{k,y}) \wedge \\ & (send(\zeta'.first) \prec_P receive(\zeta.last)) \end{aligned}$$

An example of non-causal concatenation $\zeta \overset{i,x}{\bullet} \zeta'$ is shown in Figure 6.2.d. In other words, a message chain ζ is non causally concatenated to a message chain ζ' in the checkpoint interval $I_{k,y}$ if both $send(\zeta'.first)$ and $receive(\zeta.last)$ belong to the same checkpoint interval $I_{k,y}$, with $send(\zeta'.first)$ happening before $receive(\zeta.last)$. For the sake of simplicity of the notation, whenever not necessary the index of the interval is dropped from the non-causal relation.

6.1.3 Concatenation Operators

Let consider two message chains $\zeta = [m_1, \ldots, m_q]$ and $\zeta' = [m'_1, \ldots, m'_p]$. If $\zeta \circ \zeta'$ (or $\zeta \bullet \zeta'$) then by Definition 6.1.1, there exists in the checkpoint and communication pattern of the distributed computation a message chain $\zeta'' = [m_1, \ldots, m_q, m'_1, \ldots, m'_p]$. Therefore, whenever two message chains are concatenated (either causally or non-causally), then there exists in the computation a chain resulting from that concatenation and containing all the messages of the two original chains.

This property allows to use concatenation relations applied to message chains also as concatenation operators generating message chains. For the previous example, the generated message chain is $\zeta'' = \zeta \circ \zeta'$ (or, in case of non-causal concatenation, $\zeta'' = \zeta \bullet \zeta'$).

6.1.4 A Formal Redefinition of the Z-Cycle

By using the concatenation relations, in this section the notion of Z-Cycle (ZC) is reformulated. Basically, a ZC is a checkpoint and communication pattern involving a checkpoint $C_{i,x}$ and a chain $\widehat{\zeta}$ such that:

$$\widehat{\zeta} \circ C_{i,x} \circ \widehat{\zeta}$$

(an example of such a concatenation is shown in Figure 6.3.a) ([2]). However, it is always possible to separate $\widehat{\zeta}$ into two subchains, a causal chain μ and a

[2] For the sake of simplicity, $\widehat{\zeta} \circ C_{i,x} \circ \widehat{\zeta}$ stands for $(\widehat{\zeta} \circ C_{i,x}) \wedge (C_{i,x} \circ \widehat{\zeta})$.

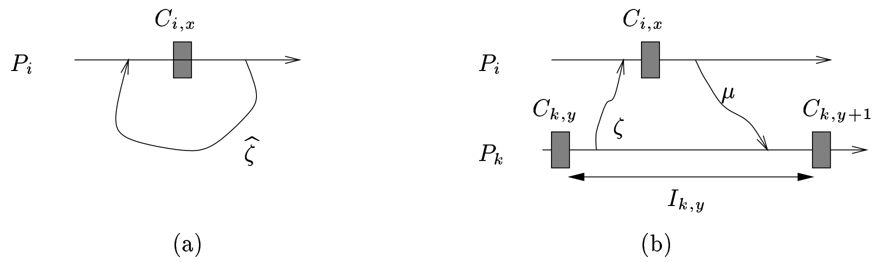


Figure 6.3: The Structure of a Z-Cycle.

message chain ζ such that $\widehat{\zeta} = \mu \overset{k,y}{\bullet} \zeta$, this concatenation is shown in Figure 6.3.b (this is an example of how the non-causal concatenation is used as an operator on message chains). This observation gives rise to the following Z-cycle definition[3]:

Definition 6.1.5
A ZC is a checkpoint and communication pattern $ZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$ *such that:*

$$\zeta \circ C_{i,x} \circ \mu \overset{k,y}{\bullet} \zeta$$

6.2 A Characterization of the No-Z-Cycle Property

To get a characterization of the $\mathcal{NZC}$ property, successive embedded subsets of Z-cycles, namely *Elementary Z-Cycle* (EZC), *Prime Z-Cycle* (PZC) and *Core Z-Cycle* (CZC) are introduced, which are Z-cycles that satisfy progressively stronger constraints on their checkpoint and communication pattern structure as depicted in Figure 6.4. In particular, an EZC is a $ZC(C, \mu \bullet \zeta)$ imposing a constraint on the dimension of ζ. A PCZ is an EZC imposing a constraint on μ and, finally, a CZC is a PZC with a constraint on the sequence of checkpoint intervals associated to ζ.

The lemmas in this section prove the following results:

(i) *if* there exists a ZC in $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ *then* an EZC exists as well;

(ii) *if* there exists an EZC in $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ *then* a PZC exists as well;

(iii) *if* there exists a PZC in $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ *then* a CZC exists as well.

[3]Recall that although the notion of Z-cycle is here expressed in a different way, it is equivalent to the Netzer-Xu formulation.

In other words, each (non-core) Z-cycle involving a checkpoint C embeds a core Z-cycle involving a checkpoint A (see Figure 6.4). This means that **ZC** is empty if, and only if, **CZC** is empty as will be proved in the characterization theorem (Section 6.2.4 of this chapter).

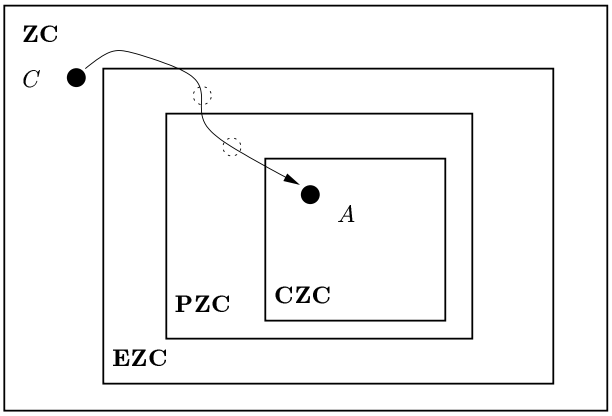


Figure 6.4: Relations Between **ZC**, **EZC**, **PZC** and **CZC**.

6.2.1 Elementary Z-Cycles

This section introduces the notion of Elementary Z-Cycle (EZC). It is interesting because of the result in Lemma 6.2.1 stating that if there is a Z-cycle in a checkpoint and communication pattern of a distributed computation then there exists in that checkpoint and communication pattern an EZC whose size of its chain ζ is smaller than, or equal to, the one of the Z-cycle.

Definition 6.2.1
$ZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$ *is an Elementary Z-Cycle, denoted* $EZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$ *if there does not exist any message chain* ζ' *such that* $|\zeta'| < |\zeta|$ *and* $ZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta')$ *exists.*

Lemma 6.2.1
If *there exists* $ZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$
then *there exists* $EZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta')$ *with* $|\zeta'| \leq |\zeta|$.

Proof
Let us consider $ZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$, if $|\zeta| = 1$ then the claim follows. Otherwise (i.e., $|\zeta| > 1$), there are two cases:

- There does not exist a chain ζ^* such that $|\zeta^*| < |\zeta|$ and $ZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta^*)$ exists. Hence, $ZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$ is an EZC by definition;

- There exists a chain ζ^* such that $|\zeta^*| < |\zeta|$ and $ZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta^*)$ exists. In this case, let consider $ZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta^*)$. If that Z-cycle is elementary then the claim follows. Otherwise we iterate previous reasoning on $ZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta^*)$. After a finite number of steps we get either an elementary Z-cycle or a Z-cycle whose size of ζ^* is equal to one (it is then elementary). Hence, the claim follows.

Q.E.D.

6.2.2 Prime Z-Cycles

This section introduces the notion of Prime Z-Cycle (PZC). It is interesting because of the result in Lemma 6.2.3 stating that if there is an elementary Z-cycle in a checkpoint and communication pattern of a distributed computation then there exists in that checkpoint and communication pattern a PZC whose size of its chain ζ is smaller than, or equal to, the one of the elementary Z-cycle.

Given a pair $(C_{i,x}, P_k)$, let us consider the set of causal chains μ starting after $C_{i,x}$ whose recipient of $\mu.last$ is P_k denoted $M(C_{i,x}, P_k)$. This set is partially ordered by the relation:

$$\mu \prec \mu' \Leftrightarrow receive(\mu.last) \prec_P receive(\mu'.last)$$

Let $min(M(C_{i,x}, P_k))$ denote the set of the *minimum* elements in $M(C_{i,x}, P_k)$ ([4]). This set contains causal chains starting after $C_{i,x}$ and sharing the last message. By using these notions the concept of Prime-Z-Cycle (PZC) is introduced as follows:

Definition 6.2.2
$EZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$ is a PZC, denoted $PZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$, iff $\mu \in min(M(C_{i,x}, P_k))$.

As an example $EZC(C_{i,x}, \mu' \overset{k,y}{\bullet} \zeta)$ shown in Figure 6.5 is not a PZC while $EZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$, shown in the same Figure, is a PZC. Let us introduce the following lemma:

Lemma 6.2.2
If *there exists $EZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$ such that $|\zeta| = 1$*
then *there exists $PZC(C_{i,x}, \mu' \overset{k,y}{\bullet} \zeta)$.*

[4]A chain $\mu \in M(C_{i,x}, P_k)$ is a minimum element if there does not exist any chain $\mu' \in M(C_{i,x}, P_k)$ such that $\mu' \prec \mu$.

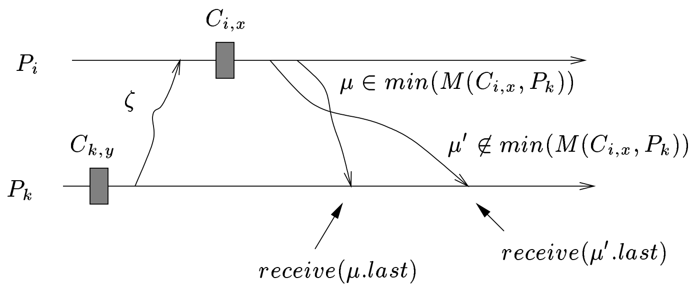


Figure 6.5: the Structure of an EZC and of a PZC.

Proof
Let us consider $EZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$ such that $\zeta = m$ (i.e., $|\zeta| = 1$). We have two alternatives:

1 **if** $\mu \in min(M(C_{i,x}, P_k))$ **then** let consider $\mu' = \mu$. By Definition 6.2.2 we get $PZC(C_{i,x}, \mu' \overset{k,y}{\bullet} \zeta)$ and the claim follows;

2 **if** $\mu \notin min(M(C_{i,x}, P_k))$ **then** let us consider $\mu' \in min(M(C_{i,x}, P_k))$ (note that μ' exists as $M(C_{i,x}, P_k)$ is not empty since it contains μ). There are two cases:

2.1 $receive(\mu'.last) \xrightarrow{e} send(m)$ (see Figure 6.6.a).
This is impossible as it would lead to a cycle in the *happened-before* relation (i.e., $send(m) \xrightarrow{e} receive(\mu'.last)$) which is acyclic [33];

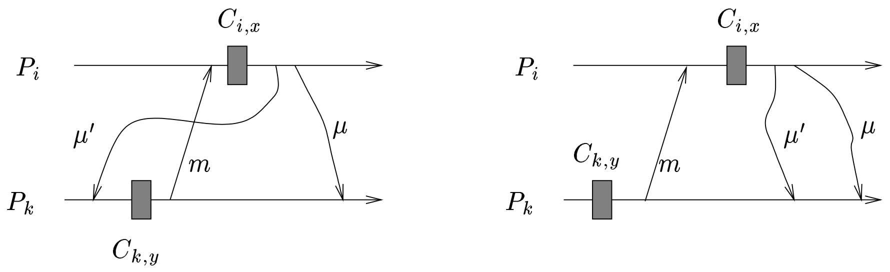
2.2 $send(m) \xrightarrow{e} receive(\mu'.last)$ (see Figure 6.6.b).
Thus, by Definition 6.2.2 we get $PZC(C_{i,x}, \mu' \overset{k,y}{\bullet} \zeta)$ and the claim follows.

Q.E.D.

Previous lemma says that if a checkpoint is involved in an Elementary Z-cycle whose chain ζ has size one, then there exists a PZC involving the same checkpoint. The following lemma extends the previous result to a chain ζ of any size:

Lemma 6.2.3
If *there exists* $EZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$
then *there exists* $PZC(C_{i,x}, \mu' \overset{l,z}{\bullet} \zeta')$ *with* $|\zeta'| \leq |\zeta|$.

Proof
Let us consider $EZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$. We have two alternatives:



(a) (b)

Figure 6.6: Proof of Lemma 6.2.2.

1 **if** $|\zeta| = 1$ **then** the claim follows from Lemma 6.2.2;

2 **if** $|\zeta| > 1$ **then** if $\mu' = \mu \in min(M(C_{i,x}, P_k))$ then the claim trivially follows. Otherwise let us consider $\mu' \in min(M(C_{i,x}, P_k))$ (note that μ' exists as $M(C_{i,x}, P_k)$ is not empty since it contains μ). There are two cases:

2.1 $send(\zeta.first) \xrightarrow{e} receive(\mu'.last)$ (see Figure 6.7.a).
In this case we get $PZC(C_{i,x}, \mu' \overset{I_{k,y}}{\bullet} \zeta)$ and the claim follows;

2.2 $receive(\mu'.last) \xrightarrow{e} send(\zeta.first)$ (see Figure 6.7.b).
In this case, by construction, we get $ZC(C_{i,x}, [\mu' \circ \mu''] \overset{h,w}{\bullet} \zeta')$ where $\zeta = \mu'' \overset{h,w}{\bullet} \zeta'$ (note that $|\mu''| \geq 1$) and $|\zeta'| < |\zeta|$ (see Figure 6.7.c). From Lemma 6.2.1, there exists an elementary Z-cycle $EZC(C_{i,x}, [\mu' \circ \mu''] \overset{h,w}{\bullet} \zeta^*)$ with $|\zeta^*| \leq |\zeta'| < |\zeta|$.

If we fall in case 2.2, the previous construction can be repeated on the elementary Z-cycle $EZC(C_{i,x}, [\mu' \circ \mu''] \overset{h,w}{\bullet} \zeta^*)$ and after a finite number of steps either we fall in case 2.1 or we get $EZC(C_{i,x}, \widehat{\mu} \overset{l,z}{\bullet} \widehat{\zeta})$ with $|\widehat{\zeta}| = 1$ thus the claim follows from Lemma 6.2.2.

Q.E.D.

6.2.3 Core Z-Cycles

This section introduces the notion of Core Z-Cycle (CZC). It is interesting because of the result in Lemma 6.2.5 stating that if there is a PZC involving a checkpoint then there exists a CZC that involves a checkpoint (not necessarily the same checkpoint involved in the PZC). Before introducing the notion of CZC, let us introduce a precedence relation on checkpoint intervals:

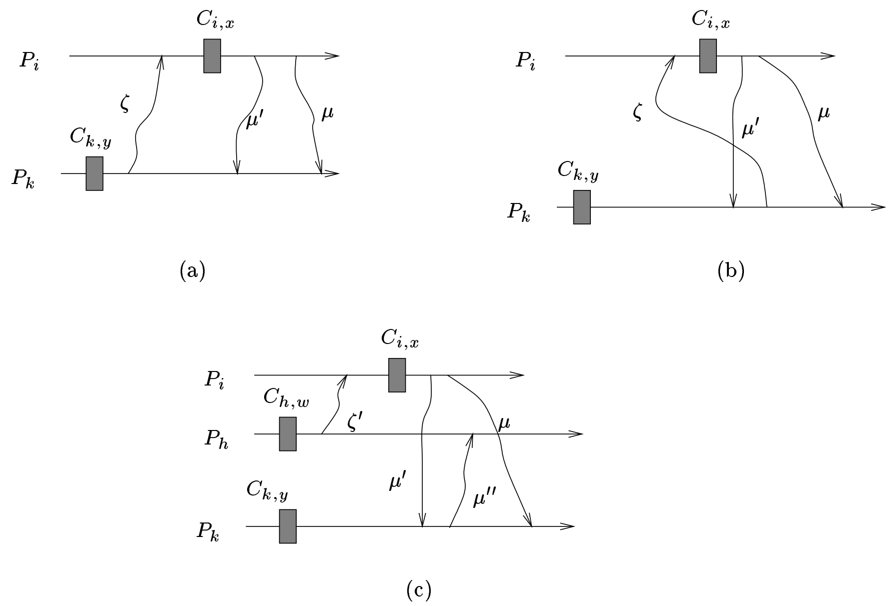


Figure 6.7: Proof of Lemma 6.2.3.

Definition 6.2.3
A checkpoint interval $I_{i,x}$ precedes a checkpoint interval $I_{j,y}$, denoted $I_{i,x} \xrightarrow{I} I_{j,y}$, iff:

$$\exists e_{i,x'} \in I_{i,x}, \exists e_{j,y'} \in I_{j,y} \quad : \quad e_{i,x'} \xrightarrow{e} e_{j,y'}$$

A CZC is actually a PZC with a restriction on its structure. This restriction derives from the sequence of checkpoint intervals related to its message chain ζ as it can be seen from the following definition:

Definition 6.2.4
Let consider $PZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$ and let $S(\zeta)$ be the sequence of checkpoint intervals associated to ζ. That PZC is a Core Z-Cycle, denoted $CZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$ iff:

$$\forall I_{j_i,z_i} \in S(\zeta) \Rightarrow \neg(I_{j_i,z_i+1} \xrightarrow{I} I_{k,y})$$

Figure 6.8 shows an example of a CZC involving $C_{i,x}$ and an example of a PZC which is not a CZC as it contradicts the restriction in Definition 6.2.4 (i.e., $I_{j,z+1} \xrightarrow{I} I_{k,y}$ due to the presence of the causal message chain μ'). Note that, in the latter case, $PZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$ embeds a Z-cycle $ZC(C_{j,z+1}, \mu' \overset{k,y}{\bullet} (\zeta - \zeta.last))$

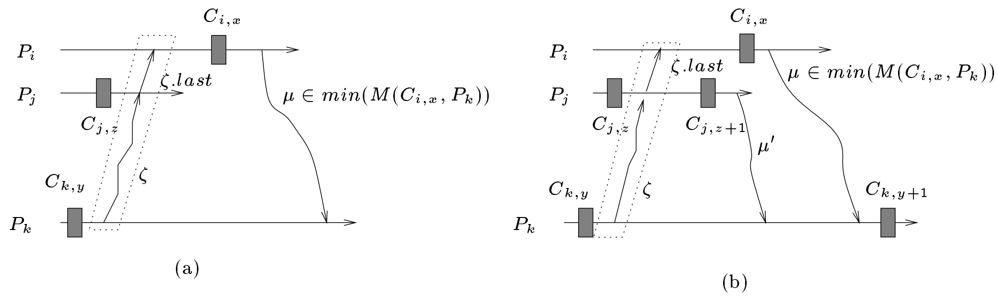


Figure 6.8: a Core Z-Cycle Involving $C_{i,x}$ (a); an Example of non-Core Z-cycle (b).

as shown in Figure 6.8.b. This recursive behavior will be exploited in the proof of Lemma 6.2.5.

Let us now prove that if there exists a PZC in a distributed computation, then there exists a CZC in that computation, assuming the size of the non-causal message chain of the PZC equal to one and then we generalize the result to a chain of any size:

Lemma 6.2.4

If *there exists* $PZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$ *such that* $|\zeta| = 1$

then *there exists* $CZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$.

Proof (By Contradiction)

Let us consider $PZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$ with $\zeta = m$ and suppose that $CZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$ does not exists. As $m \circ C_{i,x} \circ \mu \overset{k,y}{\bullet} m$, $send(m) \in I_{k,y}$ and $\mu \in min(M(C_{i,x}, P_k))$, there must exist $C_{k,y+1}$ such that:

$$I_{k,y+1} \xrightarrow{I} I_{k,y}$$

In this case, by Definition 6.2.3, there exist an event $e' \in I_{k,y+1}$ and an event $e'' \in I_{k,y}$ such that $e' \xrightarrow{e} e''$ which is not possible due to the fact that the $\xrightarrow{e}$ relation is acyclic.

Q.E.D.

Lemma 6.2.5

If *there exists* $PZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$

then *there exists a CZC.*

Proof

Let us consider $PZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$. We have two alternatives:

1 **if** $|\zeta| = 1$ **then** the claim follows from Lemma 6.2.4

2 **if** $|\zeta| > 1$ **then** let consider the sequence of checkpoint intervals $S(\zeta)$. There are two cases:

2.A $\forall I_{j_i,z_i} \in S(\zeta) \Rightarrow \neg(I_{j_i,z_i+1} \xrightarrow{I} I_{k,y})$.
By definition 6.2.4, we get $CZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$ and the claim follows;

2.B $\exists I_{j_i,z_i} \in S(\zeta) \ : \ I_{j_i,z_i+1} \xrightarrow{I} I_{k,y}$.
Let $I_{j,z+1}$ be the *first* checkpoint interval in $S(\zeta)$ satisfying the condition of Case 2.B. There exists at least one causal message chain starting after $C_{j,z+1}$ and ending in $I_{k,y}$ or in a previous checkpoint interval of P_k. Therefore, the set $M(C_{j,z+1}, P_k)$ is not empty. Let us consider $\mu' \in min(M(C_{j,z+1}, P_k))$; we have two cases:

2.B.1 $send(\zeta.first) \xrightarrow{e} receive(\mu'.last)$ (see Figure 6.9.a).
We get $ZC(C_{j,z+1}, \mu' \overset{k,y}{\bullet} \zeta^*)$ where $\zeta^* = \zeta - \widehat{\zeta}$ and $send(\widehat{\zeta}.first) \in I_{j,z}$. From the successive application of Lemma 6.2.1 and Lemma 6.2.3, there exists $PZC(C_{j,z+1}, \bar{\mu} \overset{l,t}{\bullet} \bar{\zeta})$ with $|\bar{\zeta}| \leq |\zeta^*| < |\zeta|$;

2.B.2 $receive(\mu'.last) \xrightarrow{e} send(\zeta.first)$ (see Figure 6.9.b).
We get $ZC(C_{j,z+1}, [\mu' \circ \mu''] \overset{b,s}{\bullet} \zeta')$ where $\mu'' \overset{I_{b,s}}{\bullet} \zeta' = \zeta - \widehat{\zeta}$ and $send(\widehat{\zeta}.first) \in I_{j,z}$, hence $|\zeta'| < |\zeta|$ (see Figure 6.9.c). By Lemma 6.2.1 and Lemma 6.2.3 there exists $PZC(C_{j,z+1}, \bar{\mu} \overset{l,t}{\bullet} \bar{\zeta})$ with $|\bar{\zeta}| \leq |\zeta'|$. So we have $|\bar{\zeta}| < |\zeta|$;

In both cases we obtain a PZC with $|\bar{\zeta}| < |\zeta|$.

If we fall in case 2.B, the previous construction can be applied on the obtained PZC. After a finite number of steps, either we fall in case 2.A or $|\bar{\zeta}| = 1$ thus, by Lemma 6.2.4, we get a CZC.

Q.E.D.

6.2.4 A Characterization Theorem

Let us formally introduce the No-Core-Z-Cycle property $\mathcal{NCZC}$:

Definition 6.2.5 *A checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ of a distributed computation satisfies the No-Core-Z-Cycle ($\mathcal{NCZC}$) property iff no CZC exists in $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$*

The following characterization theorem is straightforwardly derived from lemmas introduced in previous section:

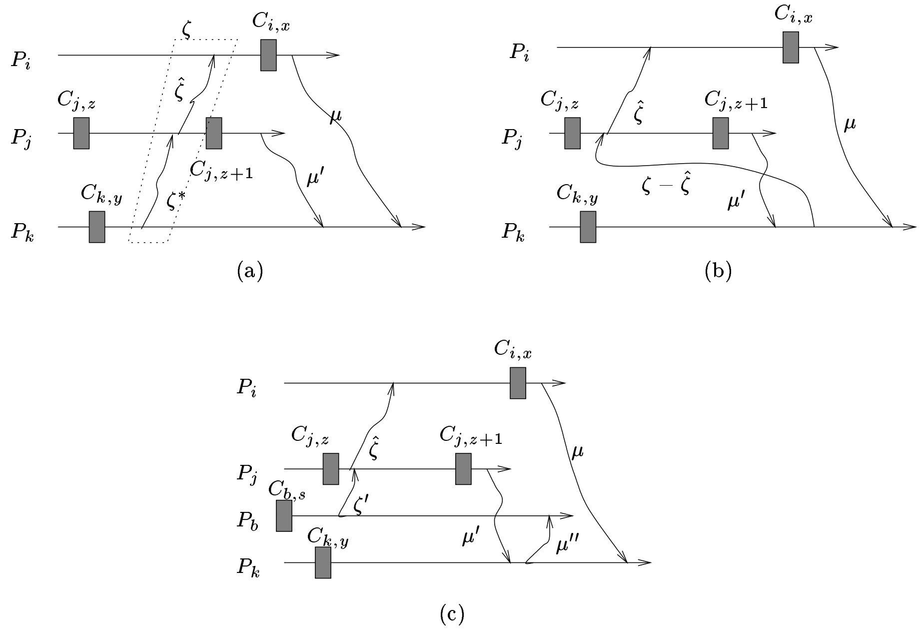


Figure 6.9: Proof of Lemma 6.2.5.

Theorem 6.2.6
A checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ of a distributed computation satisfies the $\mathcal{NZC}$ property iff $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ satisfies the $\mathcal{NCZC}$ property.

Proof

If part. By Lemma 6.2.1 if a ZC exists then an EZC exists in $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$. By Lemma 6.2.3 if an EZC exists then a PZC exists in $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$. By lemma 6.2.5 if a PZC exists then a CZC exists in $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$. Thus, in terms of properties, $\neg(\mathcal{NZC}) \Rightarrow \neg(\mathcal{NCZC})$. Hence $\mathcal{NCZC} \Rightarrow \mathcal{NZC}$.

Only if part. If the computation satisfies $\mathcal{NZC}$ then no CZC exists as CZCs are Z-cycles. So the computation satisfies $\mathcal{NCZC}$.

Q.E.D.

6.3 Deriving $\mathcal{VP}$-Accordant Protocols

6.3.1 Suspect Core Z-Cycles

From Theorem 6.2.6, a checkpoint and communication pattern $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ of a distributed computation satisfies the $\mathcal{NZC}$ property if, and only if, no CZC exists in $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$. Given $CZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$, it can be broken by placing an additional local checkpoint taken between the send of $\zeta.first$ and the receive of $\mu.last$ at process P_k as shown in Figure 6.10.a. So for any communication-induced checkpointing protocol, the instant of time before the event $receive(\mu.last)$ represents "the last opportunity" for taking an additional (forced) checkpoint in order to remove that CZC from the checkpoint and communication pattern of the computation.

Like a Z-cycle, a core Z-cycle is generally non-trackable on-the-fly at the last opportunity time by a communication-induced checkpointing protocol. This is due to a key factor: the message chain ζ could contain at least one non-causal concatenation (for example the message chain ζ shown in Figure 6.10.a contains two non-causal concatenations). In other words a CZC is trackable on-the-fly at the last opportunity time only if its chain ζ is causal.

The previous argument shows that the best a communication-induced protocol can do to *prevent* the formation of core Z-cycles is to remove from $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ those checkpoint and communication patterns whose structure represents the *common causal part of any core Z-cycle* which is detectable by a process at the last opportunity time. Those considerations lead to the introduction of a checkpoint and communication pattern, namely Suspect Core Z-Cycle (SCZC), which is trackable by a communication-induced checkpointing protocol. Such pattern is structured as follows:

Definition 6.3.1
A Suspect Core Z-Cycle (SCZC) *is a checkpoint and communication pattern* $SCZC(I_{j,z}, C_{i,x}, \mu, I_{k,y})$ *such that:*

$$\exists m, m' \ : \ C_{j,z} \circ m \circ C_{i,x} \circ \mu \overset{k,y}{\bullet} m'$$

$$with \ \begin{cases} (i) & send(m) \in I_{j,z} \\ (ii) & \mu \in min(M(C_{i,x}, P_k)) \\ (iii) & \nexists e \in I_{j,z+1} \ \ : \ \ e \overset{e}{\rightarrow} receive(\mu.last) \end{cases}$$

As an example $SCZC(I_{j,z}, C_{i,x}, \mu, I_{k,y})$ is shown in Figure 6.10.b while Figure 6.10.c shows a checkpoint and communication pattern which is not an SCZC as it violates the constraint (iii) of previous definition (due to the causal message chain μ'). Trivially, the presence of a CZC implies the existence of an SCZC (the converse being not true) so, if no SCZC exists in a checkpoint and communication pattern of a distributed computation then no CZC exists and, then, according to Theorem 6.2.6, the execution satisfies the $\mathcal{NZC}$ property. Let us now state a theorem, actually a sufficient condition for the $\mathcal{NZC}$ property, that will be used to design communication-induced protocols shown in the following sections:

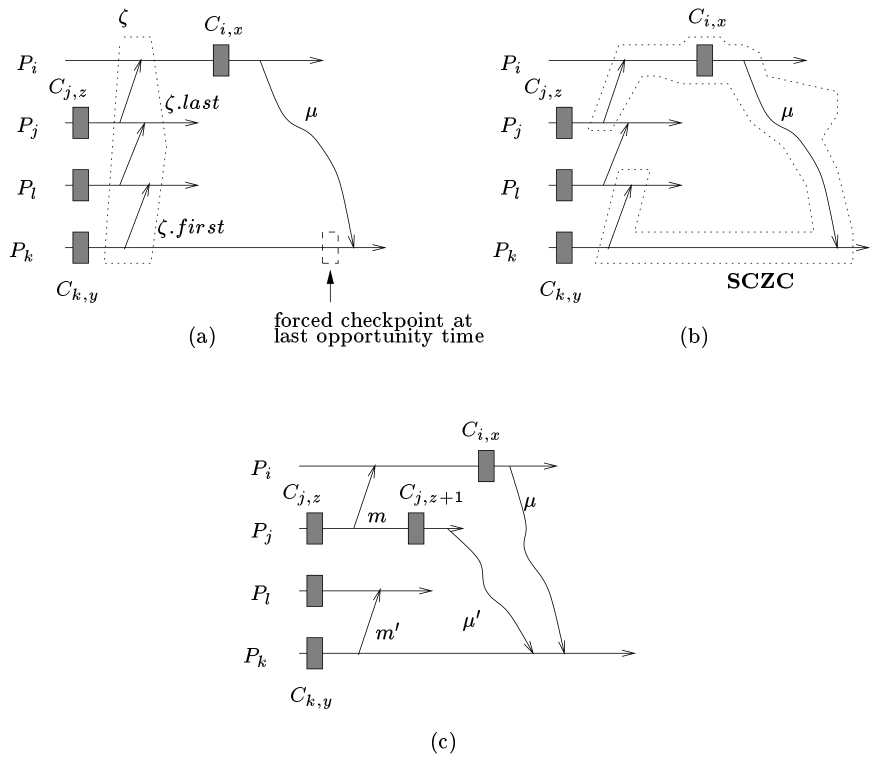


Figure 6.10: an Example of CZC non-Trackable on-the-Fly by a Communication-Induced Checkpointing Protocol (a); an Example of SCZC Pattern (b); an Example of non SCZC Pattern (c).

Theorem 6.3.1
If a checkpoint and communication pattern of a distributed computation $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ *does not include any SCZC (i.e., it satisfies the* No-Suspect-Core-Z-Cycle *property* $\mathcal{NSCZC}$*) then* $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ *satisfies the* $\mathcal{NZC}$ *property.*

Proof
From the structure of the CZC and of the SCZC, it trivially follows, in terms of properties, $\mathcal{NSCZC} \Rightarrow \mathcal{NCZC}$. From Theorem 6.2.6 we have $\mathcal{NCZC} \Rightarrow \mathcal{NZC}$. Hence we get $\mathcal{NSCZC} \Rightarrow \mathcal{NZC}$. *Q.E.D.*

The reader could now wonder if the SCZC is the right pattern to prevent in order to remove CZCs. In particular, why the SCZC structure includes only the last checkpoint interval passed through by ζ (i.e., $I_{j,z}$) and not all the checkpoint intervals associated to the *final causal part* of the non-causal message chain ζ associated to a CZC. This causal part would represent the larger part of ζ visible by P_k at the last opportunity time.

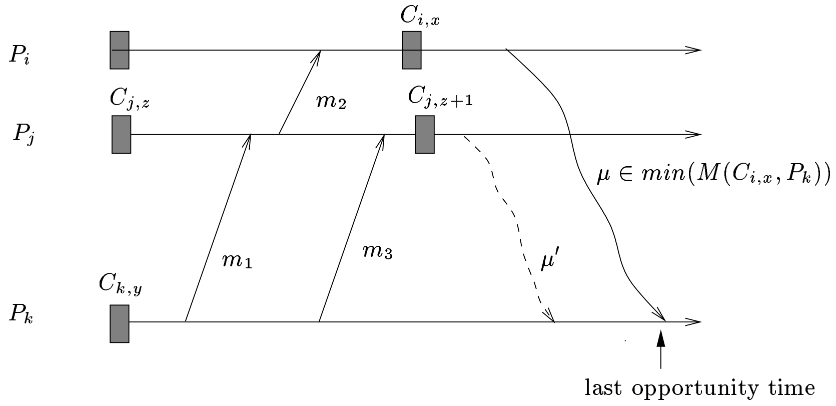


Figure 6.11: A set of PZCs Involving $C_{i,x}$.

Let **MC** be the set of message chains ζ of minimal length starting after $C_{k,y}$, terminating before $C_{i,x}$ and sharing the last message $\zeta.last$. This defines a set of PZCs **X** one for each distinct ζ in **MC**. If we consider Z-cycles involving $C_{i,x}$ in Figure 6.11 we have **MC**$\equiv$ $\{[m_1, m_2], [m_3, m_2]\}$ and **X**$\equiv$ $\{PZC(C_{i,x}, \mu \overset{k,y}{\bullet} [m_1, m_2]), PZC(C_{i,x}, \overset{k,y}{\bullet} [m_3, m_2])\}$.

Let us assume the causal message chain μ', depicted by a dotted line in Figure 6.11, does exist. As a consequence we have $\exists e \in I_{j,z+1} : e \rightarrow deliver(\mu.last)$, which implies $I_{j,z+1} \overset{I}{\rightarrow} I_{k,y}$. Hence, each PZC in **X** is not a CZC (see Definition 6.2.4).

Let us assume the causal message chain μ' in Figure 6.11 does not exist. In such a case, at the last opportunity time P_k cannot safely conclude that no CZC can be formed due to a message chain $\zeta \in$ **MC** which relies on a non-causal concatenation in $I_{j,z}$. For example, the non-causal concatenation forming the message chain $[m_3, m_2]$ is out of the usable knowledge of P_k. This chain gives rise to $CZC(C_{i,x}, \mu \overset{k,y}{\bullet} [m_3, m_2])$. Hence, a communication-induced protocol is obliged to direct a forced checkpoint before executing $receive(\mu.last)$ if no information concerning the definite delimitation of the checkpoint interval $I_{j,z}$ has been notified to P_k by means of a causal message chain.

In conclusion, it is not possible for a communication-induced protocol to prevent checkpoint and communication patterns less constrained than the SCZC pattern in order to do a safe removal of CZCs.

6.3.2 A Remark on Characterizations Stronger than $\mathcal{NCZC}$

Imposing additional constraints on the structure of a CZC can lead to characterizations stronger than $\mathcal{NCZC}$. As an example, let consider the subset **X** of

CZC such that (i) the length of μ is minimal, and (ii) ζ is a member of a set of message chains that establish the first Z-path between $C_{k,y}$ and $C_{i,x}$ (this set contains message chains sharing the last message) [5]. The existence of any CZCs in the execution implies the existence of a Z-cycle in **X**, thus, if **X** is empty, then **CZC** is empty.

Although the latter characterization could be interesting from a theoretical point of view, from a practical one, it does not add information, suitable for communication-induced protocols, in order to reduce the number of forced checkpoints compared to the one provided by CZC. In other words, this characterization does not help to find checkpoint and communication patterns more refined than SCZC and detectable on-the-fly. More specifically, the information concerning the "time" in which the chain ζ is established does not help as ζ is, generally, non-causal and, thus, it cannot be tracked at the last opportunity time by a protocol as shown in the previous section. The information on the length of μ does not help to save forced checkpoints as the concept of *min* is related to a set of causal message chains which includes the one of minimal length, thus, preventing a non-causal concatenation (e.g. $\mu \bullet m$) due to either any chain of the set $min(M(C_{i,x}, P_k))$ or the one with minimal length has the same effect in terms of forced checkpoints.

6.3.3 A Checkpointing Protocol (P1) Preventing SCZCs

The protocol presented in this section, namely P1, tracks on-the-fly all the SCZC patterns, and breaks them by introducing a forced checkpoint before the receipt of message $\mu.last$ (i.e., it breaks them at last opportunity time). This is done by exploiting the control information piggybacked on application messages, that encodes the causal past of the execution with respect to the event of the receive of a message, and the local history of a process (i.e., it fully exploits the usable knowledge at that event). The protocol uses a vector clock and a matrix of integers as control information.

Tracking SCZC Patterns

In order to track the formation of $SCZC(I_{j,z}, C_{i,x}, \mu, I_{k,y})$, upon the arrival of a message $\mu.last$, process P_k has to verify whether conditions for the existence of that checkpoint and communication pattern are satisfied. In the following paragraphs the data structures to accomplish this task are introduced.

Tracking $\mu \in min(M(C_{i,x}, P_k))$.
To detect if $\mu \in min(M(C_{i,x}, P_k))$, a vector clock mechanism is used considering checkpoints of processes as relevant events of the computation [38]. Each

[5] In such a case we have a "temporal" and a "spatial" constraint both on ζ and on μ.

process P_k maintains a vector clock VC_k whose size corresponds to the number of processes n. $VC_k[i]$ stores the maximum checkpoint rank of P_i seen by P_k and $VC_k[k]$ stores the rank of the last checkpoint taken by P_k. VC_k is initialized to zero except the k-th entry which is initialized to one. Each application message `m` sent by P_k piggybacks the current value of VC_k (denoted $\mathtt{m}.VC$). Following the classical updating rule of a vector clock, upon the receipt of a message `m`, VC_k is updated from $\mathtt{m}.VC$ by taking a component-wise maximum.

A causal message chain μ including message `m` as $\mu.last$ is *prime* (i.e., μ belongs to some $min(M(C_{i,*}, P_k))$, if, upon the receipt of `m` at process P_k, the following predicate holds:

$$\exists i \ : \ (\mathtt{m}.VC[i] > VC_k[i])$$

Tracking $\mu \overset{k,y}{\bullet} m'$.
To detect if there exists a non-causal concatenation between a prime causal message chain μ and a message m' in the interval $I_{k,y}$, process P_k maintains a boolean variable $after\_first\_send_k$. This variable is set to $TRUE$ when a send event occurs. It is set to $FALSE$ each time a local checkpoint is taken. Hence, upon the receipt of a message `m` (with $\mathtt{m} = \mu.last$), P_k detects that $\mu \overset{k,y}{\bullet} m'$ if the following predicate hold:

$$after\_first\_send_k \wedge (\exists i \ : \ (\mathtt{m}.VC[i] > VC_k[i]))$$

Tracking $C_{j,z} \circ m \circ C_{i,x}$.
Each process P_k maintains a vector of integers Imm_Pred_k of size n and a matrix of integers $Pred_k$, of size $n \times n$. $Imm\_Pred_k[\ell]$ represents the maximum rank of the checkpoint interval from which process P_ℓ sent a message m which has been received by P_k in its current checkpoint interval $I_{k,y-1}$ (in other words $C_{\ell,Imm\_Pred[\ell]}$ precedes checkpoint $C_{k,y}$ due to the $\prec_{ckpt}$ relation). Each entry of this vector is set to -1 every time a checkpoint is taken by P_k.

$Pred_k[i,j]$ represents, to the knowledge of P_k, the maximum rank of the checkpoint interval from which process P_j sent a message m which has been received by P_i in a checkpoint interval $I_{i,x-1}$ with $x \leq VC_k[i]$. Each entry of the matrix $Pred_k$ is initialized to -1. Its content is piggybacked on each message `m` sent by P_k ($\mathtt{m}.Pred$) and the rules to update its entries are the following:

1. Whenever a checkpoint is taken by P_k, $Pred_k[k,-]$ is updated according to the following rule:

$$\forall j \ \ Pred_k[k,j] = max(Pred_k[k,j], Imm\_Pred_k[j])$$

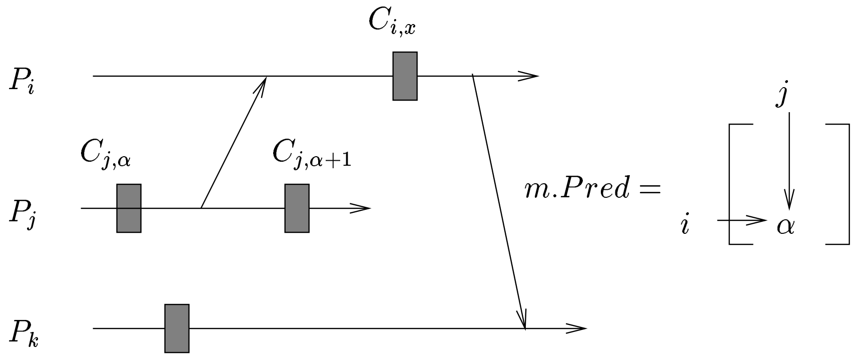


Figure 6.12: Example of Values Stored in $m.Pred$.

2. Upon the arrival of a message m at P_k:

$$\forall \ell, t \ \ Pred_k[\ell, t] = max(Pred_k[\ell, t], \mathtt{m}.Pred[\ell, t])$$

Figure 6.12 shows an example of a checkpoint and communication pattern and the content of $m.Pred[i, j]$ associated to that pattern.

Tracking $\nexists e \in I_{j,z+1} \ : \ e \xrightarrow{e} receive(\mu.last)$**.**
Upon the arrival of a message m included in a prime causal chain (recall that m ends a prime causal chain if $\exists i \ : \ (\mathtt{m}.VC[i] > VC_k[i])$), in order to track the above condition, we need to know if there exists a j such that $\mathtt{m}.Pred[i, j] + 1$ does not belong to the causal past of the receipt of m. This knowledge is encoded in $\mathtt{m}.VC[j]$ and $VC_k[j]$. Hence, the predicate becomes:

$$(\exists j \ : \ \mathtt{m}.Pred[i, j] + 1 > max(\mathtt{m}.VC[j], VC_k[j]))$$

Preventing SCZC Patterns

Upon the arrival of a message m at process P_k in $I_{k,y}$, if the following predicate holds:

$$\begin{aligned} \mathcal{P}_1 \ \equiv \ & after\_first\_send_k \wedge \\ & (\exists i \ : \ (\mathtt{m}.VC[i] > VC_k[i]) \wedge \\ & \ \ (\exists j \ : \ \mathtt{m}.Pred[i, j] + 1 > max(\mathtt{m}.VC[j], VC_k[j]))) \end{aligned}$$

then, process P_k detects that at least one $SCZC(I_{j,Pred_k[i,j]}, C_{i,x}, \mu, I_{k,y})$ is going to be formed with $\mathtt{m} = \mu.last$ and $VC_k[i] < x \leq \mathtt{m}.VC[i]$. In this case P_k directs a forced checkpoint $C_{k,y+1}$ before the receipt of m.

```
init P_k:
take a checkpoint;
after_first_send_k := FALSE;
∀i : i ≠ k VC_k[i] := 0; VC_k[k] := 1;
∀i,∀j Pred_k[i,j] := −1; ∀h Imm_Pred_k[h] := −1;

when m arrives at P_k from P_l:
if  after_first_send_k ∧ (∃i : (m.VC[i] > VC_k[i])∧
    (∃j : m.Pred[i,j] + 1 > max(m.VC[j], VC_k[j])))
then  take_ckpt();                                    % forced checkpoint %
∀i VC_k[i] := max(VC_k[i], m.VC[i]);                  % component-wise maximum %
∀i,∀j Pred_k[i,j] := max(m.Pred[i,j], Pred_k[i,j]);
Imm_Pred_k[l] := max(Imm_Pred_k[l], m.VC[l]);

procedure send(m, P_j):
m.content = data; m.VC := VC_k; m.Pred := Pred_k;     % packet the message %
send m to P_j;
after_first_send_k := TRUE;

when a basic checkpoint is scheduled from P_k:
take_ckpt();

procedure take_ckpt():
take a checkpoint;
∀h Pred_k[k,h] := max(Pred_k[k,h], Imm_Pred_k[h]);    % component-wise maximum %
∀h Imm_Pred_k[h] := −1;
VC_k[k] := VC_k[k] + 1;
after_first_send_k := FALSE;
```

Figure 6.13: Protocol P1

The behavior of process P_k is shown in Figure 6.13 (all the procedures and the message handler are executed in atomic fashion).

From an operational point of view, the elements of the diagonal of the matrix $Pred$ are never used by the protocol. Hence, when implementing the protocol, the vector clock VC can be embedded in that diagonal. Thus, the resulting control information piggybacked on application messages boils down to a matrix of $n \times n$ integers.

6.3.4 A Comparison with Previous $\mathcal{VP}$-Accordant Protocols

As protocol P1 is, to our knowledge, the first $\mathcal{VP}$-accordant protocol that ensures $\mathcal{NZC}$ but not $\mathcal{RDT}$, it is expected that it generates less overhead, in terms of forced checkpoints, compared to other $\mathcal{VP}$-accordant protocols since they ensure a stronger property.

Before comparing P1 to previous $\mathcal{VP}$-accordant protocols, a technical description of such protocols is sketched by using the introduced concatenation relations. The $\mathcal{VP}$-accordant protocols selected for the comparison are the Russell's protocol [50], the FDAS protocol [64] and the protocol by Baldoni et al. [4]. The other $\mathcal{VP}$-accordant protocols (i.e., CAS, CBR, CASBR, FDI), being derivations of the FDAS protocol, are not considered in the comparison.

Russell's Protocol [50].
This protocol accepts only causal message chains in a computation. It actually prevents the formation of $\langle send \cdot receive \rangle$ (i.e., $m \bullet m'$) patterns in any checkpoint interval by means of forced checkpoints, so no non-causal concatenation of messages can ever occur, preventing the formation of Z-cycles.

FDAS Protocol [64].
FDAS avoids the formation of checkpoint and communication patterns with the following structure:

$$C_{i,x} \circ \mu \overset{k,y}{\bullet} m'$$

with $\mu \in min(M(C_{i,x}, P_k))$. As the previous pattern is a part of the structure of a PZC, the prevention of all those patterns guarantees no prime Z-cycle in the checkpoint and communication pattern of the distributed computation and thus the $\mathcal{NZC}$ property.

Baldoni et al. Protocol [4] (BHMR).
This protocol prevents the formation of dependences between two checkpoints due to non-causal message chains composed by two causal message chains (i.e., $\zeta = \mu \overset{k,y}{\bullet} \mu'$) if they are not doubled, in a visible way, by a causal message chain. In terms of concatenation relations, we get that a non-causal message chain $\zeta = \mu \overset{k,y}{\bullet} \mu'$ is doubled by a causal one μ'' if the pair of checkpoints related by ζ is also related by μ'' (i.e., if $C_{i,x} \circ \zeta \circ C_{j,y}$ then $C_{i,x} \circ \mu'' \circ C_{j,y}$). The doubling is *visible* by P_k (the only process able to break ζ) if there exists a causal message chain μ''' such that $\mu'' \circ \mu'''$ belongs to $min(M(C_{i,x}, P_k))$.

This protocol prevents the formation of any $CZC(C_{i,x}, \mu \overset{k,y}{\bullet} \zeta)$. In particular there are two cases:

- $\zeta = \mu'$ i.e., ζ is a causal message chain. $CZC(C_{i,x}, \mu \overset{I_{k,y}}{\bullet} \mu')$ is a particular dependence between $C_{i,x}$ and itself that cannot be doubled, so the BHMR protocol prevents it by taking a forced checkpoint before the receipt of $\mu.last$;

- $\zeta = \mu_1 \bullet \mu_2 \bullet \ldots \bullet \mu_\ell$ with $\ell > 1$ where each pair of successive causal message chains establishes a dependence between two distinct checkpoints that it is not doubled. Note that, that composition of ζ must exist, otherwise we fall in the previous case. Then the protocol prevents this pattern by taking ℓ forced checkpoints. $\ell - 1$ forced checkpoints are taken to prevent each non-causal concatenation of two successive causal message chains composing ζ. The last forced checkpoint is taken by P_k to prevent the pattern $\mu \overset{k,y}{\bullet} \zeta.first$.

The Comparison

It follows trivially that the Russell's pattern, $m \bullet m'$, and the FDAS's pattern, $C_{i,x} \circ \mu \overset{k,y}{\bullet} m'$ are a part of an SCZC. When considering the same usable knowledge (i.e., the protocol decides to take a forced checkpoint based on the same past checkpoint and communication pattern), each time the proposed protocol P1 takes a forced checkpoint, Russell's protocol takes a forced checkpoint and each time P1 takes a forced checkpoint, FDAS protocol takes a forced checkpoint.

As far as BHMR is concerned, only a qualitative comparison can be done between patterns prevented by the protocols. Figure 6.14.a shows a checkpoint and communication pattern in which BHMR protocol takes a forced checkpoint while the proposed protocol P1 does not take it. Whereas Figure 6.14.b shows a scenario in which the proposed protocol takes a forced checkpoint while BHMR protocol does not take it. Note that the probability that checkpoint and communication patterns, like the one proposed in Figure 6.14.a, occur in a computation is extremely higher than that of the pattern depicted in Figure 6.14.b. For a quantitative comparison between the two protocols realized through a simulation study the reader can refer to Section 6.5 of this chapter.

6.3.5 Reducing the Size of the Control Information of P1: Protocol P2

In this section a communication-induced checkpointing protocol, namely P2, is presented. The protocol, compared to P1, has control information with reduced size (the space-complexity decreases form $O(n^2)$ to $O(n)$). In P2 a forced checkpoint is taken upon the receipt of a message m whenever a predicate $\mathcal{P}_2$ is evaluated to true. The following relation holds between the predicate $\mathcal{P}_1$ proper of protocol P1 and the predicate $\mathcal{P}_2$:

$$\mathcal{P}_1 \Rightarrow \mathcal{P}_2$$

Such an inclusion between predicates guarantees that also under P2 no

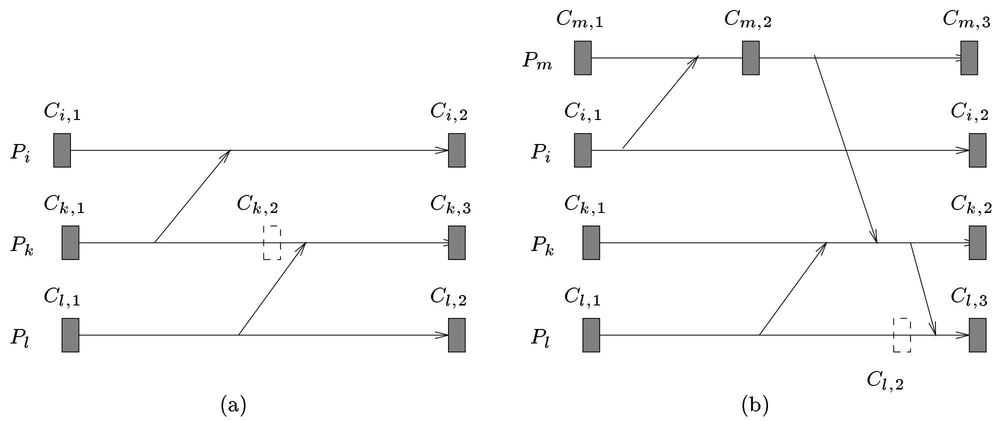


Figure 6.14: Two Checkpoint and Communication Patterns for a Comparison between BHMR and the Proposed Protocol P1.

SCZC is ever formed. Thus also P2 guarantees that the resulting checkpoint and communication pattern of the distributed computation satisfies $\mathcal{NZC}$

The advantage of $\mathcal{P}_2$ is that it can be tracked on-the-fly by using two vectors of n integers. The disadvantage is that, due to the inclusion between predicates, P2 potentially[6] induces processes to take more forced checkpoints compared to P1.

The predicate $\mathcal{P}_2$ is structured as follows:

$$\begin{aligned} \mathcal{P}_2 \equiv\ & after\_first\_send_k \wedge \\ & (\exists i\ :\ (\mathtt{m}.VC[i] > VC_k[i]) \wedge \\ & \quad (\exists j\ :\ max_{1 \leq h \leq n}\mathtt{m}.Pred[h,j] + 1 > max(\mathtt{m}.VC[j], VC_k[j]))) \end{aligned}$$

While $\mathcal{P}_1$ considers only the entry with index (i,j) of the matrix $\mathtt{m}.Pred$, $\mathcal{P}_2$ takes into account the maximum over all the rows of the matrix. This difference allows $\mathcal{P}_2$ to be tracked by using a vector of n entries instead of a matrix. More technically, process P_k is endowed with all the data structures used in protocol P1 except the matrix $Pred_k$. Instead, P_k owns a vector Max_Pred_k of n integers. $Max\_Pred_k[j]$ represents, to the knowledge of P_k, the maximum rank of the checkpoint interval from which process P_j sent a message m which has been received by whichever process P_i in a checkpoint

[6] As already discussed in Section 3.1 of Chapter 3, the inclusion between predicates means that P1 takes a forced checkpoint whenever P2 does it only under the same causal past. As there is no guaranty that the computation evolves at the same way under different checkpointing protocols, performance of P1, in terms of forced checkpoints, is not guaranteed to be better than that of P2. This is why the term "potentially" is used.

interval $I_{i,x-1}$ with $x \leq VC_k[i]$. All the entries of Max_Pred_k are initialized to -1, and its content is piggybacked on each message m sent by P_k (m.Max_Pred). The rules to update its entries are the following:

1. Whenever a checkpoint is taken by P_k:

$$\forall j \ Max\_Pred_k[j] = max(Max\_Pred_k[j], Imm\_Pred_k[j])$$

2. Upon the arrival of a message m at P_k:

$$\forall j \ Max\_Pred_k[j] = max(Max\_Pred_k[j], \ \texttt{m}.Max\_Pred[j])$$

By using Max_Pred_k, $\mathcal{P}_2$ can be expressed as:

$$\begin{aligned}\mathcal{P}_2 \ \equiv \ & after\_first\_send_k \wedge \\ & (\exists i \ : \ (\texttt{m}.VC[i] > VC_k[i]) \wedge \\ & \ \ (\exists j \ : \ \texttt{m}.Max\_Pred[j] + 1 > max(\texttt{m}.VC[j], VC_k[j])))\end{aligned}$$

The resulting checkpointing protocol P2 is shown in Figure 6.3.5.

6.3.6 A Comparison with $\mathcal{VP}$-Enforced Protocols

This section presents a performance comparison between the proposed checkpointing protocols (P1 and P2) and $\mathcal{VP}$-enforced ones. As $\mathcal{VP}$-enforced protocols are not based on the prevention of a particular type of sub-patterns, the comparison is not realized at a theoretical level (i.e., by finding inclusions between predicates triggering forced checkpoints at the receipt of a message), but through simulation results.

The $\mathcal{VP}$-enforced protocol considered here is the one by Briatico et al. [12], hereafter BCS (note that all the protocols BCS, P1 and P2 ensure the same property, i.e., $\mathcal{NZC}$). Among the set of checkpointing protocols, we chose BCS, first, for its simplicity of implementation, and, second, because simulation studies ([6]) have shown that, in the class of $\mathcal{VP}$-enforced protocols, BCS exibiths good performance, in terms of reduction of forced checkpoints[7].

[7]As shown in Chapter 5 the total number of checkpoints can be reduced in the case of periodic basic checkpoints by adopting the skipping technique [36]. However, in this section we consider also the case in which checkpoints are not triggered on a periodic basis; this is why the BCS protocol has been selected.

init P_k:
`take a checkpoint;`
$after\_first\_send_k := FALSE;$
$\forall i \;:\; i \neq k \;\; VC_k[i] := 0; \;\; VC_k[k] := 1;$
$\forall i \;\; Max\_Pred_k[i] := -1; \;\; \forall h \;\; Imm\_Pred_k[h] := -1;$

when `m arrives at` P_k `from` P_l:
if $after\_first\_send_k \wedge (\exists i \;:\; (\mathtt{m}.VC[i] > VC_k[i]) \wedge$
$(\exists j \;:\; \mathtt{m}.Max\_Pred[j] + 1 > max(\mathtt{m}.VC[j], VC_k[j])))$
then $take\_ckpt()$; `% forced checkpoint %`
$\forall i \;\; VC_k[i] := max(VC_k[i], \mathtt{m}.VC[i])$; `% component-wise maximum %`
$\forall i \;\; Max\_Pred_k[i] := max(\mathtt{m}.Max\_Pred[i], Max\_Pred_k[i])$;
$Imm\_Pred_k[l] := max(Imm\_Pred_k[l], \mathtt{m}.VC[l])$;

procedure $send(\mathtt{m}, P_j)$:
$\mathtt{m}.content = data; \;\; \mathtt{m}.VC := VC_k; \;\; \mathtt{m}.Max\_Pred := Max\_Pred_k$; `% packet the message %`
send `m` **to** P_j;
$after\_first\_send_k := TRUE$;

when `a basic checkpoint is scheduled from` P_k:
$take\_ckpt()$;

procedure $take\_ckpt()$:
`take a checkpoint;`
$VC_k[k] := VC_k[k] + 1$;
$\forall h \;\; Max\_Pred_k[h] := max(Max\_Pred_k[h], Imm\_Pred_k[h])$; `% component-wise maximum %`
$\forall h \;\; Imm\_Pred_k[h] := -1$;
$after\_first\_send_k := FALSE$

Figure 6.15: Protocol P2.

Simulation Model and Results

The performance comparison studies, for each protocol, the number of forced checkpoints per message receive (R) as a function of the average checkpoint interval size (for example, R equal to 0.2 means a forced checkpoint is taken, on the average, each 5 message receives) under two distinct strategies adopted by the processes for taking basic checkpoints:

- S1 : each process schedules N basic checkpoints periodically and the period between two succesive basic checkpoints is the same at all processes;
- S2 : each process schedules N basic checkpoints randomly distributed in the whole computation (the scheduling of checkpoints follows a distinct distribution at each process).

We simulate an uniform point-to-point environment in which each process can send a message to any other 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 $p_i = 0.9$, $p_s = 0.05$ and $p_r = 0.05$, respectively. The time to execute an operation in a process and the message propagation time are exponentially distributed with mean value equal to 1 and 5 time units respectively.

Let *Average Checkpoint Interval* (ACI) be the average distance, in terms of events, between two basic checkpoints. Experiments were conducted varying ACI from 100 to 10000 events and measuring the value of R. Each simulation run consists of one million of events and for each value of ACI several simulation runs were executed with different seeds and the result were within five percent of each other, thus, variance is not reported in the plots. As we are interested only in counting how many local states are recorded as forced checkpoints by the protocols, the overhead due to the taking of checkpoints is not considered (i.e., in the simulation model the taking of a checkpoint is an istantaneous action). However, we observed that no relevant impact on the obtained measures is noted when considering the time to take a checkpoint longer than zero.

Results of the simulation study are reported in Figure 6.16. We would like to remark that strategy S1 is the most favourable to BCS as the timestamps (i.e., the sequence numbers) increase on average at the same speed at all processes. As an extreme, if all processes would take basic checkpoints at the same physical time, no forced checkpoint will be ever taken. The behaviors of P1 and P2 are flat around 0.01.

Strategy S2 represents a bad scenario for BCS as the distributions of the basic checkpoints at distinct processes are non-correlated. So timestamps increase at different speeds at distinct processes and, then, BCS performance depends on ACI as depicted in Figure 6.16. The behaviors of P1 and P2 are, also in this case, flat and quite close to those under strategy S1. Furthermore, no relevant difference is noted for the value of R of P1 and P2 under both strategies.

From previous plots, a main observation comes out. Performance of both P1 and P2 is more stable compared to the one of BCS with respect to ACI and the basic checkpointing strategy used. This comes from the fact that a $\mathcal{VP}$-accordant protocol is not influenced by the speed a timestamp increases in a process. Its performance depends only on the particular checkpoint and communication subpatterns are going to be formed, which are not directly related to ACI and the strategy used. This makes a $\mathcal{VP}$-accordant protocol particularly appealing to be implemented in a checkpointing layer on a general-purpose system.

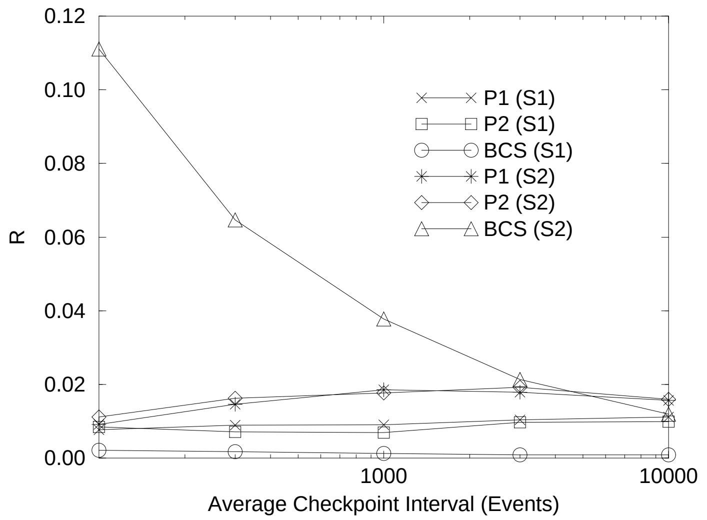


Figure 6.16: R vs. ACI.

6.4 Consistent Global Checkpoints that Contain a Given Local Checkpoint

In this section a distributed protocol for collecting a consistent global checkpoint that contains a specific checkpoint $C_{k,x}$ of process P_k is introduced (such process is the initiator of the distributed protocol). The assumption underlying the protocol is that the checkpoint and communication pattern of the distributed computation satisfies $\mathcal{NZC}$. Before describing the protocol, recall that an ordered pair of checkpoints $(C_{j,y}, C_{k,x})$ is consistent if, and only if, there does not exist any message m such that $C_{j,y} \prec_{ckpt} C_{k,x}$.

By using the notion of consistency of a pair of local checkpoints, the notion of consistent global checkpoint can be reformulated as follows. A global checkpoint GC is consistent if, and only if, every ordered pair of checkpoints in GC is consistent. As an example, in Figure 6.17.b the global checkpoint $GC = \{C_{1,3}, C_{2,2}, C_{3,2}\}$ is consistent, whereas the global checkpoint $GC = \{C_{1,2}, C_{2,2}, C_{3,2}\}$ is not consistent due to the ordered pair $(C_{1,2}, C_{3,2})$.

6.4.1 Consistent Global Checkpoint Collection

We suppose that, when a checkpoint $C_{k,x}$ is taken by P_k, a Tentative-Global-Checkpoint vector $TGC_{k,x}$ of n integers is recorded on stable storage together with $C_{k,x}$. The j-th entry of $TGC_{k,x}$ records the rank associated to a check-

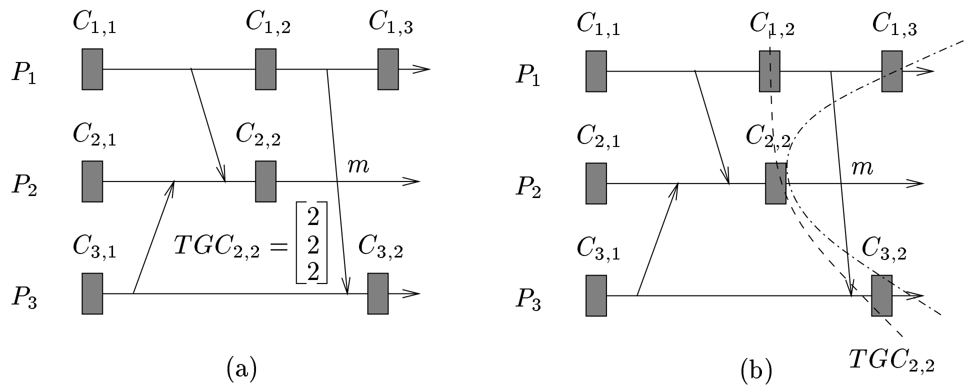


Figure 6.17: Examples of Tentative Global Checkpoints.

point of process P_j. The value of $TGC_{k,x}[j]$ is such that all the ordered pairs of checkpoints $(C_{j,l}, C_{k,x})$ with $l \geq TGC_{k,x}[j]$ are consistent. The k-th entry records the rank of $C_{k,x}$ (that is x). Note that this does not imply $TGC_{k,x}$ identifies a consistent global checkpoint as pairs of checkpoints whose ranks are stored in $TGC_{k,x}$ might be non-consistent. As an example, in Figure 6.17.a $TGC_{2,2} = [2, 2, 2]$ identifies a global checkpoint which is not consistent as the ordered pair $(C_{1,2}, C_{3,2})$ is not consistent.

In order to maintain this information, a local vector V_k of n integers is kept by P_k. All the entries are initialized to -1. When P_k receives a message m from P_j then V_k is updated as follows: $V_k[j] := max(V_k[j], m.VC[j])$. Hence, $V_k[j]$ represents the maximum rank of a checkpoint interval of P_j from which a message received by P_k has been sent. Whenever a checkpoint $C_{k,x}$ is taken, the vector $TGC_{k,x}$ is generated according to the following rules:

(1) $\forall j \neq k \;\; TGC_{k,x}[j] := V_k[j] + 1$;

(2) $TGC_{k,x}[k] := x$.

When a process P_k has to collect a consistent global checkpoint containing $C_{k,x}$, it sends to all the other processes a *checkpoint_collection*(GC_k) message, where GC_k is a copy of $TGC_{k,x}$. The content of GC_k represents P_k's proposal for the consistent global checkpoint containing $C_{k,x}$. In other words, P_k requests to include in the consistent global checkpoint the checkpoint of P_j with rank equal to $GC_k[j]$.

Upon the receipt of the *checkpoint_collection*(GC_k) message, process P_j becomes aware that P_k started a collection which must include $C_{k,GC_k[k]}$ and should include $C_{j,GC_k[j]}$. There are two possible cases:

(1) $\forall h \Rightarrow GC_k[h] \geq TGC_{j,GC_k[j]}[h]$.

procedure *collection_containing*$(C_{k,x})$:

1. $\Pi := \{P_1, \ldots, P_n\} - \{P_k\}$;
2. $GC_k := TGC_{k,x}$;
3. **while** $\Pi \neq \emptyset$
4.   send *checkpoint_collection*(GC_k) to each $P_j \in \pi$;
5.   **wait for** *reply*(GC_j) from all $P_j \in \Pi$;
6. $\Pi := \{P_j \mid \exists l \ : \ GC_l[j] > GC_k[j]\}$;
7. $\forall t \ GC_k[t] := max_{1 \leq l \leq n}(GC_k[t], GC_l[t])$;
8. **endwhile**

when *checkpoint_collection*(GC_j) arrives at P_k from P_j:

9. $\forall l \ GC_k[l] := max(GC_k[l], TGC_{k,GC_j[k]}[l])$;
10. send *reply*(GC_k) to P_j;

Figure 6.18: The Collection Protocol.

In this case, for each $h \neq j$ the ordered pair $(C_{h,GC_k[h]}, C_{j,GC_k[j]})$ is consistent;

(2) $\exists h \neq j \ : \ GC_k[h] < TGC_{j,GC_k[j]}[h]$ (i.e., $\exists m \ : \ C_{h,GC_k[h]} \prec_{ckpt} C_{j,GC_k[j]}$). In this case, there exists at least one checkpoint $C_{h,GC_k[h]}$ requested by P_k such that the ordered pair $(C_{h,GC_k[h]}, C_{j,GC_k[j]})$ is not consistent.

If case (1) is verified for each P_j, then $\{C_{1,GC_k[1]}, \ldots, C_{n,GC_k[n]}\}$ is consistent. If there exists a process P_j which falls in case (2), the global checkpoint $\{C_{1,GC_k[1]}, \ldots, C_{n,GC_k[n]}\}$ is not consistent, hence the original proposal by P_k has to be modified. As an example, considering the execution shown on Figure 6.17.a the proposal $GC_2 = TGC_{2,2} = [2, 2, 2]$, corresponding to a global checkpoint which is not consistent (due to the ordered pair $(C_{1,2}, C_{3,2})$), has to be modified by P_3 in order to include checkpoint $C_{1,3}$ of process P_1 (see Figure 6.17.b). The complete structure of the collection protocol is described in Figure 6.4.1. The protocol executes a sequence of rounds. In each round, the initiator sends its proposal and waits for possible updates. If the proposal was updated, then a new round is started, otherwise the proposal identifies a consistent global checkpoint containing $C_{k,x}$.

As a first action process P_k sends its proposal GC_k to all processes in the set Π (line 4) which initially contains all the processes except P_k (line 1). Then it waits for the reply message, one from each process (line 5). Each reply contains either GC_k or a new proposal formulated by the sender P_j. The new proposal contains local checkpoints that could form a consistent global checkpoint including $C_{k,x}$ and $C_{j,GC_k[j]}$. As an example, considering the computation shown in Figure 6.17.b if process P_3 receives a *checkpoint_collection*(GC_2) message with $GC_2 = TGC_{2,2}$, then it sends back a reply message with $GC_3 = [3, 2, 2]$.

Once collected all the replies, P_k computes (i) the new proposal as the component-wise maximum among all the proposals (line 7) and (ii) the set of processes that changed their checkpoints with respect to the previous proposal done by P_k (line 6). The set and the new proposal correspond to Π and GC_k of the next iteration. The procedure ends when all processes agree on the proposal done by P_k (i.e., $\Pi = \emptyset$ - line 3).

Actually the proposed protocol is a distributed version of the collection protocols presented in [29], therefore, for termination guarantee and correctness the reader can refer to latter paper. The collection protocol in [29] relies on the presence of a checker process. Each time a checkpoint A is taken, the dependency vector associated to that checkpoint is sent to the checker process. Then the checker process examines an $n \times n$ matrix formed by the vector associated to A and vectors received from other processes and computes the global checkpoint which, at the time the matrix is analyzed, contains A and is the closest one to the end of the computation. The major difference between such protocol and the presented one is that the latter does not require exchange of information whenever a checkpoint is taken. On the other hand, is has the disadvantage that the consistent global checkpoint identified is the minimum one containing a given checkpoint.

6.5 Applications of the Presented Protocols

In this section a discussion on two applications of the proposed protocols is presented, posing attention on advantages and disadvantages of the protocols compared to previous solutions.

6.5.1 Recovery from Transient Failures in Long Running Scientific Applications

For long running scientific applications checkpointing is used to reduce the total execution time in the presence of transient failures. As already outlined in Chapter 5, in this context, the goodness of a checkpointing protocol is usually measured in terms of overhead imposed during failure free periods and efficiency of recovery. This latter parameter depends on the amount of information which must be exchanged among processes for determining a consistent global checkpoint from which the application must be restarted after the failure. The selected consistent global checkpoint should be as close as possible to the end of the computation in order to minimize the extent of rollback (i.e., the amount of lost work).

The protocol by Briatico et al. [12] (and also all the existing $\mathcal{VP}$-enforced protocols associating a timestamp to each checkpoint) guarantees that checkpoints timestamped with the same value are members of a consistent global checkpoint. As already discussed in Chapter 3, this feature allows the design of simple and efficient schemes for identifying a consistent global checkpoint containing $C_{k,x}$ for resuming the application [36] which do not need exchange of dependency information. However, unless dependency information is exchanged between the processes, the identified global checkpoint is neither the maximum nor the minimum consistent global checkpoint including $C_{k,x}$.

The checkpointing protocols P1 and P2 presented in this chapter, compared to any $\mathcal{VP}$-enforced protocol, allow smaller checkpointing overhead whenever basic checkpointing strategies at distinct processes are not correlated. From the point of view of recovery, the proposed scheme for identifying a consistent global checkpoint containing a given checkpoint $C_{k,x}$ requires processes to exchange dependency information, furthermore, it identifies the minimum consistent global checkpoint associated to that checkpoint. Therefore, there is no guarantee that the extent of rollback obtained by resuming the execution from that global checkpoint is minimal.

However, as failures are usually rare events, a scheme which reduces the failure free overhead at the expense of the efficiency during recovery is always the best choice. Therefore, it can be concluded that the proposed protocols are well suited in any case there is no a priori knowledge about correlation of basic checkpointing strategies adopted at distinct processes.

6.5.2 The Output Commit Problem

One of the major problems in service-providing application is the output commit. In case of rollback of one of these applications, the maximum extent of rollback is such that no output must be revoked. For example, a printer cannot rollback the effects of printing a character; an automatic machine cannot recover the money it dispensed to a customer; a deleted file cannot be recovered (unless its state is included as part of the checkpoint [53, 66]).

The output commit problem has been tackled in the past assuming piecewise deterministic (PWD) execution model [20, 54]. Under the PWD assumption the execution of a process is seen as a sequence of state intervals. A new state interval starts whenever a non-deterministic event occurs (for example the receive of a message). All non-deterministic events are logged so that the process can always reply its execution from its last taken checkpoint.

An output is recorded in a checkpoint $C_{k,x}$ of P_k if the output message is sent in a checkpoint interval $I_{k,x-\epsilon}$ with $\epsilon > 0$. Recently, Wang has shown [64] that the output commit problem can be translated into the problem of determining the minimum consistent global checkpoint recording all the outputs. This problem can be easily solved through $\mathcal{RDT}$ as this property allows a process P_k to associate on-the-fly to a checkpoint $C_{k,x}$ the minimum consistent global checkpoint containing it.

More technically, as already outlined in Chapter 3, if $\mathcal{RDT}$ is satisfied, then the minimum consistent global checkpoint associated to a given local checkpoint $C_{k,x}$ is easily computed on-the-fly at the time $C_{k,x}$ is taken by piggybacking on each message a transitive dependency vector. As shown by Mattern [38], the union of consistent global checkpoints generates a consistent global checkpoint (such a result has been shown by Mattern to hold for the set of consistent global states of a computation; as the set of consistent global checkpoints is a subset of the set of consistent global states, then the result also holds for consistent global checkpoints). Therefore, the minimum consistent global checkpoint recording all the outputs can be computed by:

(i) collecting vectors identifying the consistent global checkpoint associated to the earliest checkpoint recording an output at each process, and

(ii) performing a component-wise maximum among all collected vectors.

But, which is the cost incurred to ensure the $\mathcal{RDT}$ property? If $(\widehat{\mathcal{H}}, \mathcal{C}_{\widehat{\mathcal{H}}})$ satisfies the $\mathcal{RDT}$ property, then it also satisfies the $\mathcal{NZC}$ property. As already discussed, this implication between properties usually implies that protocols which ensure the $\mathcal{RDT}$ induce processes to take more forced checkpoints compared to protocols ensuring $\mathcal{NZC}$. As a quantitative example of the performance distance in terms of forced checkpoints we report in Figure 6.19 the ratio R (i.e., forced checkpoints by a message receive), measured in the same simulation environment described in Section 6.3.6 of this chapter, for the case of the protocol presented by Baldoni et al. in [4] (BHMR), which has been demonstrated through previous performance studies to be the one inducing less forced checkpoints to guarantee $\mathcal{RDT}$ (results are reported for both basic checkpointing strategies S1 and S2 described in section 6.3.6). The obtained data are compared to those obtained with the proposed protocol P1.

There is a distance of an order of magnitude between values of R obtained with the protocol P1 and those obtained by BHMR (values of R for the protocol P2, being very similar to those of P1, are not reported in Figure 6.19).

Plots demonstrate that the usage of system resources spent for checkpointing can be reduced by using one of the two checkpointing protocols presented in this chapter. The drawback incurred is that to a local checkpoint cannot be associated on-the-fly the minimum consistent global checkpoint containing it. However, the acceptability of this drawback is justified by the following observation: the minimum consistent global checkpoint containing $C_{k,x}$ has to be identified only if $C_{k,x}$ is the earliest checkpoints recording the last output produced by P_k. $\mathcal{RDT}$ guarantees such identification on-the-fly for any checkpoint but at the expense of sometimes unacceptable checkpointing overhead. In order to avoid such an overhead one of the checkpointing protocols (P1

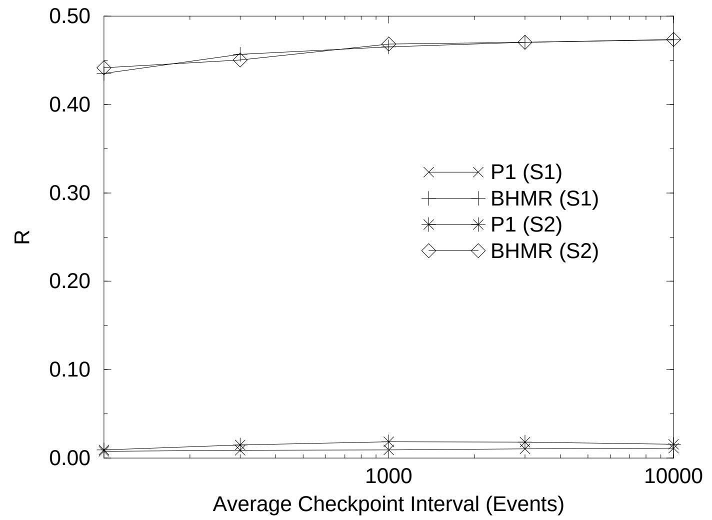


Figure 6.19: R vs. ACI.

or P2) here proposed can be adopted and, periodically the consistent global checkpoint collection protocol presented can be run in order to associate to the earliest checkpoint $C_{k,x}$ recording an output a consistent global checkpoint including it. The ranks of checkpoints identified during the collection can be recorded onto stable storage in a vector $GC_{k,x}$ associated to $C_{k,x}$. This vector is then used whenever the minimum consistent global checkpoints recording all the outputs is reclaimed.

Chapter 7

Consistent Checkpointing in Distributed Databases

Checkpointing the state of a database is important for audit or recovery purposes. When compared to its counterpart in distributed computations, the database checkpointing problem has additionally to take into account the serialization order of the transactions that manipulates the data objects forming the database. Actually, transactions create dependences among data objects which makes harder the problem of defining *consistent* global checkpoints in database systems.

Of course, it is always possible, in a database environment, to design a special transaction, that reads all data objects and saves their current values. The underlying concurrency control mechanism ensures that this transaction gets a consistent state of the data objects. However, this strategy is inefficient, intrusive (from the concurrency control point of view [52]) and not practical since, a read only transaction may take a very long time to execute and may cause intolerable delays for other transactions [41]. Moreover, as pointed out in [51], this strategy may drastically increase the cost of rerunning aborted transactions. So, it is preferable to base global checkpointing:

(1) on local checkpoints of data objects taken by their managers, and

(2) on a mechanism ensuring mutual consistency of local checkpoints (this will ensure that it will always be possible to get consistent global checkpoints by piecing together local checkpoints).

In this chapter, latter approach to checkpointing is explored. The considered database is such that each data object can be individually checkpointed (note that a data object could include, practically, a set of physical data items). If these checkpoints are taken in an independent way, there is the risk that no consistent global checkpoint can ever be formed, similarly to what happens in distributed computations. So, some kind of coordination is necessary when local checkpoints are taken in order to ensure their mutual consistency.

This chapter introduces a characterization of mutual consistency of local checkpoints. More precisely, the two following issues are considered:

- let us consider the question $\bar{\mathcal{Q}}(S)$: "Given an arbitrary set S of checkpoints of data objects, can this set be extended to get a global checkpoint (i.e., a set including exactly one checkpoint from each data object) that is consistent?". The answer to this question is well known when the set S includes exactly one checkpoint per data object [41], it becomes far from being trivial, when the set S is incomplete, i.e., when it includes checkpoints from only a subset of data objects. When S includes a single data checkpoint, the previous question is equivalent to "Can this local checkpoint belong to a consistent global checkpoint?".

- let us consider the property $\mathcal{P}(C)$: "Local checkpoint C belongs to a consistent global checkpoint". Two non-intrusive checkpointing protocols are introduced, the first one ensures the previous property $\mathcal{P}$ when C is any local checkpoint of a data object. The second one ensures $\mathcal{P}$ when C belongs to a predefined set of local checkpoints of a data object.

$\bar{\mathcal{Q}}(S)$ is analogous to question $\mathcal{Q}(S)$ stated in Chapter 2, to which the answer has been provided by Netzer and Xu. To provide an answer to question $\bar{\mathcal{Q}}(S)$, this chapter presents a study on the kind of dependences both the transactions and their serialization order create among checkpoints of distinct data objects. Therefore, the direction pointed out in [11], where it is said that "Although the problems of concurrency control and recoverability are frequently discussed separately, they are actually closely related" is investigated. More specifically, in this chapter it is shown that, while some data checkpoint dependences are causal, and consequently can be captured on-the-fly, some others are "hidden", in the sense that, they cannot be revealed by causality (analogously to what happens for dependences between checkpoints of processes of a distributed computation due to the presence of non-causal Z-paths). It is the existence of those hidden dependencies that actually makes non-trivial the answer to the previous question. Such an answer is here provided by exploiting concepts of the Netzer-Xu theory properly redefined and enriched for the context of databases.

Starting from the obtained theoretical results, Section 7.5 of this chapter is devoted to the design of "transaction-induced" data checkpointing protocols ensuring the property $\mathcal{P}$ (namely, "Local checkpoint C belongs to a consistent global checkpoint"). These protocols allow managers of data objects to take checkpoints independently on each other[1] (these checkpoints are called *basic* as in the context of communication-induced checkpointing protocols for distributed computations), and use transactions as a means to diffuse information, among data managers, encoding dependences on the previous states of data objects. When a transaction that accessed a data object is committed, the data manager of this object may be directed to take a checkpoint to guarantee that previously taken checkpoints belong to consistent global checkpoints (as in the context of communication-induced checkpointing, such a checkpoint is called *forced* checkpoint). This is done by the data manager which exploits both its local control data and the information exchanged at the transaction commit point. The presented protocols are actually adaptations to the context of distributed databases of the protocols by Briatico et al. [12] and the protocol by Wang and Fuchs [61].

7.1 Database Model

We consider a classical distributed database model. The system consists of a finite set of data objects, a set of transactions and a concurrency control mechanism [10, 26].

7.1.1 Data Objects

Each data object is managed by a data manager DM. A set of data objects can be managed by the same data manager DM. For the sake of clarity, we suppose that the set of data managed by the same DM constitutes a single logical data. So, there is a data manager DM_x per data x ([2]).

7.1.2 Transactions

A transaction is defined as a partial order on *read* and *write* operations on data objects and terminates with a *commit* or an *abort* operation. $R_i(x)$ (resp. $W_i(x)$) denotes a read (resp. write) operation issued by transaction T_i on data object x. Each transaction is managed by an instance of the transaction manager (TM) that forwards its operations to the scheduler which runs a specific concurrency control protocol. The write set of a transaction is the set of all the data objects it wrote.

[1] They can be taken, for example, during CPU idle time.

[2] Notations adopted in this chapter slightly differ from those of previous chapters. As an example, x, y and z denote here data objects instead of ranks of checkpoints. Furthermore, as it will be clear later, checkpoints of data objects are identified by a subscript and a superscript, instead of a subscript only.

7.1.3 Concurrency control

A concurrency control protocol schedules read and write operations issued by transactions in such a way that any execution of transactions is *strict* and *serializable*. This is not a restriction as concurrency control mechanisms used in practice (e.g., two-phase locking 2PL and timestamp ordering) generate schedules ensuring both properties [11]. The *strictness* property states that no data object may be read or written until the transaction that currently writes it either commits or aborts. So, a transaction actually writes a data object at its commit point. Hence, at some abstract level, which is the one considered by our checkpointing mechanisms, transactions execute atomically at their commit points. If a transaction is aborted, strictness ensures no cascading aborts and the possibility to use *before images* for implementing abort operations which restore the value of an object before the transaction access. For example, a 2PL mechanism, that requires transactions to keep their write locks until they commit (or abort), generates such a behavior [11].

7.2 Distributed Database

A distributed database consists of a finite set of sites, each site containing one or several (logical) data objects. So, each site contains one or more data managers, and possibly an instance of the TM. TMs and DMs exchange messages on a communication network which is asynchronous (message transmission delays are unpredictable but finite) and reliable (each message will eventually be received).

7.2.1 Execution

Let $T = \{T_1, \ldots, T_n\}$ be a set of transactions accessing a set $O = \{o_1, \ldots, o_m\}$ of data objects (to simplify notations, data object o_i is identified by its index i). An execution E over T is a partial order on all read and write operations of the transactions belonging to T; this partial order respects the order defined in each transaction. Moreover, let $<_x$ be the partial order defined on all operations accessing a data object x, i.e., $<_x$ orders all pairs of conflicting operations (two operations are conflicting if they access the same object and one of them is a write operation).

Given an execution E defined over T, T is structured as a *partial order* $\widehat{T} = (T, <_T)$ where $<_T$ is the following (classical) relation defined on T:

$$
\begin{aligned}
T_i <_T T_j \quad \Longleftrightarrow \quad & (i \neq j) \wedge (\exists x \Rightarrow (R_i(x) <_x W_j(x)) \vee \\
& (W_i(x) <_x W_j(x)) \vee (W_i(x) <_x R_j(x)))
\end{aligned}
$$

7.3 Consistent Global Checkpoints

This section is devoted to the introduction of the notion of consistent global checkpoints of the distributed database. This is done by recalling the notion of dependence between states of data objects.

7.3.1 Local States and Their Relations

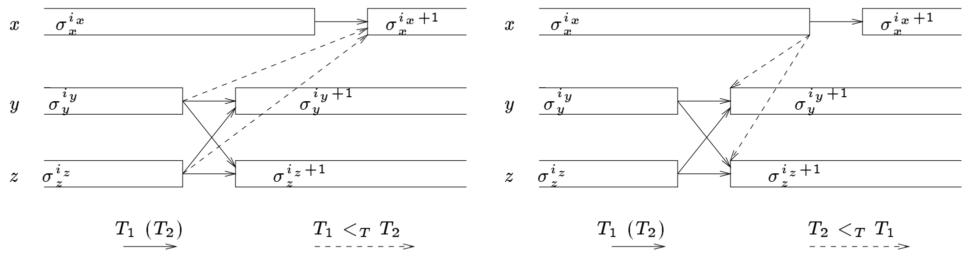
Each write on a data object x issued by a transaction defines a new version of x. Let σ_x^i denote the i-th version of x; σ_x^i is called a *local state* (σ_x^1 is the initial local state of x). Transactions establish dependences between local states. This can be formalized in the following way. When T_k issues a write operation $W_k(x)$, it changes the state of x from σ_x^i to σ_x^{i+1}. More precisely, σ_x^i and σ_x^{i+1} are the local states of x, just before and just after the execution[3] of T_k, respectively. This can be expressed in the following way by extending the relation $<_T$ to include local states:

$$T_k \; changes \; x \; from \; \sigma_x^i \; to \; \sigma_x^{i+1} \iff (\sigma_x^i <_T T_k) \wedge (T_k <_T \sigma_x^{i+1})$$

Let $<_T^+$ be the transitive closure of the extended relation $<_T$. When we consider only local states, we get the following *happened-before* relation denoted $<_{LS}$ (which is similar to Lamport's happened-relation defined on events [33] in a distributed computation):

Definition 7.3.1 *(Precedence on local states, denoted $<_{LS}$)*

$$\sigma_x^i \;\; <_{LS} \sigma_y^j \iff \sigma_x^i <_T^+ \sigma_y^j$$



(a) *T_1 precedes T_2* (b) *T_2 precedes T_1*

Figure 7.1: Partial Order on Local States.

As the relation $<_T$ defined on transactions is a partial order, it is easy to see that the relation $<_{LS}$ defined on local states is also a partial order. Figure

[3]Remind that, as we consider strict and serializable executions, "Just before and just after the execution of T_k" means "Just before and just after T_k is committed".

7.1 shows examples of relation $<_{LS}$. It considers three data objects x, y, and z, and two transactions T_1 and T_2. Transactions are defined in the following way:

$$T_1 : \; R_1(x);\; W_1(y);\; W_1(z);\; commit_1$$
$$T_2 : \; R_2(y);\; W_2(x);\; commit_2$$

As there is a read-write conflict on x, two serialization orders are possible. Figure 7.1.a shows the case $T_1 <_T T_2$ while Figure 7.1.b shows the case $T_2 <_T T_1$. Each horizontal axis depicts the evolution of the state of a data object. For example, the second axis is devoted to the evolution of y: $\sigma_y^{i_y}$ and $\sigma_y^{i_y+1}$ are the states of y before and after T_1, respectively.

Let us consider Figure 7.1.a. It shows that $W_1(y)$ and $W_1(z)$ add four pairs of local states to the relation $<_{LS}$, namely:

$$\sigma_y^{i_y} <_{LS} \sigma_y^{i_y+1};\; \sigma_z^{i_z} <_{LS} \sigma_z^{i_z+1};\; \sigma_y^{i_y} <_{LS} \sigma_z^{i_z+1};\; \sigma_z^{i_z} <_{LS} \sigma_y^{i_y+1}$$

The relation $<_T$ adds two pairs of local states to $<_{LS}$:

$$\sigma_y^{i_y} <_{LS} \sigma_x^{i_x+1};\; \sigma_z^{i_z} <_{LS} \sigma_x^{i_x+1}$$

The latter two dependences are due to the serialization order.

Precedence on local states, due to write operations of transactions T_1 and T_2, are indicated with continuous arrows, while the ones due to the serialization order are indicated with dashed arrows. Figure 7.1.b shows precedences on local states when the serialization order is reversed.

7.3.2 Consistent Global States

A *global state* of the database is a set of local states, one from each data object. A global state $G = \{\sigma_1^{i_1}, \sigma_2^{i_2}, \ldots, \sigma_m^{i_m}\}$ is *consistent* if it does not contain two dependent local states, i.e., if:

$$\forall x, y \in [1, \ldots, m] \Rightarrow \neg(\sigma_x^{i_x} <_{LS} \sigma_y^{i_y})$$

Let us consider again Figure 7.1.a. The three global states $(\sigma_x^{i_x}, \sigma_y^{i_y}, \sigma_z^{i_z})$, $(\sigma_x^{i_x}, \sigma_y^{i_y+1}, \sigma_z^{i_z+1})$ and $(\sigma_x^{i_x+1}, \sigma_y^{i_y+1}, \sigma_z^{i_z+1})$ are consistent. The global state $(\sigma_x^{i_x+1}, \sigma_y^{i_y}, \sigma_z^{i_z+1})$ is not consistent either because $\sigma_y^{i_y} <_{LS} \sigma_x^{i_x+1}$ (due to the fact $T_1 <_T T_2$) or because $\sigma_y^{i_y} <_{LS} \sigma_z^{i_z+1}$ (due to the fact T_1 writes both y and z). Intuitively, a non-consistent global state of the database is a global state that could not be seen by any omniscient observer of the database.

7.3.3 Consistent Global Checkpoints

A *local checkpoint* (or equivalently a *data checkpoint*) of a data object x is a local state of x that has been saved in a safe place[4] by the data manager of x. So, all the local checkpoints are local states, but only a subset of local states are defined as local checkpoints. Let C_x^i $(i \geq 1)$ denote the i-th local checkpoint of x; i is called the rank of C_x^i ([5]). Note that C_x^i corresponds to some σ_x^j with $i \leq j$. A *global checkpoint* is a set of local checkpoints one for each data object. It is *consistent* if it is a consistent global state.

We assume that all initial local states are checkpointed. Moreover, we also assume that, when we consider any point of an execution E, each data object will eventually be checkpointed.

7.4 Extension of Netzer-Xu Theory to Distributed Databases

This section extends the Netzer-Xu theory to distributed databases. This is done by introducing the notion of *Dependence Path* on data checkpoints, which is analogous to the Z-path on checkpoints of process states in distributed computations. Then the theorem stating the necessary and sufficient condition for mutual consistency is proved. The structure of the proof of the theorem is similar to the one of a theorem presented in [2] which proves an analogous result for the case of shared memory.

7.4.1 Dependence on Data Checkpoints

As indicated in the previous section, due to write operations of each transaction, or due to the serialization order, transactions create dependences among local states of data objects. Let us consider the following 7 transactions accessing data objects x, y, z and u:

$$\begin{array}{l} T_1 : \ R_1(u);\ W_1(u);\ commit_1 \\ T_2 : \ R_2(z);\ W_2(z);\ commit_2 \\ T_3 : \ R_3(z);\ W_3(z);\ W_3(x);\ commit_3 \\ T_4 : \ R_4(z);\ R_4(u);\ W_4(z);\ commit_4 \\ T_5 : \ R_5(z);\ W_5(y);\ W_5(z);\ commit_5 \\ T_6 : \ R_6(y);\ W_6(y);\ commit_6 \\ T_7 : \ R_7(x);\ W_7(x);\ commit_7 \end{array}$$

[4] For example, if x is stored on a disk, a copy is saved on another disk.

[5] Checkpoints of data objects are denoted by a subscript and a superscript in order to distinguish them from checkpoints of process states.

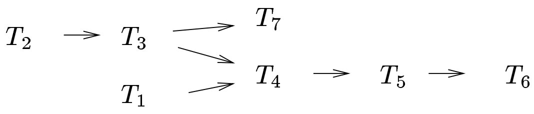


Figure 7.2: A Serialization Order.

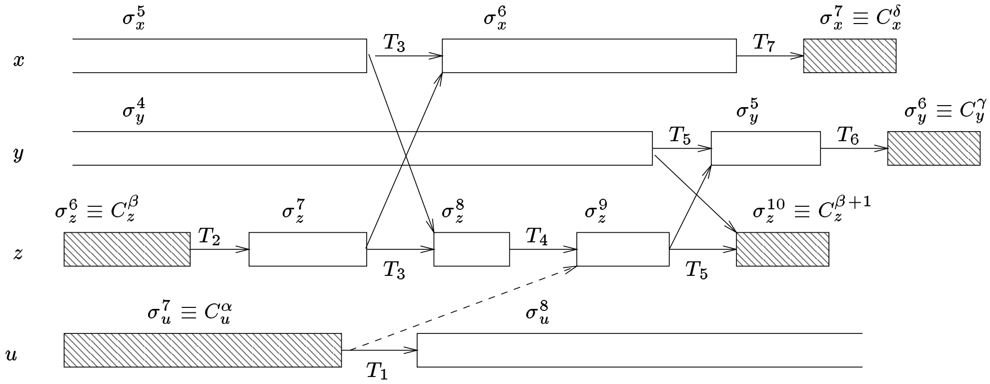


Figure 7.3: Data Checkpoint Dependences.

Figure 7.2 depicts the serialization imposed by the concurrency control mechanism. Figure 7.3 describes dependences between local states generated by this execution. Five local states are defined as data checkpoints (they are indicated by dark rectangles). We study dependences between those data checkpoints. Let us first consider C_u^α and C_y^γ. C_u^α is the (checkpointed) state of u before T_1 wrote it, while C_y^γ is the (checkpointed) state of y after T_6 wrote it (i.e., just after T_6 is committed). The serialization order (see Figure 7.2) shows that $T_1 <_T T_6$, and consequently $C_u^\alpha <_{LS} C_y^\gamma$, i.e., the data checkpoint C_y^γ is causally dependent [33] on the data checkpoint C_u^α (Figure 7.3 shows that there is a directed path from C_u^α to C_y^γ). Now let us consider the pair of data checkpoints consisting of C_u^α and C_x^δ. Figure 7.3 shows that C_u^α precedes T_1, and that C_x^δ follows T_7. Figure 7.2 indicates that T_1 and T_7 are not connected in the serialization graph. So, there is no causal dependence between C_u^α and C_x^δ (Figure 7.3 shows that there is no directed path from C_u^α to C_x^δ). But there is no consistent global checkpoint including both C_u^α and C_x^δ. In particular, adding C_y^γ and C_z^β to C_u^α and C_x^δ cannot produce a consistent global state as $C_z^\beta <_{LS} C_x^\delta$; adding $C_z^{\beta+1}$ instead of C_z^β has the same effect as $C_u^\alpha <_{LS} C_z^{\beta+1}$. So there is a *hidden* dependence between C_u^α and C_x^δ which prevents them to belong to the same consistent global checkpoint.

7.4.2 Dependence Path

In this section an unified definition of dependence is provided which takes into account both causal and hidden dependences.

Definition 7.4.1 *(Interval)*
A checkpoint interval I_x^i is associated with data checkpoint C_x^i. It consists of all the local states σ_x^k such that:

$$(\sigma_x^k = C_x^i) \vee (C_x^i <_{LS} \sigma_x^k <_{LS} C_x^{i+1})$$

As an example, Figure 7.3 shows that I_z^β includes 4 consecutive local states of z. Note that, due to the assumptions on data checkpoints stated in Section 7.3.3, any local state belongs to exactly one interval. Let us call an edge of the partial order on local states ($<_{LS}$) a *dependence edge.*

Definition 7.4.2 *(Dependence Path)*[6]
There is a dependence path (DP) from a data checkpoint C_x^i to C_y^j (denoted $C_x^i \stackrel{DP}{\rightarrow} C_y^j$) iff:
(i) $x = y$ and $i < j$; or
(ii) there is a sequence $(d_1, d_2, \ldots, d_r)$ of dependence edges, such that:

(1) d_1 starts after C_x^i;

(2) $\forall d_q$: $1 \leq q < r$: let I_z^k be the interval in which d_q arrives; then d_{q+1} starts in the same or in a later interval (i.e., an interval I_z^h such that $k \leq h$)[7]*;*

(3) d_n arrives before C_y^j.

In the example depicted in Figure 7.3, the hidden dependence between C_u^α and C_x^δ can be now denoted $C_u^\alpha \stackrel{DP}{\rightarrow} C_x^\delta$ as $C_u^\alpha = \sigma_u^7 <_{LS} \sigma_z^9$ (due to relation $<_T$), $\sigma_z^7 <_{LS} \sigma_x^6$ and $\sigma_x^6 <_{LS} \sigma_x^7 = C_x^\delta$. Note that σ_z^9 and σ_z^7 belong to the same checkpoint interval I_z^β.

[6]This definition generalizes the Z-path notion introduced in [40]. Recall that a Z-path is a sequence of messages establishing a relation between two checkpoints of distinct processes. While a message is a "concrete entity", a dependence edge is an "abstract entity". So, as it will be shown by the theorem in next section, the *dependence edge* abstraction allows to extend results of [40] to data checkpoints.

[7]Note that d_{q+1} can "start" before d_q "arrives". This is where the dependence is "hidden". If $\forall q$ d_{q+1} "starts" after d_q "arrives", then, the dependence path $(d_1, d_2, \ldots, d_r)$ is purely causal.

7.4.3 Necessary and Sufficient Condition

Theorem 7.4.1
Let $\mathcal{I} \subseteq \{1, \ldots, m\}$ and $\mathcal{S} = \{C_x^{i_x}\}_{x \in \mathcal{I}}$ be a set of data checkpoints. Then $\mathcal{S}$ is a part of a consistent global checkpoint if and only if:

$$\mathcal{R} \equiv \forall x, y \in \mathcal{I} \Rightarrow \neg(C_x^{i_x} \overset{DP}{\to} C_y^{i_y})$$

Proof
If Part. It is proved that if $\mathcal{R}$ is satisfied then $\mathcal{S}$ can be included in a consistent global checkpoint. Let us consider the global checkpoint defined as follows:

- if $x \in \mathcal{I}$, we take $C_x^{i_x}$;
- if $x \notin \mathcal{I}$, for each $y \in \mathcal{I}$ we consider the integer $m_x(y) = \min\{i \mid \neg(C_x^i \overset{DP}{\to} C_y^{i_y})\}$ (with $m_x(y) = 1$ if $i_y = 1$ or if this set is empty). Then we take $C_x^{i_x}$ with $i_x = \max_{y \in \mathcal{I}}(m_x(y))$. Let us note that, from that definition, it is possible that $i_x = 1$ (in that case, $C_x^{i_x}$ is an initial data checkpoint).

By construction, this global checkpoint satisfies the two following properties :

$$\forall x \notin \mathcal{I},\ \forall y \in \mathcal{I} \Rightarrow \neg(C_x^{i_x} \overset{DP}{\to} C_y^{i_y}) \tag{7.1}$$

$$\forall x \notin \mathcal{I} \text{ such that } i_x > 1,\ \exists z \in \mathcal{I}\ :\ (i_z > 1) \wedge (C_x^{i_x - 1} \overset{DP}{\to} C_z^{i_z}) \tag{7.2}$$

We show that $\{C_1^{i_1}, C_2^{i_2}, \ldots, C_m^{i_m}\}$ is consistent. Assume the contrary. So, there exists x and y and a dependence edge d that starts after $C_x^{i_x}$ and arrives before $C_y^{i_y}$. So, it follows that:

$$(i_y > 1) \wedge (C_x^{i_x} \overset{DP}{\to} C_y^{i_y}) \tag{7.3}$$

Four cases have to be considered:

1. $x \in \mathcal{I}$, $y \in \mathcal{I}$. (7.3) is contradicted by assumption $\mathcal{R}$.
2. $x \in \mathcal{I}$, $y \notin \mathcal{I}$. Since $i_y > 1$, from (7.2) we have: $\exists z \in \mathcal{I}\ :\ (i_z > 1) \wedge (C_j^{i_y - 1} \overset{DP}{\to} C_z^{i_z})$.

   As, at data x both the dependence edge ending the path $C_x^{i_x} \overset{DP}{\to} C_y^{i_y}$, and the dependence edge starting the path $C_y^{i_y - 1} \overset{DP}{\to} C_z^{i_z}$ belong to the same interval, we conclude from (7.2) that $\exists z \in \mathcal{I}\ :\ (i_z > 1) \wedge (C_x^{i_x} \overset{DP}{\to} C_z^{i_z})$ which contradicts the assumption $\mathcal{R}$.
3. $x \notin \mathcal{I}$, $y \in \mathcal{I}$. (7.3) contradicts (7.1).

4. $x \notin \mathcal{I}$, $y \notin \mathcal{I}$. Since $i_y > 0$, from (7.2) we have: $\exists z \in \mathcal{I} \ : \ (i_z > 1) \wedge (C_y^{i_y - 1} \stackrel{DP}{\to} C_z^{i_z})$.

   As in case 2, we can conclude that $\exists z \in \mathcal{I} \ : \ (i_z > 1) \wedge (C_x^{i_x} \stackrel{DP}{\to} C_z^{i_z})$ which contradicts (7.1).

Only If Part. It is proved that, if there is a consistent global checkpoint $\{C_1^{i_1}, C_2^{i_2}, \ldots, C_n^{i_n}\}$ including $\mathcal{S}$, then $\mathcal{R}$ holds for any $\mathcal{I} \subseteq \{1, \ldots, m\}$. Assume the contrary. So, there exist $x \in \mathcal{I}$ and $y \in \mathcal{I}$ such that $(C_x^{i_x} \stackrel{DP}{\to} C_y^{i_y})$. From the definition of $\stackrel{DP}{\to}$, there exists a sequence of dependence edges $d_1, d_2, \ldots, d_p$ such that:

$$
\begin{array}{lll}
 & d_1 \ starts \ in \ I_x^{i_x}, & \\
d_1 \ arrives \ after \ I_{x_1}^{i_1}, & d_2 \ starts \ in \ I_{x_1}^{j_1} & \text{with } i_1 \leq j_1 \\
 & \ldots & \\
d_{p-1} \ arrives \ in \ I_{x_{p-1}}^{i_{p-1}}, & d_p \ starts \ in \ I_{x_{p-1}}^{j_{p-1}} & \text{with } j_{p-1} \leq i_{p-1} \\
d_p \ arrives \ in \ I_y^{i_y - 1} & &
\end{array}
$$

We show by induction on p that, $\forall t \geq i_y$, $C_x^{i_x}$ and C_y^t cannot belong to the same consistent global checkpoint.

Base step. $p = 1$. In this case, d_1 starts after $C_x^{i_x}$ and arrives before $C_y^{i_y}$, and consequently the pair $(C_x^{i_x}, C_y^{i_y})$ cannot belong to a consistent global checkpoint.
Induction step. We suppose the result true for some $p \geq 1$ and show that it holds for $p + 1$. We have:

$$
\begin{array}{lll}
 & d_1 \ starts \ in \ I_x^{i_x}, & \\
 & \ldots & \\
d_p \ arrives \ in \ I_{x_p}^{i_p}, & d_{p+1} \ starts \ in \ I_{x_p}^{j_p} & \text{with } i_p \leq j_p \\
d_{p+1} \ arrives \ in \ I_y^{i_y - 1} & &
\end{array}
$$

From the assumption induction applied to the path of dependence edges $d_1, \ldots, d_p$, we have: for any $t \geq i_p + 1$, $C_x^{i_x}$ and $C_{x_p}^t$ cannot belong to the same consistent global checkpoint. Moreover, d_{p+1} starts in $I_{x_p}^{j_p}$ and arrives in $I_y^{i_y - 1}$ imply that, for any $h \leq j_p$ and for any $t \geq i_y$, $C_{x_p}^h$ and C_y^t cannot belong to the same consistent checkpoint. Since $i_p \leq j_p$, it follows that no checkpoint of x_p can be included with $C_x^{i_x}$ and $C_y^{i_y}$ to form a consistent global checkpoint.

Q.E.D.

7.5 Deriving "Transaction-Induced" Checkpointing Protocols

This section shows how previous theoretical results can be exploited to derive checkpointing protocols for distributed databases.

Supposing that the set S includes only a checkpoint C of a data object, the previous theorem leads to an interesting corollary:

Corollary 7.5.1
C belongs to a consistent global checkpoint iff $\neg(C \stackrel{DP}{\rightarrow} C)$.

Hence, providing checkpointing protocols ensuring that $\neg(C \stackrel{DP}{\rightarrow} C)$, guarantees the property $\mathcal{P}(C)$ defined at the beginning of this chapter. These type of protocols are interesting for two reasons:

1. They avoid wasting time in taking a data checkpoint that will never be used in any consistent global checkpoint, and

2. In case checkpointing is used for recovery purposes, no domino effect can ever take place as any data checkpoint belongs to a consistent global checkpoint.

To this purpose, let us assume that to each checkpoint C_x^i is associated a sequence number, denoted $C_x^i.sn$, and that that each data manager DM_x has a variable sn_x, which stores the sequence number of the last checkpoint of x (it is initialized to zero); furthermore, let i_x denotes the rank of the last checkpoint of x.

Consider the following property $\mathcal{TS}$: "Let $\mathcal{S}_n$ be the set formed by data checkpoints with sequence number n. If $\mathcal{S}_n$ includes a checkpoint per data object, then it constitutes a consistent global checkpoint". In what follows two checkpointing protocols are provided:

- the first protocol ($\mathcal{A}$) guarantees $\mathcal{P}$ for all local checkpoints, and guarantees $\mathcal{TS}$ for any value of n.

- The second protocol ($\mathcal{B}$) ensures $\mathcal{P}$ only for a subset of local checkpoints, and $\mathcal{TS}$ for some particular values of n.

As already mentioned, actually those protocols can be seen as adaptations (to the data-object/transaction model) of protocols in [12, 61].

In the proposed protocols, data managers can take checkpoints independently of each other (*basic checkpoints*), for example, by using a periodic algorithm which could be implemented by associating a timer with each data manager (a local timer is set whenever a checkpoint is taken; and a basic checkpoint is taken by the data manager when its timer expires). Data managers are directed to take additional data checkpoints (*forced checkpoints*) in order to ensure $\mathcal{P}$ or $\mathcal{TS}$. The decision to take forced checkpoints is based on the control information piggybacked by commit messages of transactions.

The protocols consist of two interacting parts. The first part, shared by both protocols, specifies the checkpointing-related actions of transaction managers. The second part defines the rules data managers have to follow to take data checkpoints.

7.5.1 Protocols $\mathcal{A}$ and $\mathcal{B}$: Behavior of a Transaction Manager

Let W_{T_i} be the write set of a transaction T_i managed by a transaction manager TM_i. We assume each time an operation of T_i is issued by TM_i to a data manager DM_x, it returns the value of x plus the value of its current sequence number sn_x. TM_i stores in $MAX\_SN_{T_i}$ the maximum value among the sequence numbers of the data objects read or written by T_i. When transaction T_i is committed, the transaction manager TM_i sends a *commit* message to each data manager DM_x involved in W_{T_i}. Such *commit* messages piggyback $MAX\_SN_{T_i}$.

7.5.2 Protocol $\mathcal{A}$: Behavior of a Data Manager

As far as checkpointing is concerned, the behavior of a data manager DM_x is defined by the two following procedures namely `take-basic-ckpt` and `take-forced-ckpt`. They define the rules associated with checkpointing.

`take-basic-ckpt`($\mathcal{A}$) :

> **When** the timer expires:
> > (AB1) $i_x \leftarrow i_x + 1$; $sn_x \leftarrow sn_x + 1$;
> > (AB2) Take checkpoint $C_x^{i_x}$; $C_x^{i_x}.sn \leftarrow sn_x$;
> > (AB3) Reset the local timer.

`take-forced-ckpt`($\mathcal{A}$) :

> **When** DM_x **receives** *commit*$(MAX\_SN_{T_i})$ **from** TM_i:
> > **if** $sn_x < MAX\_SN_{T_i}$ **then**
> > > (A1) $i_x \leftarrow i_x + 1$; $sn_x \leftarrow MAX\_SN_{T_i}$;
> > > (A2) Take a (forced) checkpoint $C_x^{i_x}$;
> > > $C_x^{i_x}.sn \leftarrow sn_x$;
> > > (A3) Reset the local timer.
> >
> > **endif**;
> > (A4) process the *commit* message.

From the increase of the timestamp variable sn_x of a data object x, and from the rule associated with the taking of forced checkpoints (which forces a data checkpoint whenever $sn_x < MAX\_SN_{T_i}$), the condition $\neg(C_x^{i_x} \stackrel{DP}{\rightarrow} C_x^{i_x})$ follows for any data checkpoint $C_x^{i_x}$. Actually, this simple protocol ensures that, if $C_x^{i_x} \stackrel{DP}{\rightarrow} C_y^{i_y}$, then $C_x^{i_x}.sn < C_y^{i_y}.sn$ (analogously to the protocols in [12, 36, 28] discussed in Chapter 3).

It follows from the previous observation that if two data checkpoints have the same sequence number, then they cannot be related by $\stackrel{DP}{\rightarrow}$. So, all the sets $\mathcal{S}_n$ that exist are consistent. Note that the `take-forced-ckpt(`$\mathcal{A}$`)` rule may produce gaps in the sequence of timestamps assigned to data checkpoints of a data object x. When no data checkpoint of a data object x has sequence number n, then the first data checkpoint of x with sequence number greater than n can be included in a set containing data checkpoints with sequence number n, to form a consistent global checkpoint (analogously to what happens in checkpoint and communication patterns of distributed computations when considering the protocol by Briatico et al. [12]).

7.5.3 Protocol $\mathcal{B}$: Behavior of a Data Manager

This protocol introduces a system parameter $Z \geq 1$ known by all the data managers [61]. When considering a data object x, this protocol ensures $\neg(C_x \stackrel{DP}{\rightarrow} C_x)$ only for a subset of data checkpoints, namely, those whose sequence numbers are equal to $a \times Z$ (where $a \geq 0$ is an integer). Moreover, when there is a data checkpoint with sequence number $a \times Z$ for each data object x, then the global checkpoint $\mathcal{S}_{aZ}$ exists and is consistent.

The rule `take-basic-ckpt(`$\mathcal{B}$`)` is the same as the one of the protocol $\mathcal{A}$. In addition to the previous control variables, each data manager DM_x has an additional variable V_x, which is incremented by Z each time a data checkpoint with sequence number aZ is taken. The rule `take-forced-ckpt(`$\mathcal{B}$`)` is the following:

`take-forced-ckpt(`$\mathcal{B}$`)` :

When DM_x **receives** *commit*$(MAX\_SN_{T_i})$ **from** TM_i:
- **if** $V_x < MAX\_SN_{T_i}$ **then**
  - (B1) $i_x \leftarrow i_x + 1$; $sn_x \leftarrow \lfloor MAX\_SN_{T_i}/Z \rfloor \times Z$;
  - (B2) Take a (forced) checkpoint $C_x^{i_x}$;
    $C_x^{i_x}.sn \leftarrow sn_x$;
  - (B3) Reset the local timer;
  - (B4) $V_x \leftarrow V_x + Z$.
- **endif**;
- (B5) Process the *commit* message.

7.5.4 Short Comparison with Previous Protocols

This section presents three checkpointing protocols proposed in the context of distributed databases [41, 43, 52]. Then the main differences among these protocols and the solutions proposed in this chapter are discussed.

The protocol in [52] determines a consistent global checkpoint by means of a two phase protocol using a checkpoint coordinator process that exchanges messages with its checkpoint subordinates processes one for each site. Each site maintains an independent local timestamp (like Lamport scalar clocks [33]) and a timestamp is associated with each transaction[8]. The first phase is used to agree on a common timestamp value among all sites. This value, say n, actually slits database's transactions into two groups the one that has a timestamp less or equal to n and the ones with timestamp greater than n. In the second phase, the checkpoint process in each site is delayed till all transactions whose timestamp is less than or equal to n are committed. Once the checkpoint process dumped the database state in a safe place, transactions whose timestamp is greater than n are executed. Note that during the first phase, the transactions are not stopped, however their updates are stored in a private area that can be read by the checkpointing process to execute a transaction-consistent dump. A similar approach using control messages to split transactions in two groups in order to get globally transaction consistent checkpoints has been proposed by Kim and Park in [30].

The protocol in [43] assumes each data object has a colour either black or white. Before the checkpointing process starts all data objects are white. The black colour indicates that the data object has been read by the checkpointing process. The checkpointing process continues till all data objects are black. Transactions takes a colour from the data objects they access. A transaction is white (resp. black) if all data object it accessed were white (resp. black). A transaction is grey if it accessed at least one black and one white data object. In a first version of the protocol, written by Pu ([42]), the protocol aborted each grey transaction in order to ensure serializability and to determine transaction-consistent global checkpoints. The protocol in [43] is a more refined version of [42], namely *save some*, which avoids to abort grey transactions by saving the *before values* of the data objects updated by each grey or white transaction in a private memory area accessible to the checkpointing process. This allows to execute a transaction consistent dump of the database. Compared to [52], the protocol in [43] splits transaction into two groups in a lazy way (by means of an "infection" from data objects) without exchanging control messages. However this approach increases the transaction response time and requires an unbounded memory capacity (the private memory required for saving before

[8] As it uses timestamps, this protocol is well suited to concurrency control based on timestamp.

values could be larger than the size of the database itself) as it is expected that grey transactions will be a wider majority of all transactions.

The protocol in [41] modifies [43] in order to bound the size of the required private memory. The checkpoint process is implemented as a set of read-only transactions one for each data object. Each data object has a colour white, grey or black. Initially each data object is white. Transactions can be black or white. Initially checkpointing transactions are black and normal transactions can be either black or white. A normal transaction turns black after either overwriting a gray data object or accessing a black data object. A data object changes from white to gray (resp. black) when a finally black transaction (*i.e.,* a transaction which is black at the commit time) reads (resp. overwrites) it. A data object changes from grey to black when written by any transaction. A consistent global checkpoint of the database is formed by the final non-black state of each data object. The introduction of the grey colour actually delays the time of reading of the data object by the checkpoint transaction, this reduces the size of the required private memory compared to [43]. On the other hand, transaction response time is increased by the previous delay and by the fact that the concurrency control has to manage the checkpointing transactions.

Compared to [52], the checkpointing protocols presented in this chapter employ a *lazy* coordination among data managers (neither control messages or a checkpoint coordinator is required). As opposed to [43] and [41], the proposed protocols do not overload the concurrency control with special purpose checkpointing transactions and do not need to manage colours. Moreover, they do not need private memory to be read by the checkpointing process to store partial transaction updates. On the negative side, the safe memory area where storing a copy of the checkpointed data objects can be larger than the size of the entire database (many checkpoints with distinct sequence numbers of a data object can be stored in the safe area at the same time). However, this size can be kept as small as possible by running frequently a garbage collection procedure.

Bibliography

[1] R. Baldoni, G. Cioffi, J.M. Hélary and M. Raynal, Direct Dependency-Based Determination of Consistent Global Checkpoints, Tech. Rep. 12-98, Dipartimento di Informatica e Sistemistica, Universita' di Roma "La Sapienza", 1998.

[2] R. Baldoni, J.M. Hélary and M. Raynal, Consistent Records in Asynchronous Computations, *Acta Informatica* 35:441-455, 1998.

[3] R. Baldoni, J.M. Hélary and M. Raynal, Rollback Dependency Trackability: Visible Characterizations, *Proc. ACM Symposium on the Principles on Distributed Computing*, 1999.

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

[5] R. Baldoni, J.M. Hélary, A. Mostefaoui and M. Raynal, Adaptive Checkpointing in Message Passing Distributed Systems, *International Journal of Systems Science*, 28(11):1145-1161, 1997.

[6] R. Baldoni, F. Quaglia and P. Fornara, An Index-Based Checkpointing Algorithm for Autonomous Distributed Systems, *Proc. IEEE Int. Symposium on Reliable Distributed Systems*, 1997, pp. 27-34 (an expanded version appeared on IEEE Transactions on Parallel and Distributed Systems, vol. 10, no.2, February 1999).

[7] R. Baldoni, F. Quaglia and B. Ciciani, A VP-Accordant Checkpointing Protocol Preventing Useless Checkpoints, *Proc. IEEE Int. Symposium on Reliable Distributed Systems*, 1998, pp. 61-67.

[8] G. Barigazzi and L. Strigini, Application-Transparent Setting of Recovery Points, *Proc. IEEE Fault Tolerant Computing Symposium*, 1983, pp. 48-55.

[9] B. Bhargava and S.R. Lian, Independent Checkpointing and Concurrent Rollback for Recovery, *Proc. IEEE Int. Symposium on Reliable Distributed Systems*, 1988, pp. 3-12.

[10] P.A. Bernstein, V. Hadzilacos and Goodman, Concurrency Control and Recovery in Database systems, *Addison Wesley Publishing Co.*, Reading, MA, 1987.

[11] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz and A. Silberschatz, On Rigorous Transaction Scheduling, *IEEE Transactions on Software Engineering*, 17(9):954-960, 1991.

[12] 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.

[13] J. Cao and K.C. Wang, An Abstract Model for Rollback Recovery Control in Distributed Systems, *ACM Operating Systems Review*, 1992, pp. 62-76.

[14] K.M. Chandy and L. Lamport, Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Transactions on Computer Systems*, 3(1):63-75, 1985.

[15] B. Ciciani and G. Cantone, An Approach to an Optimal Strategy of Recovery Point Insertion in Distributed Fault Tolerant Computing Systems, *Proc. 24th Allerton Conference on Communication, Control and Computing*, 1986, pp. 964-972.

[16] R. Cooper and K. Marzullo, Consistent Detection of Global Predicates, *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, 1991, pp. 163-173.

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

[18] C. Critchlow and K. Taylor, The Inhibition Spectrum and the Achievement of Causal Consistency, Tech. Rep. TR 90-1101, Cornell University, 1990.

[19] 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.

[20] E.N. Enolzahy and W. Zwaenepoel, Manetho: Transparent Rollback-Recovery with Low Overhead, *IEEE Transactions on Computers*, 41(5):526-531, 1992.

[21] C. Fidge, Logical Time in Distributed Computing Systems, *IEEE Computer*, pp. 28-33, August 1991.

[22] J. Fowler and W. Zwaenepoel, Causal Distributed Breakpoints, *Proc. IEEE Int. Conference on Distributed Computing Systems*, 1990, pp. 134-141.

[23] E. Fromentin, N. Plouzeau and M. Raynal, An Introduction to the Analysis and Debug of Distributed Computations, *Proc. IEEE International Conference on Algorithms and Architectures for Parallel Processing*, 1995, pp. 545-554.

[24] E. Fromentin and M. Raynal, Shared Global States in Distributed Computations, *Journal of Computer and System Sciences*, vol. 55, no. 3, 1997.

[25] K. Geihs and M. Seifert, Automated Validation of a Co-operation Protocol for Distributed Systems, *Proc. IEEE Int. Conference on Distributed Computing Systems*, 1986, pp. 436-443.

[26] J.N. Gray and A. Reuter, Transaction Processing: Concepts and Techniques, *Morgan Kaufmann*, 1070 pages, 1993.

[27] J.M. Hélary, A. Mostéfaoui and M. Raynal, Virtual Precedence in Asynchronous Distributed Systems: Concept and Applications. *Proc. 11th Int. Workshop on Distributed Algorithms*, Springer-Verlag LNCS 13220, 1997, pp. 170-184.

[28] J.M. Hélary, A. Mostéfaoui, R.H.B. Netzer and M. Raynal, Preventing Useless Checkpoints in Distributed Computations. *Proc. 16th IEEE Symposium on Reliable Distributed Systems*, 1997, pp. 183-190.

[29] D.B. Johnson and W. Zwaenepoel, Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing, *Journal of Algorithms*, 11(3):462-491, 1990.

[30] J.L. Kim and T. Park, An Efficient Recovery Scheme for Locking-Based Distributed Database Systems, *Proc. 13th IEEE Symposium on Reliable Distributed Systems*, 1997, pp. 183-190.

[31] R. Koo and S. Toueg, Checkpointing and Rollback-Recovery for Distributed Systems, *IEEE Transactions on Software Engineering*, 13(1):23-31, 1987.

[32] T.H. Lay and T.H. Yang, On Distributed Snapshots, *Information Processing Letters*, 25:153-158, 1987.

[33] L. Lamport, Time, Clocks and The Ordering of Events in a Distributed System, *Communications of the ACM*, 21(7):558-565, 1978.

[34] K. Marzullo and G. Neiger, Detection of Global State Predicates, *Proc. Int. Workshop on Distributed Algorithms*, 1991.

[35] D. Manivannan, R.H.B. Netzer and M. Singhal, Finding Consistent Global Checkpoints in a Distributed Computation, *IEEE Transactions on Parallel and Distributed Systems*, 8(6):623-627, 1997.

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

[37] D. Manivannan and M. Singhal, Quasi-Synchronous Checkpointing: Models, Characterization, and Classification, *TR No. OSU-CISRC-5/96-TR33, Dept. of Computer and Information Science*, The Ohio State University, 1996.

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

[39] B. Miller and J. Choi, Breakpoints and Halting in Distributed Programs, *Proc. IEEE International Conference on Distributed Computing Systems*, 1988, pp. 316-323.

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

[41] S. Pilarski and T. Kameda, Checkpointing for Distributed Databases: Starting from the Basics, *IEEE Transactions on Parallel and Distributed Systems*, 3(5):602-610, 1992.

[42] C. Pu, On-the-fly, Incremental, Consistent Reading of Entire Databases, *Algorithmica*, 1(3):271-287, 1986.

[43] C. Pu, H. Hong and J.M. Wha, Performance Evaluation of Global Reading of Entire Databases, *Proc. International Symposium on Databases in Parallel and Distributed Systems*, 1988, pp. 167-176.

[44] F. Quaglia, R. Baldoni and B. Ciciani, A Low-Overhead Z-Cycle-Free Checkpointing Algorithm for Distributed Systems, *Proc. European Research Seminar on Advances in Distributed Systems*, 1997, pp. 198-203.

[45] F. Quaglia, B. Ciciani and R. Baldoni, A Checkpointing-Recovery Scheme for Distributed Systems, *in Dimiter R. Avresky, David R. Kaeli, editors, "Fault Tolerant Parallel and Distributed Systems" (Chapter 5)*, Kluwer Academic Publishers, 1998.

[46] F. Quaglia, R. Baldoni and B. Ciciani, On the No-Z-Cycle Porperty in Distributed Executions, Tech. Rep. 01-99, Dipartimento di Informatica e Sistemistica, Universita' di Roma "La Sapienza", January 1999.

[47] F. Quaglia, B. Ciciani and R. Baldoni, Checkpointing Protocols in Distributed Systems with Mobile Hosts: a Performance Analysis, *Proc. 3rd Workshop on Fault Tolerant Parallel and Distributed Systems*, LNCS 1388, 1998, pp.742-755.

[48] P. Ramanathan and K.G. Shin, Use of Common Time Base for Checkpointing and Rollback Recovery in Distributed Systems, *IEEE Transactions on Software Engineering*, 19(6):571-583, 1993.

[49] B. Randell, System Structure for Software Fault Tolerance, *IEEE Transactions on Software Engineering*, SE1(2):220-232, 1975.

[50] D.L. Russell, State Restoration in Systems of Communicating Processes, *IEEE Transactions on Software Engineering*, SE6(2): 183-194, 1980.

[51] K. Salem and H. Garcia-Molina, Checkpointing Memory Resident Databases, *Tech. Rep. CS-TR-126-87*, Department of Computer Science, Princeton University, December 1987.

[52] S.H. Son and A.K. Agrawala, Distributed Checkpointing for Globally Consistent States of Databases, *IEEE Transactions on Software Engineering*, 15(10):1157-1166, 1989.

[53] R.E. Strom, S.A. Yemini and D.F. Bacon, A Recoverable Object Store, *Proc. Hawaii Int. Conference on Sistem Science*, 1998, pp. II-215-II-221.

[54] 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*, 1988, pp. 44-49.

[55] Y. Tamir and C.H Sequin, Error Recovery in Multicomputers Using Global Checkpoints, *Proc. Int. Conference on Parallel Processing*, 1984, pp. 32-41.

[56] Z. Tong, R.Y. Kain and T. Tsai, Rollback Recovery in Distributed Systems Using Loosely Synchronized Clocks, *IEEE Transactions on Parallel and Distributed Systems*, 3(2):246-251, 1992.

[57] K.Tusuoka, A. Kaneko and Y. Nishihara, Dynamic Recovery Schemes for Distributed Processes, *Proc. IEEE Int. Symposium on Reliability in Distributed Software and Databases*, 1981, pp.124-130.

[58] K. Vankatesh, T. Radakrishanan, and H.L. Li. Optimal Checkpointing and Local Recording for Domino-Free Rollback-Recovery, *Information Processing Letters*, 25:295-303, 1987.

[59] Y.M. Wang, A. Lowry and W.K. Fuchs, Consistent Global Checkpoints Based on Direct Dependency Traking, *Information Processing Letters*, 50(4): 223-230, 1994.

[60] Y.M. Wang and W.K. Fuchs, Optimistic Message Logging for Independent Checkpointing in Message Passing Systems, *in Proc. IEEE Int. Symposium on Reliable Distributed Systems*, pp. 147-154, 1992.

[61] Y.M. Wang and W.K. Fuchs, Lazy Checkpoint Coordination for Bounding Rollback Propagation, *in Proc. IEEE Int. Symposium on Reliable Distributed Systems*, pp. 78-85, 1993.

[62] Y.M. Wang, Space Reclamation for Uncordinated Checkpointing in Message-Passing Systems, PhD Thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana Champaign, 1993.

[63] Y.M. Wang, Maximum and Minimum Consistent Global Checkpoints and Their Applications, *Proc. IEEE Int. Symposium on Reliable Distributed Systems*, 1995, pp. 86-95.

[64] Y.M. Wang, Consistent Global Checkpoints That Contain a Given Set of Local Checkpoints. *IEEE Transactions on Computers*, 46(4):456-468, 1997.

[65] Y.M. Wang, Y. Huang, W.K. Fuchs, C. Kintala and G. Suri, Progressive Retry for Software Failure Recovery in Message-Passing Applications. *IEEE Transactions on Computers*, 46(10):1137-1141, 1997.

[66] Y.M. Wang, Y. Huang, K.P. Vo, P.Y. Chung and C. Kintala, Checkpointing and its Applications. *IEEE Int. Symposium on Fault Tolerant Computing*, 1995, pp. 22-31.

[67] J. Xu and R. Netzer, Adaptive Independent Checkpointing for Reducing Rollback Propagation, *Proc. IEEE Symposium on Parallel and Distributed Processing*, 1993, pp. 154-161.

Glossary

List of abbreviations

CAS	:	Checkpoint-After-Send
CASBR	:	Checkpoint-After-Send-Before-Receive
CBR	:	Checkpoint-Before-Receive
CZC	:	Core Z-Cycle
DP	:	Dependence Path
PESCM	:	Prime-Elementary-Simple-Causal-Message
EZC	:	Elementary Z-Cycle
FDAS	:	First Dependency-After-Send
FDI	:	Fixed Dependency Interval
MRS	:	Mark Receive Send
PWD	:	Piecewise-Deterministic
$\mathcal{NZC}$	:	No-Z-Cycle property
PZC	:	Prime Z-Cycle
$\mathcal{RDT}$	:	Rollback-Dependency-Trackability property
SCZC	:	Suspect Core Z-Cycle
SZpF	:	Strictly Z-path Free
$\mathcal{VP}$	:	Virtual Precedence property
ZpF	:	Z-path Free

Notations

m	:	message
$[m_1, \ldots, m_q]$	:	sequence of q messages constituting a Z-path
$send(m)$	:	send event of message m
$receive(m)$	:	receive event of message m
ζ	:	message chain
μ	:	causal message chain
$\lvert\zeta\rvert$	:	number of messages of the chain ζ

$S(\zeta)$	:	sequence of checkpoint intervals associated to ζ
P	:	set of all processes
P_i	:	process of identity i
$e_{i,x}$	:	x-th event of P_i
$C_{i,x}$	:	x-th checkpoint of P_i
$I_{i,x}$	:	x-th checkpoint interval of P_i
$\prec_P$	:	precedence on events in a process
$\prec_m$	:	precedence on events due to message exchange
$\xrightarrow{e}$	:	Happened-Before relation on events
$\xrightarrow{I}$	:	precedence relation on checkpoint intervals
$\circ$	:	causal concatenation
$\bullet$	:	non-causal concatenation
$\prec_{ckpt}$	:	precedence relation on checkpoints
$M(C_{i,x}, P_k)$	:	set of causal message chains from $C_{i,x}$ to P_k
$min(M(C_{i,x}, P_k))$	:	set of minimum elements in $M(C_{i,x}, P_k)$
sn_i	:	sequence number of P_i
en_i	:	equivalence number of process P_i
$\mathcal{H}$	:	set of all events
$\widehat{\mathcal{H}}$	:	partially ordered set $(\mathcal{H}, \xrightarrow{e})$
$\mathcal{C}_{\widehat{\mathcal{H}}}$	:	set of all local checkpoints
T	:	set of all transactions
T_i	:	transaction of identity i
$<_T$	:	precedence relation on transactions
σ_x^i	:	i-th version of data object x
$<_{LS}$	:	precedence on local states of data objects
C_x^i	:	i-th checkpoint of data object x
$\xrightarrow{DP}$	:	precedence relation due to a Dependence Path

Università *La Sapienza*
Dottorato di Ricerca in Ingegneria Informatica
Collana delle tesi
Collection of Theses

V-93-1 Marco Cadoli. *Two Methods for Tractable Reasoning in Artificial Intelligence: Language Restriction and Theory Approximation.* June 1993.

V-93-2 Fabrizio d'Amore. *Algorithms and Data Structures for Partitioning and Management of Sets of Hyperrectangles.* June 1993.

V-93-3 Miriam Di Ianni. *On the complexity of flow control problems in Store-and-Forward networks.* June 1993.

V-93-4 Carla Limongelli. *The Integration of Symbolic and Numeric Computation by p-adic Construction Methods.* June 1993.

V-93-5 Annalisa Massini. *High efficiency self-routing interconnection networks.* June 1993.

V-93-6 Paola Vocca. *Space-time trade-offs in directed graphs reachability problem.* June 1993.

VI-94-1 Roberto Baldoni. *Mutual Exclusion in Distributed Systems.* June 1994.

VI-94-2 Andrea Clementi. *On the Complexity of Cellular Automata.* June 1994.

VI-94-3 Paolo Giulio Franciosa. *Adaptive Spatial Data Handling.* June 1994.

VI-94-4 Andrea Schaerf. *Query Answering in Concept-Based Knowledge Representation Systems: Algorithms, Complexity, and Semantic Issues.* June 1994.

VI-94-5 Andrea Sterbini. *2-Thresholdness and its Implications: from the Synchronization with PVchunk to the Ibaraki-Peled Conjecture.* June 1994.

VII-95-1 Piera Barcaccia. *On the Complexity of Some Time Slot Assignment Problems in Switching Systems.* June 1995.

VII-95-2 Michele Boreale. *Process Algebraic Theories for Mobile Systems.* June 1995.

VII-95-3 Antonella Cresti. *Unconditionally Secure Key Distribution Protocols.*

June 1995.

VII-95-4 Vincenzo Ferrucci. *Dimension-Independent Solid Modeling.* June 1995.

VII-95-5 Esteban Feuerstein. *On-line Paging of Structured Data and Multi-threaded Paging.* June 1995.

VII-95-6 Michele Flammini. *Compact Routing Models: Some Complexity Results and Extensions.* June 1995.

VII-95-7 Giuseppe Liotta. *Computing Proximity Drawings of Graphs.* June 1995.

VIII-96-1 Luca Cabibbo. *Querying and Updating Complex-Object Databases.* May 1996.

VIII-96-2 Diego Calvanese. *Unrestricted and Finite Model Reasoning in Class-Based Representation Formalisms.* May 1996.

VIII-96-3 Marco Cesati. *Structural Aspects of Parameterized Complexity.* May 1996.

VIII-96-4 Flavio Corradini. *Space, Time and Nondeterminism in Process Algebras.* May 1996.

VIII-96-5 Stefano Leonardi. *On-line Resource Management with Application to Routing and Scheduling.* May 1996.

VIII-96-6 Rosario Pugliese. *Semantic Theories for Asynchronous Languages.* May 1996.

IX-97-1 Paola Alimonti. *Local search and approximability of MAX SNP problems.* May 1997.

IX-97-2 Tiziana Calamoneri. *Does Cubicity Help to Solve Problems?.* May 1997.

IX-97-3 Paolo Di Blasio. *A Calculus for Concurrent Objects: Design and Control Flow Analysis.* May 1997.

IX-97-4 Bruno Errico. *Intelligent Agents and User Modelling.* May 1997.

IX-97-5 Roberta Mancini. *Modelling Interactive Computing by exploiting the Undo.* May 1997.

IX-97-6 Riccardo Rosati. *Autoepistemic Description Logics.* May 1997.

IX-97-7 Luca Trevisan. *Reductions and (Non-)Approximability.* May 1997.

X-98-1 Gianluca Battaglini. *Analysis of Manufacturing Yield Evaluation of VLSI/WSI Systems: Methods and Methodologies.* April 1998.

X-98-2 Piergiorgio Bertoli. *Using OMRS in practice: a case study with Acl-2.* April 1998.

X-98-3 Chiara Ghidini. *A semantics for contextual reasoning: theory and two relevant applications.* April 1998.

X-98-4 Roberto Giaccio. *Visiting complex structures.* April 1998.

X-98-5 Giampaolo Greco. *Dimension and structure in Combinatorics.* April 1998.

X-98-6 Paolo Liberatore. *Compilation of intractable problems and its application to artificial intelligence.* April 1998.

X-98-7 Fabio Massacci. *Efficient approximate tableaux and an application to computer security.* April 1998.

X-98-8 Chiara Petrioli. *Energy-Conserving Protocols for Wireless Communications.* April 1998.

X-98-9 Giulio Balestreri. *Algebraic Semantics of Shared Spaces Coordination Languages.* April 1999.

XI-99-1 Luca Becchetti. *Efficient Resource Management in High Bandwidth Networks.* April 1999.

XI-99-2 Nicola Cancedda. *Text Generation from Message Understanding Conference Templates.* April 1999.

XI-99-3 Luca Iocchi. *Design and Development of Cognitive Robots.* April 1999.

XI-99-4 Francesco Quaglia. *Consistent checkpointing in distributed computations: theoretical results and protocols.* April 1999.

XI-99-5 Milton Romero. *Disparity/Motion Estimation For Stereoscopic Video Processing.* April 1999.