Progressive Construction of Consistent Global Checkpoints

Islene Calciolari Garcia
Luiz Eduardo Buzato

Relatório Técnico IC–98-36

Outubro de 1998

Progressive Construction of Consistent Global Checkpoints[*]

Islene Calciolari Garcia
Luiz Eduardo Buzato

Abstract

A checkpoint pattern is an abstraction of the computation performed by a distributed application. A *progressive view* of this abstraction is formed by a sequence of consistent global checkpoints that may have occurred in this order during the execution of the application. Considering pairs of checkpoints, we have determined that a checkpoint must be observed before another in a progressive view if the former *Z-precedes* the latter. Based on the Z-precedence and characteristics of the checkpoint pattern, we propose original algorithms for the progressive construction of consistent global checkpoints. We demonstrate that the Z-precedence between a pair of checkpoints is a much simpler way to express the existence of a zigzag path connecting them, and we discuss other advantages of our relation.

Keywords: distributed checkpointing, consistent global states, causality, zigzag paths, monitoring systems.

1 Introduction

Checkpoints are part of the solution for a wide range of problems that arise in distributed applications, including fault-tolerant computing, debugging, monitoring and reconfiguration. A process of a distributed application is supposed to cooperate with the monitoring system by selecting distinguished states of its execution—called *checkpoints*. The set of all checkpoints taken by the processes of a distributed application form a *checkpoint pattern* that is an abstraction of the computation performed by the application.

A global checkpoint is a set of checkpoints, one per process. A global checkpoint is *consistent* if it could have been observed by an idealized external monitor [1]. The evaluation of global predicates [2]—the core of many monitors—is only meaningful when verified against consistent global checkpoints. Moreover, a *progressive view* of the computation may be required, in the sense that a monitor should construct a sequence of consistent global checkpoints that may have occurred in this order during the computation.

[*]Institute of Computing, UNICAMP, Caixa Postal 6176, 13083-970 Campinas, SP. This work has been supported by FAPESP under grant no. 95/1983-8 for Islene Calciolari Garcia and grant no. 96/1532-9 for the Laboratory of Distributed Systems.

A checkpoint pattern may contain *useless* checkpoints, that is, checkpoints that cannot be part of any consistent global checkpoint [12]. Quasi-synchronous checkpointing protocols [10] allow processes to take checkpoints spontaneously, but sometimes they are forced by the protocol to take additional checkpoints in order to reduce or eliminate the presence of useless checkpoints. The possibility of rollback recovery has been the motivation for the development of most of the quasi-synchronous checkpointing protocols [3]. Given a failure, a procedure is triggered to rollback the application from its last global state to a consistent global checkpoint. However, for some classes of checkpoint patterns, the efficient construction of consistent global checkpoints is still an open problem [10]. Thus, we can identify two *orthogonal* problems: (i) the adequate selection of checkpoints and (ii) the construction of consistent global checkpoints.

Our approach is to use quasi-synchronous checkpointing protocols to build a progressive view of a computation. In order to attain our goal, we have determined the Z-precedence between checkpoints: a checkpoint a must be observed before a checkpoint b in a progressive view if a Z-precedes b. This relationship is a generalization of the Lamport's causal precedence [8] and equivalent to the zigzag paths proposed by Netzer and Xu [12]. Based on the Z-precedence between checkpoints, we propose original algorithms for the progressive construction of consistent global checkpoints. Because its very simple and has an intuitive meaning, we believe that the Z-precedence can provide a better understanding of consistent global checkpoints.

The paper is structured as follows. Section 2 describes the computational model adopted. Section 3 explores the progressive view of a computation, introducing the Z-precedence. Section 4 describes algorithms to build consistent global checkpoints progressively. In Section 5, we compare the Z-precedence with zigzag paths [12]. Finally, Section 6 concludes the paper.

2 Model

A distributed application is composed of n sequential processes $(p_0, \ldots, p_{n-1})$ that communicate only by exchanging messages. Messages cannot be corrupted, but can be delivered out of order or even lost. The activity of a process is modeled as a sequence of *events* that can be divided into internal events and communication events, realized through the sending and the reception of messages. Figure 1 illustrates a space-time diagram [8] augmented with checkpoints (black squares). Checkpoints are internal events. We assume that each process takes a checkpoint immediately after execution begins (initial checkpoint) and immediately before execution ends (final checkpoint).

Let σ_i^ι denote the state of p_i immediately after the execution of its ι-th event and let $\hat{\sigma}_i^\gamma$ denote the γ-th checkpoint taken by p_i, correspondent to a state σ_i^ι with $\gamma \leq \iota$. Considering Lamport's causal precedence [8] between events, let $\hat{\sigma}_a^\alpha \rightarrow \hat{\sigma}_b^\beta$ indicate that the event that generated $\hat{\sigma}_a^\alpha$ has causally preceded the event that generated $\hat{\sigma}_b^\beta$. Let $\hat{\sigma}_a^\alpha \| \hat{\sigma}_b^\beta$ indicate checkpoints taken by causally unrelated events (concurrent events). In Figure 1, we can see that $\hat{\sigma}_0^1 \rightarrow \hat{\sigma}_1^2$ and $\hat{\sigma}_0^1 \| \hat{\sigma}_2^2$.

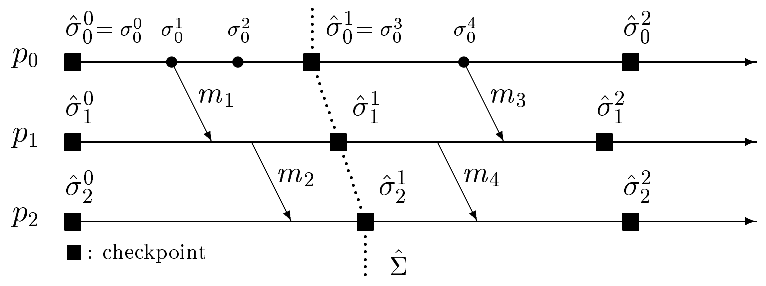


Figure 1: A distributed computation with checkpoints

3 Progressive View of a Distributed Computation

A consistent global checkpoint is a set of checkpoints, one per process, that could have been observed by an idealized external monitor. Therefore, a consistent global checkpoint must contain only causally unrelated checkpoints [12].

Definition 3.1 Consistent Global Checkpoint—*A global checkpoint* $\hat\Sigma = (\hat\sigma_0^{\iota_0}, \ldots, \hat\sigma_{n-1}^{\iota_{n-1}})$ *is consistent if, and only if,*

$$\forall i, j : 0 \leq i, j < n : (\hat\sigma_i^{\iota_i} \not\rightarrow \hat\sigma_j^{\iota_j})$$

A progressive view of a distributed computation is a sequence of consistent global checkpoints such that each global checkpoint in the sequence appears to have happened after the other. Obviously, a distributed computation may have many progressive views.

Definition 3.2 Progressive View—*A progressive view of a computation is a sequence of consistent global checkpoints* $\{\hat\Sigma^0, \hat\Sigma^1, \ldots, \hat\Sigma^m\}$ *such that*

$$\forall k : 0 \leq k < m : (\hat\sigma \in \hat\Sigma^k) \wedge (\hat\sigma' \in \hat\Sigma^{k+1}) \Rightarrow (\hat\sigma' \not\rightarrow \hat\sigma)$$

The global checkpoint $\hat\Sigma$ is an example of a consistent global checkpoint and the sequence $\{(\hat\sigma_0^0, \hat\sigma_1^0, \hat\sigma_2^0), (\hat\sigma_0^1, \hat\sigma_1^1, \hat\sigma_2^1), (\hat\sigma_0^2, \hat\sigma_1^1, \hat\sigma_2^1), (\hat\sigma_0^2, \hat\sigma_1^2, \hat\sigma_2^2)\}$ is a progressive view of the computation depicted in Figure 1.

3.1 Z-Precedence Between Checkpoints

A precedence relation between a pair of checkpoints, say $\hat\sigma_i^{\iota_i} \rightarrow \hat\sigma_j^{\iota_j}$, implies that they cannot be part of the same consistent global checkpoint. For example, consider checkpoints $\hat\sigma_0^1$ and $\hat\sigma_1^2$ (Figure 1). Message m_3 has been sent after $\hat\sigma_0^1$ and it has been received before $\hat\sigma_1^2$. Clearly, $\hat\sigma_1^2$ cannot be part of the same consistent global checkpoint with any checkpoint $\hat\sigma_0^\alpha$ such that $\alpha \leq 1$. Consequently, checkpoint $\hat\sigma_0^1$ must be *observed before* checkpoint $\hat\sigma_1^2$ in a progressive view.

Definition 3.3 *A checkpoint* $\hat\sigma_i^{\iota_i}$ *must be* observed before *a checkpoint* $\hat\sigma_j^{\iota_j}$ *if, and only if,* $\hat\sigma_j^{\iota_j}$ *cannot be part of the same consistent global checkpoint with* $\hat\sigma_i^{\iota_i'}$ *such that* $\iota_i' \leq \iota_i$.

Concurrent checkpoints may also have a well-defined order in a progressive view. Consider the concurrent checkpoints $\hat{\sigma}_0^1$ and $\hat{\sigma}_2^2$ in Figure 1. Due to m_4, $\hat{\sigma}_1^1$ must be observed before $\hat{\sigma}_2^2$. Thus, we must consider checkpoint $\hat{\sigma}_1^2$ and conclude that due to m_3, $\hat{\sigma}_0^1$ must be observed before $\hat{\sigma}_1^2$. Consequently, checkpoint $\hat{\sigma}_0^1$ must be observed before $\hat{\sigma}_2^2$. Extending this scenario, we introduce the Z-precedence between checkpoints. This relation indicates when a checkpoint must be observed before another in a progressive view.

Definition 3.4 Z-precedence between checkpoints
Checkpoint $\hat{\sigma}_a^\alpha$ Z-precedes *checkpoint* $\hat{\sigma}_b^\beta$ *(*$\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_b^\beta$*) if, and only if,*

- $\hat{\sigma}_a^\alpha \to \hat{\sigma}_b^\beta$, *or*
- $\exists \hat{\sigma}_c^\gamma : (\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_c^\gamma) \wedge (\hat{\sigma}_c^{\gamma-1} \rightsquigarrow \hat{\sigma}_b^\beta)$.

Theorem 3.1 *If* $\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_b^\beta$*,* $\hat{\sigma}_a^\alpha$ *must be observed before* $\hat{\sigma}_b^\beta$*.*

Proof: The Z-precedence $\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_b^\beta$ can be expressed as a sequence of p causal precedence relations between pairs of checkpoints:

$$\hat{\sigma}_a^\alpha = \hat{\sigma}_{i_0}^{\gamma_0-1} \quad (\hat{\sigma}_{i_0}^{\gamma_0-1} \to \hat{\sigma}_{i_1}^{\gamma_1}), (\hat{\sigma}_{i_1}^{\gamma_1-1} \to \hat{\sigma}_{i_2}^{\gamma_2}), \ldots, (\hat{\sigma}_{i_{p-1}}^{\gamma_{p-1}-1} \to \hat{\sigma}_{i_p}^{\gamma_p}) \quad \hat{\sigma}_{i_p}^{\gamma_p} = \hat{\sigma}_b^\beta$$

By induction on p, we prove that $\hat{\sigma}_a^\alpha$ must be observed before $\hat{\sigma}_b^\beta$.

Base: ($p = 1$) The causal precedence $\hat{\sigma}_a^\alpha \to \hat{\sigma}_b^\beta$ implies that $\hat{\sigma}_a^\alpha$ must be observed before $\hat{\sigma}_b^\beta$. That is, in order to consider $\hat{\sigma}_b^\beta$ we must consider $\hat{\sigma}_a^{\alpha+1}$.

Step: ($p > 1$) Assume that $\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_{i_p}^{\gamma_p}$ implies that $\hat{\sigma}_a^\alpha$ must be observed before $\hat{\sigma}_{i_p}^{\gamma_p}$. Also, assume that $\hat{\sigma}_{i_p}^{\gamma_p-1} \to \hat{\sigma}_b^\beta$ implies that $\hat{\sigma}_{i_p}^{\gamma_p-1}$ must be observed before $\hat{\sigma}_b^\beta$. In order to consider $\hat{\sigma}_b^\beta$, we must consider $\hat{\sigma}_{i_p}^{\gamma_p}$ and also $\hat{\sigma}_a^{\alpha+1}$ (Figure 3). Consequently, $\hat{\sigma}_a^\alpha$ must be observed before $\hat{\sigma}_b^\beta$. □

Definition 3.5 Z-cycle—*A checkpoint* $\hat{\sigma}$ *participates in a Z-cycle if, and only if,* $\hat{\sigma} \rightsquigarrow \hat{\sigma}$*.*

A checkpoint may have a Z-precedence to itself, called a Z-cycle. Using Theorem 3.1, we could conclude that such checkpoint must be observed before itself, what does not make sense. The conclusion to be drawn is that this checkpoint is *useless*: it cannot be part of any consistent global checkpoint, allowing us to formulate the following corollary.

Corollary 3.1 *A checkpoint* $\hat{\sigma}$ *that participates in a Z-cycle is an useless checkpoint.*

For example, checkpoints $\hat{\sigma}_2^1$ and $\hat{\sigma}_1^2$ in Figure 2 participate in Z-cycles. Therefore, they are useless.

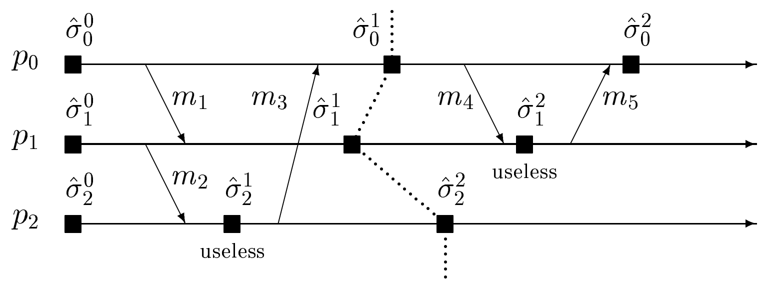
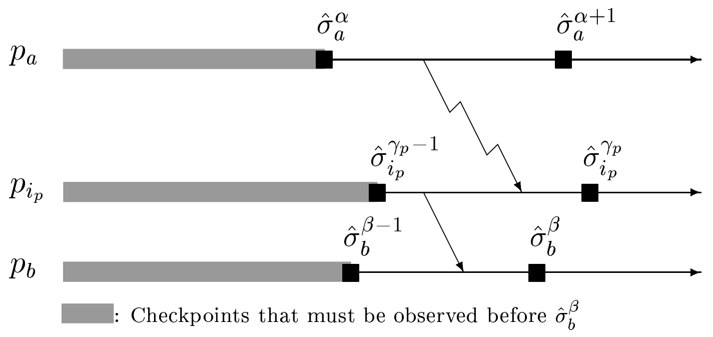


Figure 2: Checkpoint pattern with useless checkpoints



p_a
p_{i_p}
p_b
$\hat{\sigma}_a^\alpha$
$\hat{\sigma}_a^{\alpha+1}$
$\hat{\sigma}_{i_p}^{\gamma_p-1}$
$\hat{\sigma}_{i_p}^{\gamma_p}$
$\hat{\sigma}_b^{\beta-1}$
$\hat{\sigma}_b^{\beta}$
▨: Checkpoints that must be observed before $\hat{\sigma}_b^\beta$

Figure 3: Induction used in the proof of Theorem 3.1

3.2 A Step in a Progressive View

Assume that $\hat{\Sigma} = \{\hat{\sigma}_0^{\iota_0}, \ldots, \hat{\sigma}_{n-1}^{\iota_{n-1}}\}$ is a consistent global checkpoint of the monitor's progressive view. Let $\hat{\sigma}_i^{\iota_i+1}$ be the immediate successor of $\hat{\sigma}_i^{\iota_i}$, if $\hat{\sigma}_i^{\iota_i}$ is not a final checkpoint, and let $S = \{\hat{\sigma}_0^{\iota_0+1}, \ldots, \hat{\sigma}_{n-1}^{\iota_{n-1}+1}\}$ be the set of immediate successors of the checkpoints in $\hat{\Sigma}$. The behavior of a monitor that constructs a progressive view of a distributed application is simple, it selects $\mathit{Step}(\hat{\Sigma}) \subseteq S$ to build $\mathit{Next}(\hat{\Sigma})$, the successor of $\hat{\Sigma}$ in its progressive view.

Theorem 3.2 *Let $\hat{\Sigma} = \{\hat{\sigma}_0^{\iota_0}, \ldots, \hat{\sigma}_{n-1}^{\iota_{n-1}}\}$ be a consistent global checkpoint and let its succeeding checkpoints form the set $S = \{\hat{\sigma}_0^{\iota_0+1}, \ldots, \hat{\sigma}_{n-1}^{\iota_{n-1}+1}\}$. We define*

$$\mathit{Step}(\hat{\Sigma}) = \{\hat{\sigma}_i^{\iota_i+1} \in S :: \nexists \hat{\sigma}_j^{\iota_j+1} \in S : \hat{\sigma}_j^{\iota_j+1} \rightsquigarrow \hat{\sigma}_i^{\iota_i+1}\} \qquad (*)$$

$\mathit{Next}(\hat{\Sigma})$, formed by the replacement of $\mathit{Step}(\hat{\Sigma})$ in $\hat{\Sigma}$, is a consistent global checkpoint.

Proof: Assume that $\mathit{Next}(\hat{\Sigma})$ is inconsistent. Thus, there must exist a pair of checkpoints, say $\hat{\sigma}_a^\alpha$ and $\hat{\sigma}_b^\beta$, in $\mathit{Next}(\hat{\Sigma})$ such that $\hat{\sigma}_a^\alpha \to \hat{\sigma}_b^\beta$. There are four possibilities of membership for $\hat{\sigma}_a^\alpha$ and $\hat{\sigma}_b^\beta$:

- $\hat{\sigma}_a^\alpha \in \hat{\Sigma}$, $\hat{\sigma}_b^\beta \in \hat{\Sigma}$—Violates the hypothesis that $\hat{\Sigma}$ is a consistent global checkpoint.
- $\hat{\sigma}_a^\alpha \in \hat{\Sigma}$, $\hat{\sigma}_b^\beta \in \mathit{Step}(\hat{\Sigma})$—Since $\hat{\sigma}_a^{\alpha+1}$ does not belong to $\mathit{Step}(\hat{\Sigma})$, there must exist a checkpoint $\hat{\sigma}_i^{\iota_i+1}$ in S such that $\hat{\sigma}_i^{\iota_i+1} \rightsquigarrow \hat{\sigma}_a^{\alpha+1}$. The concatenation of $\hat{\sigma}_a^\alpha \to \hat{\sigma}_b^\beta$ and

$\hat{\sigma}_i^{\iota_i+1} \rightsquigarrow \hat{\sigma}_a^{\alpha+1}$ forms a Z-precedence $\hat{\sigma}_i^{\iota_i+1} \rightsquigarrow \hat{\sigma}_b^{\beta}$ that violates the rule $(*)$ used to build $Step(\hat{\Sigma})$.

- $\hat{\sigma}_a^{\alpha} \in Step(\hat{\Sigma})$, $\hat{\sigma}_b^{\beta} \in \hat{\Sigma}$—Violates the hypothesis that $\hat{\Sigma}$ is a consistent global checkpoint.
- $\hat{\sigma}_a^{\alpha} \in Step(\hat{\Sigma})$, $\hat{\sigma}_b^{\beta} \in Step(\hat{\Sigma})$—Violates the rule $(*)$ used to form $Step(\hat{\Sigma})$. $\square$

The monitor can build a progressive view, using Theorem 3.2, starting from any consistent global checkpoint, for example, the consistent global checkpoint formed by the initial checkpoints of the application. An additional requirement may be that all useful checkpoints must have to be considered and all useless ones must have to be discarded. In the next Section, we will present algorithms to calculate $Step(\hat{\Sigma})$.

4 Algorithms

In this Section, we introduce algorithms to build consistent global checkpoints progressively. The details of each algorithm are dictated by the checkpoint pattern of the underlying computation. Following this, we introduce a classification of checkpoint patterns that is based on the one proposed by Manivannan and Singhal [10].

- **Z-Precedence Free (ZPF) Pattern**: For every pair of checkpoints in this pattern, say $\hat{\sigma}_a^{\alpha}$ and $\hat{\sigma}_b^{\beta}$, the following condition holds: $(\hat{\sigma}_a^{\alpha} \prec \hat{\sigma}_b^{\beta}) \iff (\hat{\sigma}_a^{\alpha} \rightsquigarrow \hat{\sigma}_b^{\beta})$. In other words, all Z-precedences are causal precedences.
- **Z-Cycle Free (ZCF) Pattern**: In this pattern, checkpoints do not participate in Z-cycles; it contains only useful checkpoints.
- **Partially Z-Cycle Free (PZCF) Pattern**: In this pattern, checkpoints may participate in Z-cycles; it may contain useless checkpoints.

The algorithms are presented in Java[1], because it is the language we adopted for our implementation of the monitor. Additionally, Java [7, 13] is easy to read and has a precise description.

Processes behavior: We assume that processes maintain and propagate vector clocks [11], as described by class `Process` (Class 4.1). Vectors clocks are used to characterize casual precedence among checkpoints, and they are initialized to guarantee that the set of initial checkpoints form a consistent global checkpoint. When a message is sent, the vector clock of the sender is piggybacked onto it. Before consuming a message, each process takes a component-wise maximum of its vector clock and the received vector clock. When a process takes a checkpoint, it increments its corresponding entry in the vector clock. A checkpoint is described by class `VC_Ckpt` (Class 4.2).

[1]Java is a trademark of Sun Microsystems, Inc.

Class 4.1 Process.java

```
public class Process {

    public static final int N = 100;  // Number of processes in the application

    public int pid;  // An unique identifier in the range 0..N-1
    public int[ ] VC = new int[N];  // Process' vector clock

    public class Message {
        public int[ ] VC;  // Message's vector clock
        // Message body
    }

    public void takeCheckpoint() {
        VC[pid]++;
        // Take checkpoint
    }

    public Process(int pid) {  // Constructor
        this.pid = pid;
        for (int i=0; i < N; i++) VC[i] = -1;  // Vector clock initialization
        takeCheckpoint();  // VC[pid] is set to 0
    }

    public void finalize() { takeCheckpoint(); }   // Destructor-like method

    public void sendMessage(Message m) {
        m.VC = (int[ ]) VC.clone();  // Copies the whole array
        // Send message
    }

    public void receiveMessage(Message m) {
        for (int i=0; i < N; i++)  // Component-wise maximum
            if (m.VC[i] > VC[i]) VC[i] = m.VC[i];
        // Receive message
    }
}
```

Class 4.2 VC_Ckpt.java

```
public class VC_Ckpt {
    public int v[ ];
    public int pid;
    // Process' checkpoint

    boolean precedes(VC_Ckpt ckpt) {  // Returns true if this object causally precedes ckpt
        return (v[pid] < ckpt.v[pid]) || ((v[pid] == ckpt.v[pid]) && (pid ≠ ckpt.pid));
    }
}
```

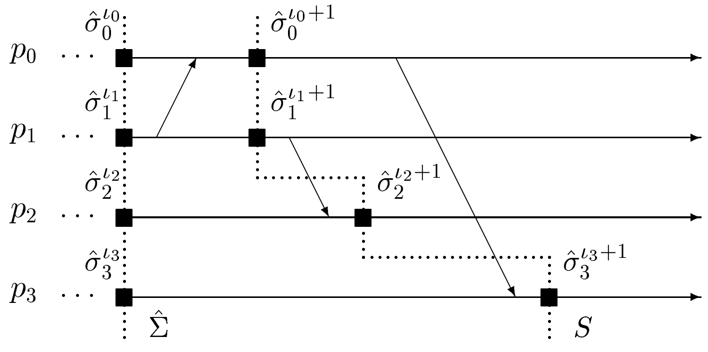
Class 4.3 ZPF_Pattern.java

```
public class ZPF_Pattern {

    VC_Ckpt[ ] C;  // Consistent global checkpoint
    VC_Ckpt[ ] S;  // Succeeding checkpoints of C

    private boolean[ ] M = new boolean[Process.N];

    public void next() {

        for (int i=0; i < Process.N; i++) M[i] = false;

        for (int i=0; i < Process.N; i++)
            for (int j = 0; !M[i] && j < Process.N; j++)
                if (S[j].precedes(S[i]))
                    M[i] = true;

        for (int i=0; i < Process.N; i++)
            if (!M[i]) { C[i] = S[i]; S[i] = null; }
    }
}
```

General structure of the algorithms: For each pattern, we define a method called `next`, whose function is to allow the monitor to move forward in its progressive view of the application. The monitor maintains variables `C` and `S` to implement the sets $\hat{\Sigma}$ and S, respectively, as in Theorem 3.2. It also maintains an auxiliary vector `M` to mark processes whose checkpoints in `S` are Z-preceded by other checkpoints in `S`. The checkpoints in `S` taken by the unmarked processes can be substituted in `C`. Useless checkpoints should be discarded. For simplicity, we assume that when `next` is called, `S` has no checkpoint missing.

Progressive view in a ZPF pattern: In this pattern, $\mathit{Step}(\hat{\Sigma})$ can be simply determined by checkpoints in S that are not causally preceded by other checkpoints in S. Class `ZPF_Pattern` (Class 4.3) describes an implementation of `next` for this pattern. Figure 4 illustrates a result of its execution: $\hat{\sigma}_2^{\iota_2+1}$ has been marked by the algorithm because its causally preceded by $\hat{\sigma}_1^{\iota_1+1}$ and $\hat{\sigma}_3^{\iota_3+1}$ has been marked because its causally preceded by $\hat{\sigma}_0^{\iota_0+1}$. Thus, $\mathit{Step}(\hat{\Sigma})$ is formed by $\{\hat{\sigma}_0^{\iota_0+1}, \hat{\sigma}_1^{\iota_1+1}\}$ and $\mathit{Next}(\hat{\Sigma}) = \{\hat{\sigma}_0^{\iota_0+1}, \hat{\sigma}_1^{\iota_1+1}, \hat{\sigma}_2^{\iota_2}, \hat{\sigma}_3^{\iota_3}\}$.

Progressive view in a ZCF pattern: Initially, processes whose checkpoints in `S` are causally preceded by other checkpoints in `S` are marked. Furthermore, the algorithm recursively marks processes whose checkpoints in `S` are preceded by checkpoints in `C` taken by marked processes. For example, if p_i is marked and there is a checkpoint $\hat{\sigma}_j^{\iota_j+1}$ such that $\hat{\sigma}_i^{\iota_i} \rightarrow \hat{\sigma}_j^{\iota_j+1}$, p_j must also be marked. $\mathit{Next}(\hat{\Sigma})$ is a consistent global checkpoint; $\mathit{Step}(\hat{\Sigma})$ and $\hat{\Sigma}$ contain only concurrent checkpoints. Assume a causal precedence from a checkpoint $\hat{\sigma}_i^{\iota_i}$ in $\hat{\Sigma} - \mathit{Step}(\hat{\Sigma})$ to a checkpoint $\hat{\sigma}_j^{\iota_j+1}$ in $\mathit{Step}(\hat{\Sigma})$. Since p_i is a marked process, p_j should also have been marked and $\hat{\sigma}_j^{\iota_j+1}$ could not belong to $\mathit{Step}(\hat{\Sigma})$.

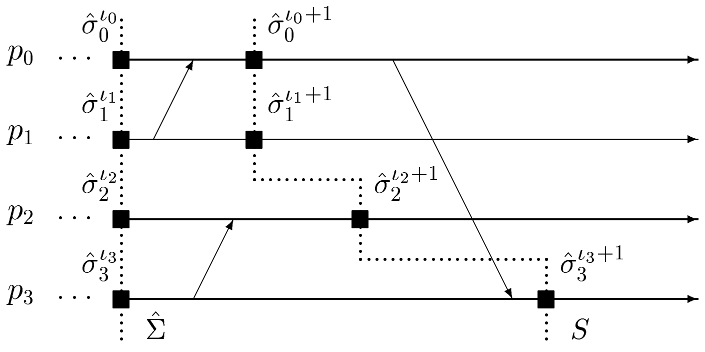

$\hat{\sigma}_0^{\iota_0}$
$\hat{\sigma}_0^{\iota_0+1}$
p_0 ...
$\hat{\sigma}_1^{\iota_1}$
$\hat{\sigma}_1^{\iota_1+1}$
p_1 ...
$\hat{\sigma}_2^{\iota_2}$
$\hat{\sigma}_2^{\iota_2+1}$
p_2 ...
$\hat{\sigma}_3^{\iota_3}$
$\hat{\sigma}_3^{\iota_3+1}$
p_3 ...
$\hat{\Sigma}$
S

$$Step(\hat{\Sigma}) = \{\hat{\sigma}_0^{\iota_0+1}, \hat{\sigma}_1^{\iota_1+1}\}$$

Figure 4: Progressive view in a ZPF pattern

Class 4.4 ZCF_Pattern.java

```
public class ZCF_Pattern {

    VC_Ckpt[ ] C;  // Consistent global checkpoint
    VC_Ckpt[ ] S;  // Succeeding checkpoints of C

    private boolean[ ] M = new boolean[Process.N];

    public void next() {

        for (int i=0; i < Process.N; i++) M[i] = false;

        for (int i=0; i < Process.N; i++)
            for (int j=0; !M[i] && j < Process.N; j++)
                if (S[j].precedes(S[i]))
                    mark(i);

        for (int i=0; i < Process.N; i++)
            if (!M[i]) { C[i] = S[i]; S[i] = null; }
    }

    protected void mark(int i) {
        if (!M[i]) {
            M[i] = true;
            for (int k=0; k < Process.N; k++)
                if (C[i].precedes(S[k])) mark(k);
        }
    }
}
```


$\hat{\sigma}_0^{\iota_0}$
$\hat{\sigma}_0^{\iota_0+1}$
$\hat{\sigma}_1^{\iota_1}$
$\hat{\sigma}_1^{\iota_1+1}$
$\hat{\sigma}_2^{\iota_2}$
$\hat{\sigma}_2^{\iota_2+1}$
$\hat{\sigma}_3^{\iota_3}$
$\hat{\sigma}_3^{\iota_3+1}$
p_0
p_1
p_2
p_3
$\hat{\Sigma}$
S

$$Step(\hat{\Sigma}) = \{\hat{\sigma}_0^{\iota_0+1}, \hat{\sigma}_1^{\iota_1+1}\}$$

Figure 5: Progressive view in a ZCF pattern

Since this pattern does not admit Z-cycles, $Step(\hat{\Sigma})$ is guaranteed to be non-empty. Assume that the pattern is ZCF but there exists a non-final consistent global checkpoint $\hat{\Sigma} = \{\hat{\sigma}_0^{\iota_0}, \ldots, \hat{\sigma}_{n-1}^{\iota_{n-1}}\}$ such that $Step(\hat{\Sigma})$ is empty. Then, for all checkpoints $\hat{\sigma}_i^{\iota_i+1}$ in $S = \{\hat{\sigma}_0^{\iota_0+1}, \ldots, \hat{\sigma}_{n-1}^{\iota_{n-1}+1}\}$ we can choose a checkpoint $\hat{\sigma}_j^{\iota_j+1}$ such that $\hat{\sigma}_j^{\iota_j+1} \rightsquigarrow \hat{\sigma}_i^{\iota_i+1}$. Since the number of checkpoints in S is finite, we have a Z-cycle. This result is important not only because it guarantees that ZCF protocols originally developed for backward error recovery can also be used for monitoring. Additionally, it is possible to explore these protocols in the context of fault-tolerant systems where the seamless integration of error recovery and monitoring can be seen as a basic requirement.

Class `ZCF_Pattern` (Class 4.4) describes an implementation of method `next` for this pattern. In Figure 5, consider the Z-precedences within S: (i) $\hat{\sigma}_0^{\iota_0+1} \rightsquigarrow \hat{\sigma}_3^{\iota_3+1}$, (ii) $\hat{\sigma}_0^{\iota_0+1} \rightsquigarrow \hat{\sigma}_2^{\iota_2+1}$. At this level of abstraction, (i) and (ii) are sufficient to compute $Step(\hat{\Sigma})$ and $Next(\hat{\Sigma})$. In contrast, the sequence of causal precedences $\hat{\sigma}_0^{\iota_0+1} \rightarrow \hat{\sigma}_3^{\iota_3+1}$ and $\hat{\sigma}_3^{\iota_3} \rightarrow \hat{\sigma}_2^{\iota_2+1}$ must be considered to determine that $\hat{\sigma}_2^{\iota_2+1}$ cannot be part of $Step(\hat{\Sigma})$.

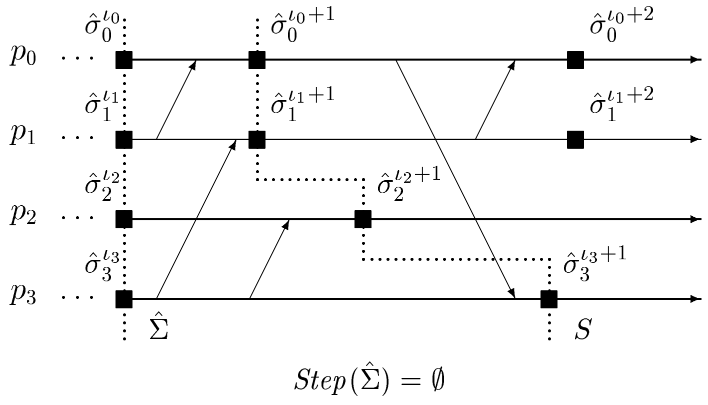
Progressive view in a PZCF pattern: In this pattern, it is necessary to know if a checkpoint Z-precedes itself: this information is necessary to determine useless checkpoints. An auxiliary matrix, called `Z`, is used. An entry `Z[i,j]` indicates whether $\hat{\sigma}_i^{\iota_i+1} \rightsquigarrow \hat{\sigma}_j^{\iota_j+1}$. If `Z[i,i]` is true, $\hat{\sigma}_i^{\iota_i+1}$ is an useless checkpoint. Class `PZCF_Pattern` describes an implementation of method `next`, that is similar to the implementation `next` for the `ZCF_Pattern`, it only adds the computation related to the `Z` matrix. This algorithm is guaranteed to progress, because if $Step(\hat{\Sigma})$ is empty, at least one useless checkpoint will be identified. Figure 6 illustrates a result of its execution in which $Step(\hat{\Sigma})$ is empty and checkpoints $\hat{\sigma}_0^{\iota_0+1}$ and $\hat{\sigma}_1^{\iota_1+1}$ are useless. To verify why $\hat{\sigma}_0^{\iota_0+1}$ and $\hat{\sigma}_1^{\iota_1+1}$ are useless we can iterate through the algorithm. The first iteration detects the Z-cycle: $\hat{\sigma}_0^{\iota_0+1} \rightarrow \hat{\sigma}_3^{\iota_3+1}$, $\hat{\sigma}_3^{\iota_3} \rightarrow \hat{\sigma}_1^{\iota_1+1}$, and

Class 4.5 PZCF_Pattern.java

```
public class PZCF_Pattern {

    VC_Ckpt[ ] C;  // Consistent global checkpoint
    VC_Ckpt[ ] S;  // Succeeding checkpoints of C

    private boolean[ ] M = new boolean[Process.N];
    private boolean[ ][ ] Z = new boolean[Process.N][Process.N];

    public void next() {

        for (int i=0; i < Process.N; i++) {
            M[i] = false;
            for (int j=0; j < Process.N; j++) Z [i][j] = false;
        }

        for (int i=0; i < Process.N; i++)
            for (int j=0; j < Process.N; j++) {
                if (S[j].precedes(S[i]))
                    mark(j,i);
            }

        for (int i=0; i < Process.N; i++)
            if (Z[i][i]) { S[i] = null; }   // useless checkpoint
            else if (!M[i]) { C[i] = S[i]; S[i] = null; }
    }

    protected void mark(int j, int i) {
        if (!Z[j][i]) {
            Z[j][i] = M[i] = true;
            for (int k=0; k < Process.N; k++)
                if (C[i].precedes(S[k])) mark (i,k);
        }
    }
}
```

$\hat{\sigma}_1^{\iota_1} \rightarrow \hat{\sigma}_0^{\iota_0+1}$, discarding $\hat{\sigma}_0^{\iota_0+1}$. The second iteration detects the Z-cycle: $\hat{\sigma}_1^{\iota_1+1} \rightarrow \hat{\sigma}_0^{\iota_0+2}$, $\hat{\sigma}_0^{\iota_0} \rightarrow \hat{\sigma}_3^{\iota_3+1}$, $\hat{\sigma}_3^{\iota_3} \rightarrow \hat{\sigma}_1^{\iota_1+1}$, discarding $\hat{\sigma}_1^{\iota_1+1}$. Finally, the monitor is able to construct $\hat{\Sigma}' = \{\hat{\sigma}_0^{\iota_0+2}, \hat{\sigma}_1^{\iota_1+2}, \hat{\sigma}_2^{\iota_2+1}, \hat{\sigma}_3^{\iota_3+1}\}$.

Optimization: It is possible to modify the algorithms to work even if the set S has some checkpoints missing. Consider a pair of distinct processes, say p_i and p_j. The precedence test `S[j].precedes(S[i])` that evaluates whether $VC(S[j])[j] \leq VC(S[i])[j]$ could be substituted for one that evaluates whether $VC(C[j])[j] < VC(S[i])[j]$. This would be useful for a monitor to construct consistent global checkpoints without having to wait for all processes. If S is incomplete, however, the algorithm is not guaranteed to make progress.



$\hat{\sigma}_0^{\iota_0}$
$\hat{\sigma}_0^{\iota_0+1}$
$\hat{\sigma}_0^{\iota_0+2}$
$\hat{\sigma}_1^{\iota_1}$
$\hat{\sigma}_1^{\iota_1+1}$
$\hat{\sigma}_1^{\iota_1+2}$
$\hat{\sigma}_2^{\iota_2}$
$\hat{\sigma}_2^{\iota_2+1}$
$\hat{\sigma}_3^{\iota_3}$
$\hat{\sigma}_3^{\iota_3+1}$
p_0
p_1
p_2
p_3
$\hat{\Sigma}$
S
$Step(\hat{\Sigma}) = \emptyset$

Figure 6: Progressive view in a PZCF pattern

5 Z-precedence versus Zigzag Paths

Netzer and Xu [12] have demonstrated that the absence of zigzag paths is a necessary and sufficient condition for a set of checkpoints to be part of the same consistent global checkpoint. In this Section, we demonstrate that, although expressed in a simpler manner, a Z-precedence between a pair of checkpoints is equivalent to a zigzag path [12] connecting them, as defined by Manivannan, Netzer and Singhal [9].

Let S be a set of checkpoints and let $S \not\rightsquigarrow S$ indicate that no checkpoint in S Z-precedes a checkpoint (including itself) in S. Given a single checkpoint $\hat{\sigma}$, let $\hat{\sigma} \not\rightsquigarrow S$ indicate that $\hat{\sigma}$ does not Z-precedes any checkpoint in S. We are going to prove that if $S \not\rightsquigarrow S$, there exists a consistent global checkpoint $\hat{\Sigma}$ including S.

Theorem 5.1 *A set of checkpoints S can be part of the same consistent global checkpoint if $S \not\rightsquigarrow S$.*

Proof: We construct a consistent global checkpoint $\hat{\Sigma}$ using S and considering, for every process p_j that does not have a checkpoint in S, the checkpoint $\hat{\sigma}_j^{\iota_j}$ such that

$$\iota_j = \min\{\gamma : \hat{\sigma}_j^{\gamma} \not\rightsquigarrow S\}$$

We should note that if $\iota_j > 0$ there exists a checkpoint $\hat{\sigma}_i^{\iota_i} \in S$ such that $\hat{\sigma}_j^{\iota_j-1} \rightsquigarrow \hat{\sigma}_i^{\iota_i}$.

Assume that $\hat{\Sigma}$ is not consistent. There must exist a causal precedence between a pair of checkpoints in $\hat{\Sigma}$, say $\hat{\sigma}_a^{\alpha}$ and $\hat{\sigma}_b^{\beta}$:

$$(\hat{\sigma}_a^{\alpha} \rightarrow \hat{\sigma}_b^{\beta}) \Rightarrow (\hat{\sigma}_a^{\alpha} \rightsquigarrow \hat{\sigma}_b^{\beta})$$

We should note that $\beta > 0$, because no checkpoint precedes an initial checkpoint. There are four possibilities of membership for these checkpoints:

- $\hat{\sigma}_a^{\alpha} \in S$, $\hat{\sigma}_b^{\beta} \in S$: Since $\hat{\sigma}_a^{\alpha} \rightsquigarrow \hat{\sigma}_b^{\beta}$, this violates the hypothesis that $S \not\rightsquigarrow S$.

- $\hat{\sigma}_a^\alpha \in S$, $\hat{\sigma}_b^\beta \in \hat{\Sigma} - S$: There must exist $\hat{\sigma}_i^{\iota i} \in S$ such that $\hat{\sigma}_b^{\beta-1} \rightsquigarrow \hat{\sigma}_i^{\iota i}$. However, $\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_b^\beta$ and $\hat{\sigma}_b^{\beta-1} \rightsquigarrow \hat{\sigma}_i^{\iota i}$, implies that $\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_i^{\iota i}$, with $\hat{\sigma}_a^\alpha, \hat{\sigma}_i^{\iota i} \in S$ and this also violates the hypothesis that $S \not\rightsquigarrow S$.

- $\hat{\sigma}_a^\alpha \in \hat{\Sigma} - S$, $\hat{\sigma}_b^\beta \in S$: The Z-precedence $\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_b^\beta$ violates the rule used to build $\hat{\Sigma}$.

- $\hat{\sigma}_a^\alpha \in \hat{\Sigma} - S$, $\hat{\sigma}_b^\beta \in \hat{\Sigma} - S$: As in case (2), there must exist $\hat{\sigma}_c^\gamma \in S$ such that $\hat{\sigma}_b^{\beta-1} \rightsquigarrow \hat{\sigma}_i^{\iota i}$. Since $\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_b^\beta$ and $\hat{\sigma}_b^{\beta-1} \rightsquigarrow \hat{\sigma}_i^{\iota i}$ implies that $\hat{\sigma}_a^\alpha \rightsquigarrow \hat{\sigma}_i^{\iota i}$, this violates the rule used to build $\hat{\Sigma}$. □

Theorem 3.1 states that checkpoints related by the Z-precedence cannot be part of the same consistent global checkpoint. Theorem 5.1 demonstrates that a set of checkpoints unrelated by the Z-precedence can be extended to form a consistent global checkpoint. Therefore, the absence of a Z-precedence between a pair of checkpoints is a necessary and sufficient condition for them to be part of the same consistent global checkpoint. Thus, Z-precedence is equivalent to zigzag path, but represents a simpler abstraction, easier to write and to reason about.

Clearly, the Z-precedence relation is a generalization of the Lamport's causal precedence (Definition 3.4, first part). The same definition, second part, captures an extended notion of transitivity between checkpoints that has been used to analyze and write proofs on consistent global checkpoints. In particular, it was a valuable tool for the design of the algorithms that build a progressive view of a computation.

The Z-precedence relation is based *only* on checkpoints and causal precedence. This is consistent with the abstraction level we are working with: events, including messages, belong to a lower level of abstraction. This makes it possible to apply the results obtained to other computational models. For example, a computational model that is in evidence nowadays is the object and action model [14]. In this model, distributed atomic actions are used to organize the flux of method invocations on the set of objects that form the application. We have already applied our work [6] to asynchronously construct a progressive view of a computation in this model [5]; we believe that our result represented an advance in relation to an earlier result [4].

6 Conclusions

Algorithms for obtaining consistent global checkpoints are an useful aid in a vast class of problems that can be formulated as the evaluation of a predicate over the global state of an application [2]. In this paper we have introduced the notion of a *progressive view* of a computation: a sequence of consistent global checkpoints that may have occurred in this order during the execution of an application. We have also proposed original algorithms to construct such a view. We believe that monitoring and dynamic reconfiguration of distributed systems can benefit from such algorithms to guarantee that reconfiguration occurs adequately.

During the development of our algorithms we have determined the exact conditions under which a checkpoint must be observed before another in a progressive view. The *Z-precedence* between checkpoints is a generalization of the Lamport's causal precedence [8], and it is equivalent to the zigzag paths proposed by Netzer and Xu [12]. Besides its technical advantages, we believe that the intuitive meaning of the Z-precedence can provide a better understanding of consistent global checkpoints.

The number of applications that require not only on-the-fly but also *a posteriori* collection and analysis of consistent global checkpoints represents a considerable part of modern distributed applications. For these applications, Z-precedence can be used to reason about and build algorithms that may succeed in reducing the overhead related to the processing of consistent global checkpoints.

Acknowledgment

This work has been supported by FAPESP under grant no. 95/1983-8 for Islene Calciolari Garcia and grant no. 96/1532-9 for the Laboratory of Distributed Systems. We would like to thank Alexandre Oliva for his help with the paper. We would also like to thank our colleagues from the Institute of Computing, specially from the Laboratory of Distributed Systems, for providing us with interesting and motivating environment.

References

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

[2] R. Cooper and K. Marzullo. Consistent Detection of Global Predicates. *SIGPLAN Notices*, 26(12):167–174, Dec. 1991.

[3] E. N. Elnozahy, D. Johnson, and Y.M.Yang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, 1996.

[4] M. J. Fischer, N. D. Griffeth, and N. A. Lynch. Global States of a Distributed System. *IEEE Transactions on Software Engineering*, SE-8(3):198–202, may 1982.

[5] I. C. Garcia. Estados Globais Consistentes em Sistemas Distribuídos. Master's thesis, Instituto de Computação—Universidade Estadual de Campinas, July 1998. In Portuguese.

[6] I. C. Garcia and L. E. Buzato. Asynchronous Construction of Consistent Global Snapshots in the Object and Action Model. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, Annapolis, Maryland, EUA, May 1998. IEEE. Available as Technical Report IC–98–16 at `<http://www.dcc.unicamp.br/ic-tr/>`.

[7] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification.* Java Series. Addison–Wesley, Sept. 1996. Version 1.0.

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

[9] D. Manivannan, R. H. B. Netzer, and M. Singhal. Finding Consistent Global Checkpoints in a Distributed Computation. In *IEEE Transactions on Parallel and Distributed Systems*, pages 623–627, June 1997.

[10] D. Manivannan and M. Singhal. Quasi–Synchronous Checkpointing: Models, Characterization, and Classification. Technical Report OH 43210, Department of Computer and Information Science, The Ohio State University, 1997.

[11] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[12] 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.

[13] Sun Microsystems Computer Corporation, Mountain View, CA, USA. *Java API Documentation*, Dec. 1996. Version 1.1.

[14] S. M. Wheater and D. L. McCue. Configuring Distributed Applications using Object Decomposition in an Atomic Action Environment. In J. Kramer, editor, *Proceedings of the International Workshop on Configurable Distributed Systems*, pages 33–44. IEE (UK), Imperial College of Science, Technology and Medicine, UK, March 1992. ISBN 0-85296-544-3.