Coleta de Lixo para Protocolos de *Checkpointing*

Rodrigo Malta Schmidt

Dissertação de Mestrado

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Schmidt, Rodrigo Malta

Sch55c Coleta de lixo para protocolos de checkpointing / Rodrigo Malta Schmidt -- Campinas, [S.P. :s.n.], 2003.

Orientadores : Luiz Eduardo Buzato; Islene Calciolari Garcia

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Processamento distribuído. 2. Algoritmos. 3. Tolerância a falhas. I. Buzato, Luiz Eduardo. II. Garcia, Islene Calciolari. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Instituto de Computação
Universidade Estadual de Campinas

Coleta de Lixo para Protocolos de *Checkpointing*

Rodrigo Malta Schmidt[1]

3 de Outubro de 2003

Banca Examinadora:

- Prof. Dr. Luiz Eduardo Buzato (Orientador)
- Prof. Dr. Fábio Kon
  Instituto de Matemática e Estatística — USP
- Prof. Dr. Ricardo de Oliveira Anido
  Instituto de Computação — Unicamp
- Prof. Dr. Edmundo Roberto Mauro Madeira (Suplente)
  Instituto de Computação — Unicamp

[1]**Apoio financeiro da CAPES. Apoio financeiro adicional da FAPESP, processo número 96/1532-9 para o Laboratório de Sistemas Distribuídos e processo número 97/10982-0 para o Laboratório de Computação de Alto Desempenho.**

Coleta de Lixo para Protocolos de *Checkpointing*

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Rodrigo Malta Schmidt e aprovada pela Banca Examinadora.

Campinas, 10 de Outubro de 2003.

Prof. Dr. Luiz Eduardo Buzato (Orientador)

Profª Drª Islene Calciolari Garcia
(Co-orientadora)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Agradecimentos

À Bianca, por ser esta pessoa tão especial que me incentivou nos momentos difíceis e compartilhou comigo dos momentos de alegria, tornando-os ainda mais felizes. Seu amor, compreensão e apoio foram fundamentais para o desenvolvimento deste trabalho. Sua dedicação, paciência e responsabilidade sempre me serviram de exemplo, apesar de nunca ter conseguido me igualar a ela nestes aspectos.

Aos meus pais, Ronaldo e Zadir, e aos meus padrinhos, Dilson e Zair, pela educação e apoio, que me ajudaram a chegar até aqui.

À minha irmã, Carolina, pelo carinho.

Ao Buzato, meu orientador, pela motivação e confiança.

À Islene, minha co-orientadora, pela ajuda na transcrição de minhas idéias para o papel e pelas várias sugestões para a escrita deste texto e para a elaboração da apresentação final.

Ao Ulisses, por dar continuidade a este trabalho e por assistir várias prévias de minha apresentação, contribuindo com importantes sugestões.

Ao Prof. Fernando Pedone, por ter me incentivado desde a época do colégio técnico e por ter me apresentado à área de Algoritmos Distribuídos.

Ao meu amigo e ex-colega de república, Felipe Renon, por ter conseguido conviver comigo durante um ano.

Aos vários amigos e colegas do IC, principalmente Eduardo, Felipe Pereira, Felipe Renon, Guilherme e Lásaro, pelas experiências compartilhadas.

A todos os meus colegas do Laboratório de Sistemas Distribuídos, principalmente Cândida, Fábio, Gustavo, Islene, Oliva e Tiemi, pela companhia e pela amizade.

Aos meus colegas de competições de programação, principalmente Guilherme, Elbio, Felipe, Pedro Demasi, Rodolfo, Vinícius e Ulisses, que me ajudaram a desenvolver, ao longo dos anos, um interesse especial pela área de projeto e análise de algoritmos.

Aos professores do Instituto de Computação da Unicamp e do curso de Engenharia de Computação da FURG, por terem colaborado muito para a minha formação.

Aos membros da banca examinadora pelas sugestões e pelo incentivo.

À CAPES, pelo apoio financeiro.

Resumo

Checkpointing é o nome dado à tarefa de seleção de *checkpoints* em sistemas distribuídos tolerantes a falhas baseados em recuperação por retrocesso, para a qual existem inúmeros protocolos propostos na literatura. Como *checkpoints* consomem recursos de armazenamento, um outro problema importante é a eliminação dos *checkpoints* que se tornam obsoletos à medida que a computação progride e novos *checkpoints* são gravados pelos processos. Este problema, chamado de coleta de lixo, é o foco principal desta dissertação.

Revisitamos o problema de coleta de lixo com uma abordagem baseada na relação de precedência existente entre os *checkpoints* (Z-precedência). Como os protocolos para *checkpointing* costumam restringir esta relação, nossa abordagem se mostra uma ferramenta importante para o desenvolvimento de algoritmos de coleta de lixo eficientes. Além disso, apresentamos um algoritmo de coleta de lixo original para protocolos de *checkpointing* que garantem que todas as precedências existentes entre os *checkpoints* são causais. Nosso algoritmo, chamado RDT-LGC, é o primeiro que não troca mensagens de controle e que garante o limite ótimo para o número máximo de *checkpoints* armazenados por processo durante toda a sua execução.

Abstract

Checkpointing is the name given to the task of selecting checkpoints in fault-tolerant distributed systems based on rollback-recovery, for which there is a number of protocols proposed in the literature. As checkpoints require the use of storage resources, another important problem is the elimination of those checkpoints that become obsolete as computation progresses and new checkpoints are stored by processes. This problem, called garbage collection, is the main focus of this dissertation.

We revisit the garbage collection problem with an approach based on the precedence relation existing between checkpoints (Z-precedence). As checkpointing protocols usually restrict this relation, our approach seems to be an important tool for the development of efficient garbage collection algorithms. Moreover, we present an original garbage collection algorithm for checkpointing protocols which ensure that all precedences existing between checkpoints are causal. Our algorithm, named RDT-LGC, is the first one that does not exchange control messages and that ensures the optimal upper bound on the number of checkpoints stored by a process throughout its execution.

Conteúdo

Lista de Figuras

Lista de Algoritmos

Lista de Símbolos e Abreviaturas

Símbolo	Significado	Página
n	Número de processos da aplicação distribuída	5
P	Conjunto de processos da aplicação	5
p_i	Processo específico	5
e	Evento de um processo	6
e_i^γ	γ-ésimo evento do processo p_i	6
m	Mensagem da aplicação	6
$\rightarrow$	Relação de precedência causal	8
$\|\|$	Concorrência entre eventos	8
LC	Relógio lógico	8
$LC(e)$	Relógio lógico de um evento	9
$LC(m)$	Relógio lógico de uma mensagem	9
VC	Relógio vetorial	9
$VC(e)$	Relógio vetorial de um evento	9
$VC(m)$	Relógio vetorial de uma mensagem	9
$\mathcal{C}$	Corte na execução da computação distribuída	10
$\mathcal{P}$	Propriedade	11
σ	Estado corrente de um processo	13
s_i^γ	γ-ésimo *checkpoint* estável do processo p_i	14
$last\_s(i)$	Índice do último *checkpoint* estável do processo p_i	14
s_i^{last}	Último *checkpoint* estável do processo p_i	14
v_i	*Checkpoint* volátil do processo p_i	14
c	*Checkpoint* geral (estável ou volátil)	14
c_i^γ	γ-ésimo *checkpoint* geral do processo p_i	14
CCP	Padrão de *checkpoints* e mensagens	14
I_i^γ	γ-ésimo intervalo entre *checkpoints* do processo p_i	14
C	*Checkpoint* global	15
$\rightsquigarrow$	Relação de Z-precedência entre *checkpoints*	19
DV	Vetor de dependências	23

Símbolo	Significado	Página
$DV(e)$	Vetor de dependências de um evento	23
$DV(m)$	Vetor de dependências de uma mensagem	23
F	Conjunto de processos falhos	25
R_F	Linha de recuperação para o conjunto F de processos falhos	25
SZPF	Classe de protocolos quase-síncronos livres de todos os caminhos-Z	33
ZPF	Protocolos quase-síncronos livres de Z-precedências não-causais	33
ZCF	Protocolos quase-síncronos livres de ciclos-Z	33
PZCF	Protocolos quase-síncronos que permitem alguns ciclos-Z	33
NRAS	*No-Receive-After-Send* – protocolo da classe SZPF	34
CAS	*Checkpoint-After-Send* – protocolo da classe SZPF	36
RDT	Propriedade presente nos protocolos da classe ZPF	37
FDAS	*Fixed-Dependency-After-Send* – protocolo da classe ZPF	38
BCS	Briatico-Ciuffoletti-Simoncini – protocolo da classe ZCF	38
$\mathcal{L}$	Corte no futuro da execução da computação distribuída	59
DEP	Conjunto de dependências dos *checkpoints* da computação	64
DEP_i	Dependências dos *checkpoints* de p_i	64
N_OBSOL	Conjunto de *checkpoints* não-obsoletos do CCP	64
UC	Vetor de índices dos últimos *checkpoints* estáveis dos processos	72
UC_i	Vetor UC mantido localmente pelo processo p_i	73
$last\_k_i(j)$	Índice do último *checkpoint* estável de p_j conhecido por p_i	75
CM	Vetor de *checkpoints* mantidos utilizado pelo RDT-LGC	78
BCC	Bloco de controle de *checkpoints*	79
UI	Vetor de últimos intervalos entre *checkpoints* dos processos	82
IR	Índice de retrocesso	83
CM_i	Vetor CM mantido localmente pelo processo p_i	85

Capítulo 1

Introdução

Cada vez mais, aplicações se valem de processamento distribuído para resolver problemas que antes eram intratáveis em áreas como otimização combinatória, genômica, processamento de imagens e astronomia, entre outras. Atualmente, vários trabalhos estão sendo desenvolvidos relacionados à utilização de arquiteturas como *clusters* e *grids* de computadores para este fim. Entretanto, mesmo com as vantagens oferecidas pela divisão de tarefas entre várias máquinas, muitas dessas aplicações executam por longos períodos de tempo pela natureza combinatória dos problemas envolvidos. Além disso, uma topologia distribuída aumenta os riscos de uma falha parcial, o que poderia causar o reinício de toda a computação, acarretando a perda de recursos humanos e computacionais. Sendo assim, para evitar ou minimizar este problema, é necessário o desenvolvimento de mecanismos de tolerância a falhas.

Aplicações distribuídas tolerantes a falhas via retrocesso de estado são baseadas na determinação de um estado global consistente para o qual a computação pode retroceder após a ocorrência de uma falha parcial, chamado de linha de recuperação. Um *checkpoint* corresponde a um estado de execução de um determinado processo, geralmente gravado em meio de armazenamento estável, que pode ser utilizado no retrocesso da computação. O conjunto formado por um *checkpoint* de cada processo é chamado de *checkpoint* global. No entanto, devido às dependências criadas pela troca de mensagens, nem todos os *checkpoints* globais representam estados globais consistentes da computação. Os *checkpoints* globais que respeitam esta propriedade são chamados de *checkpoints* globais consistentes e são eles que devem ser utilizados quando o estado da computação precisar ser retrocedido.

Existem vários protocolos para a seleção dos estados dos processos que devem ser gravados como *checkpoints* de forma a garantir um retrocesso pequeno após uma falha. Em geral, estes protocolos deixam a aplicação selecionar seus próprios *checkpoints* e podem forçar a seleção de *checkpoints* extras que garantam o avanço da linha de recuperação, por meio de uma abordagem assíncrona, síncrona ou quase-síncrona. Na abordagem assíncrona os *checkpoints* são selecionados sem nenhuma coordenação, o que pode impedir o avanço da

linha de recuperação. Protocolos síncronos utilizam uma sincronização explícita baseada em mensagens de controle. Quando a aplicação seleciona um *checkpoint*, o protocolo se encarrega de trocar mensagens com os outros processos de forma a sincronizá-los para a composição de um *checkpoint* global consistente. Por fim, a abordagem quase-síncrona utiliza *timestamps* nas mensagens da própria aplicação como forma de sincronização. Dessa maneira, quando um processo recebe uma mensagem, o protocolo para *checkpointing* avalia o *timestamp* recebido e decide se deve forçar a seleção de um novo *checkpoint*.

Na medida que a aplicação executa e novos *checkpoints* são selecionados, novas linhas de recuperação se formam e os *checkpoints* mais antigos se tornam obsoletos, não sendo mais necessários para nenhuma linha de recuperação futura. Se o espaço utilizado por estes *checkpoints* obsoletos não for reaproveitado, pode-se esgotar a capacidade do meio de armazenamento estável, o que impediria a seleção de novos *checkpoints*. A solução deste problema está na utilização de algoritmos para a coleta de lixo, responsáveis por periodicamente identificar e liberar o espaço ocupado por *checkpoints* obsoletos.

Protocolos para *checkpointing* síncronos, em sua grande maioria, criam um novo *checkpoint* global consistente sempre que um processo seleciona um novo *checkpoint*. Desta forma, uma linha de recuperação completa é criada e os *checkpoints* anteriores a ela podem ser eliminados. Protocolos assíncronos e quase-síncronos, por outro lado, têm sua coleta de lixo prejudicada pela ausência de sincronização explícita para a seleção de *checkpoints*. Os algoritmos utilizados na literatura para coleta de lixo nessas classes de protocolos não apresentam um limite para o número de *checkpoints* não coletados ou necessitam de alguma forma de sincronização explícita entre os processos. A ausência de um limite para o número de *checkpoints* não coletados impede a determinação da capacidade máxima necessária para o meio de armazenamento estável, o que é muito ruim em modelos onde tais dispositivos têm capacidade reduzida ou custo alto, como ocorre em computação móvel e em sistemas embutidos. Além disso, a necessidade de sincronização se opõe às vantagens dos protocolos assíncronos e quase-síncronos que não requerem sincronização dos processos para a seleção de *checkpoints*.

Neste contexto, nossa primeira contribuição consiste em uma nova abordagem teórica para a análise do problema de coleta de lixo, baseada na relação de precedência que ocorre entre os *checkpoints* (Z-precedência). Esta relação costuma ser restringida pelos protocolos para *checkpointing*. Desta maneira, nossa abordagem permite ver o problema de coleta de lixo de uma forma mais próxima dos protocolos para *checkpointing* e pode servir de base para a construção de algoritmos de coleta de lixo mais eficientes.

Além disso, apresentamos um novo algoritmo de coleta de lixo para protocolos de *checkpointing* que garantem que todas as dependências entre os *checkpoints* são causais. Este algoritmo garante o limite ótimo para o número de *checkpoints* não coletados por processo, não faz nenhuma consideração sobre sincronismo de relógio entre os processos e não troca nenhuma mensagem extra além daquelas da própria aplicação distribuída, além de ser baseado na propagação dos mesmos *timestamps* utilizados pela classe de protocolos quase-síncronos

que garante este tipo de dependência entre os *checkpoints*. Com ele é possível construir um sistema de recuperação por retrocesso baseado em *checkpointing* totalmente quase-síncrono e com limite máximo para a quantidade de armazenamento estável necessária, vantagens não encontradas em nenhum sistema de recuperação até o momento.

Esta dissertação está organizada da seguinte forma. No Capítulo 2 são apresentados diversos aspectos teóricos e práticos referentes ao processo de *checkpointing* e recuperação por retrocesso. O Capítulo 3 descreve diversos protocolos para *checkpointing* e os compara sob diferentes pontos de vista. O problema de coleta de lixo é abordado de maneira geral no Capítulo 4 e um algoritmo eficiente para protocolos de *checkpointing* que garantem apenas dependências causais entre *checkpoints* é apresentado no Capítulo 5.

Capítulo 2

Checkpointing e Recuperação

A construção de um estado global recuperável é feita pela união de *checkpoints* pertencentes a cada processo da aplicação distribuída. No entanto, a dependência de eventos provocada pela troca de mensagens entre os processos dificulta este procedimento, podendo inclusive fazer com que uma simples falha force a computação a retroceder ao seu estado inicial. Este problema é conhecido como *efeito dominó* [29]. Chama-se de *checkpointing* o procedimento empregado pela aplicação para decidir quais estados dos processos individuais serão armazenados na forma de *checkpoints*, geralmente tentando-se prevenir a ocorrência do efeito dominó.

Um problema ortogonal ao da seleção dos *checkpoints* em sistemas tolerantes a falhas baseados em recuperação por retrocesso de estado refere-se à identificação das falhas e sua recuperação por meio do mecanismo de retrocesso. A principal tarefa desta etapa consiste em identificar, dado o conjunto de processos falhos, o *checkpoint* global consistente mais recente a ser utilizado como linha de recuperação.

Este capítulo apresenta estes problemas detalhadamente e fornece a fundamentação teórica necessária para a compreensão dos resultados desta pesquisa. Primeiramente, apresentaremos o modelo computacional adotado em nosso estudo, abordando as definições de causalidade de eventos e de consistência de estados. Na seqüência, são apresentados diversos aspectos relativos à consistência de um conjunto de *checkpoints* e à obtenção de um *checkpoint* global consistente. Na última seção são apresentadas várias considerações referentes ao mecanismo de retrocesso e à definição e cálculo da linha de recuperação.

2.1 Modelo Computacional e Definições

Uma *aplicação distribuída* é composta por um conjunto de n *processos* seqüenciais cooperativos: $P = \{p_0, p_1, \ldots, p_{n-1}\}$. Supõe-se um modelo de computação assíncrona, no qual os processos são autônomos e não têm acesso a um relógio global. Os processos executam

de forma totalmente não-determinística, o que impede a adoção de técnicas como *message logging* [3] que são baseadas no modelo PWD (*PieceWise Determinism*) [32]. Além disso, nosso modelo não permite o compartilhamento de memória entre os processos, restringindo a sua comunicação apenas à possível troca de mensagens sobre uma *rede de comunicação*.

A rede de comunicação é definida por um grafo orientado composto por n vértices que representam os processos da aplicação e um conjunto de arestas orientadas que representam a existência de canais unidirecionais diretos entre dois processos. As mensagens só podem ser enviadas por meio destes canais, mas supõe-se a existência de um mecanismo de comunicação capaz de conduzir uma mensagem através de diversos canais até o seu destino. Sendo assim, um processo pode enviar mensagens para qualquer outro processo para o qual exista um caminho a partir dele na rede de comunicação. O mecanismo de troca de mensagens garante que elas não são corrompidas. Porém, o tempo de entrega não é limitado superiormente (variando entre 0 e ∞) e as mensagens podem ser entregues em uma ordem diferente da ordem de envio.

A execução de um processo é definida como uma seqüência de eventos $(e_i^0, e_i^1, \dots)$, onde e_i^γ representa o γ-ésimo evento executado pelo processo p_i. Tais eventos, por sua vez, podem ser classificados em dois tipos: eventos internos e eventos de comunicação. Os eventos internos estão relacionados com a execução local de um processo, enquanto os eventos de comunicação representam o envio ou a entrega de uma mensagem. O estado corrente de um processo é mantido em meio de armazenamento volátil e pode ser mudado exclusivamente pela ocorrência de eventos. Desta forma, podemos associar o estado de um processo em um determinado instante de tempo ao último evento executado pelo processo em questão.

A maneira usual de se representar graficamente a execução de uma computação distribuída é por meio de um diagrama de espaço-tempo [24]. Linhas horizontais representam a execução dos processos ao longo do tempo, que progride da esquerda para a direita. Eventos são representados como pontos sobre estas retas e mensagens trocadas por processos são representadas por setas ligando o evento de envio ao evento de entrega da mensagem. A Figura 2.1 é um exemplo de diagrama de espaço-tempo representando os eventos de 3 processos.

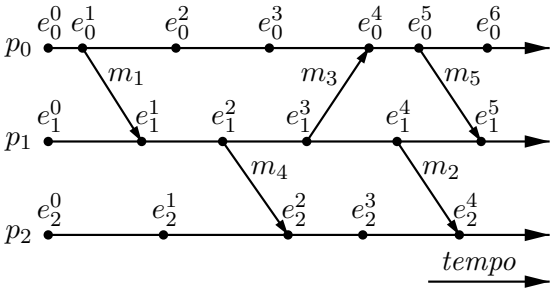

Figura 2.1: Diagrama de espaço-tempo.

Processos podem sofrer falhas do tipo *crash* nas quais param e perdem o seu estado corrente de execução. No entanto, cada processo tem acesso a um meio de armazenamento estável local ou global que pode ser utilizado para gravar o seu estado como um *checkpoint*. O armazenamento de um *checkpoint* é um evento interno do processo em execução, mas que não altera seu estado atual. Entretanto, para prover tolerância a falhas é necessário, além de um mecanismo para controlar a seleção e o armazenamento de *checkpoints*, um sistema para a detecção e outro para a recuperação de falhas.

O mecanismo de detecção de falhas pode ser facilmente implementado por meio de *watchdog daemons* capazes de monitorar a execução dos processos [21]. Sendo assim, quando um conjunto de processos falhos é detectado, o sistema de recuperação é acionado. Este sistema primeiramente identifica o conjunto de processos falhos. Depois disso ele pode aguardar que tais processos sejam reiniciados ou criar novos processos que passarão a responder por eles. No entanto, esta segunda alternativa só é válida se os processos identificados tiverem gravado seus *checkpoints* em um meio de armazenamento estável global, que possa ser acessado pelos novos processos criados. Uma vez que todos os processos estejam novamente prontos para continuar sua execução, o sistema de recuperação deve determinar a linha de recuperação a ser utilizada e forçar todos os processos a recomeçarem a sua execução a partir de seu *checkpoint* local correspondente. Sendo assim, nosso modelo de execução da computação distribuída é dividido em períodos de execução normal alternados com sessões de recuperação, conforme é mostrado na Figura 2.2.

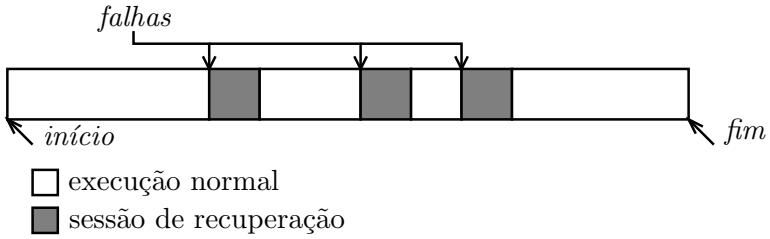

Figura 2.2: Modelo de execução da aplicação distribuída.

2.2 Causalidade e Consistência

Em computações distribuídas baseadas no modelo assíncrono, a inexistência de um relógio global impede uma ordenação total dos eventos baseada nos seus instantes de execução. Contudo, a ordem de execução entre os eventos de um mesmo processo e a ordem implícita entre os eventos de envio e entrega de uma mensagem podem ser utilizadas para a construção de uma ordenação dos eventos por meio da *relação de precedência causal* [24]. Esta relação indica quando um evento necessariamente ocorreu antes de outro na computação distribuída e é a base para a definição de estados e *checkpoints* globais consistentes.

2.2.1 Precedência Causal de Eventos

A precedência causal captura a dependência entre eventos. Dois eventos ligados por uma relação de causa e efeito têm uma ordem intrínseca que deve ser respeitada em uma ordenação consistente dos eventos do sistema. Dessa forma, um evento e precede causalmente outro evento e' se e pode ter influenciado a ocorrência de e'.

Definição 2.1 ***Precedência Causal*** — *Um evento e_a^α precede causalmente um evento e_b^β ($e_a^\alpha \rightarrow e_b^\beta$) se, e somente se,*

(i) $a = b$ *e* $\beta = \alpha + 1$*; ou*

(ii) $\exists m \mid e_a^\alpha = envio(m)$ *e* $e_b^\beta = entrega(m)$*; ou*

(iii) $\exists e_c^\gamma \mid e_a^\alpha \rightarrow e_c^\gamma \wedge e_c^\gamma \rightarrow e_b^\beta$.

Dois eventos de um mesmo processo que ocorrem em seqüência representam uma dependência clara do segundo com relação ao primeiro, uma vez que a execução do segundo evento ocorre em um estado gerado após a execução do primeiro. O mesmo acontece com os eventos de envio e entrega de uma mensagem pois não é possível entregá-la sem que ela tenha sido enviada. Além disso, a relação de precedência causal é transitiva; se um evento depende de outro evento, também depende de todas as dependências deste evento.

A relação de precedência causal é não-reflexiva pois é impossível que um evento dependa de si próprio. Para isso ocorrer, uma mensagem teria que ser entregue antes de ter sido enviada. Dois eventos e e e' são ditos concorrentes ($e \mid\mid e'$) quando não existe precedência causal entre eles, isto é, $e \not\rightarrow e' \wedge e' \not\rightarrow e$. A relação de precedência causal não define uma ordenação total dos eventos do sistema porque não estabelece uma ordem para os eventos concorrentes. De forma geral, se é necessária uma ordenação total dos eventos, uma ordenação arbitrária dos eventos concorrentes é suficiente, mantendo-se consistente com a relação de causalidade.

2.2.2 Captura da Precedência Causal

É possível capturar a relação de precedência causal entre eventos durante a execução da computação distribuída. Para isto, é necessário utilizarmos mecanismos chamados de relógios. A forma mais simples desta estrutura são os *relógios lógicos*[1] (LC), definidos por Lamport no mesmo trabalho em que define a relação de dependência causal [24]. Cada processo mantém um contador inteiro que representa o tempo lógico da ação corrente, o qual é incrementado a cada evento. Toda mensagem enviada por um processo carrega consigo, na forma de um *timestamp*, o valor do seu relógio lógico no momento do envio. Quando uma mensagem é

[1]Em inglês: *Logical Clocks*.

recebida, o processo que a recebe atualiza o seu relógio para o máximo entre o seu valor atual e o valor recebido na mensagem. Utilizamos $LC(e)$ para representar o valor do relógio lógico de um processo após a ocorrência do evento e e $LC(m)$ para representar o valor do relógio enviado como *timestamp* na mensagem m. A Figura 2.3 apresenta a propagação de relógios lógicos para o diagrama de espaço-tempo apresentado anteriormente na Figura 2.1.

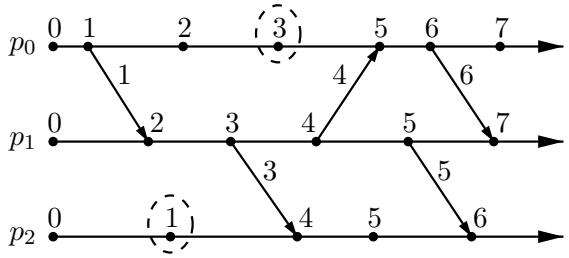

Figura 2.3: Propagação de relógios lógicos.

Pelo mecanismo de propagação e atualização de relógios lógicos, podemos facilmente chegar a uma relação direta com a precedência causal. Se um evento e precede causalmente outro evento e', então $LC(e) < LC(e')$ [12]:

$$e \rightarrow e' \Rightarrow LC(e) < LC(e') \tag{2.1}$$

Porém, apesar de consistentes com a relação de precedência, os relógios lógicos não a caracterizam. Dados dois eventos e e e' tais que $LC(e) < LC(e')$, não é possível determinar se $e \rightarrow e'$ ou se $e \mid\mid e'$. Como exemplo, considere os eventos e_2^1 e e_0^3 destacados na Figura 2.3. Como $LC(e_2^1) = 1$ e $LC(e_0^3) = 3$, verificamos que $LC(e_2^1) < LC(e_0^3)$ e $e_2^1 \mid\mid e_0^3$.

Para capturar completamente a relação de precedência causal precisamos acrescentar, para cada evento, informação a respeito de todos os eventos da computação que o precedem causalmente. Isto pode ser feito por meio de *relógios vetoriais*[2] (VC) [12]. Cada processo p_i mantém um vetor de n entradas que armazena o histórico causal completo do evento corrente, sendo que a entrada $VC[i]$ é incrementada a cada novo evento local. Toda mensagem enviada por um processo carrega consigo, na forma de um *timestamp*, o relógio vetorial do processo remetente no instante do envio. Quando uma mensagem é recebida, o processo que a recebe compara o seu relógio vetorial corrente com o relógio vetorial recebido na mensagem e atualiza cada entrada do seu relógio com o maior dentre os dois valores (existente e recebido). De forma semelhante aos relógios lógicos, utilizamos $VC(e)$ para representar o valor do relógio vetorial de um processo após a ocorrência do evento e e $VC(m)$ para representar o relógio vetorial enviado como *timestamp* na mensagem m. A Figura 2.4 apresenta a propagação de relógios vetoriais para o diagrama de espaço-tempo apresentado anteriormente na Figura 2.1.

[2]Em inglês: *Vector Clocks*.

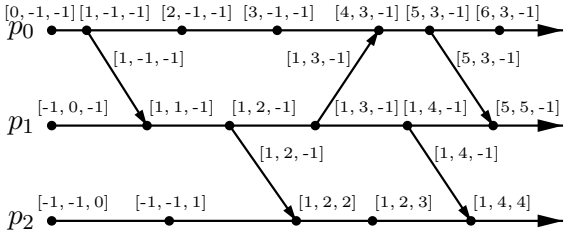

Figura 2.4: Propagação de relógios vetoriais.

Por representar todo o histórico causal de um evento, é possível demonstrar as seguintes relações entre relógios vetoriais e a precedência causal [12]:

$$e \rightarrow e' \iff e \neq e' \ \wedge \ \forall i \ : \ VC(e)[i] \leq VC(e')[i] \tag{2.2}$$

$$e_i \rightarrow e_j \iff \begin{cases} VC(e_i)[i] \leq VC(e_j)[i], & (i \neq j) \\ VC(e_i)[i] < VC(e_j)[i], & (i = j) \end{cases} \tag{2.3}$$

A existência destas relações nos permite caracterizar a precedência causal entre eventos e afirmar, comparando seus relógios vetoriais, se dois eventos estão relacionados causalmente ou são concorrentes.

2.2.3 Cortes Consistentes e Propriedades Estáveis

Um *corte* de uma computação distribuída é um conjunto formado por prefixos das seqüências de eventos executadas por cada processo [12]. Para cada processo p_i, um corte $\mathcal{C}$ contém os seus primeiros c_i eventos, para um número natural c_i. Sendo assim, um corte pode ser especificado por um conjunto destes n números naturais $\{c_0, c_1, \ldots, c_{n-1}\}$. O conjunto dos últimos eventos de cada processo em um corte $\{e_0^{c_0}, e_1^{c_1}, \ldots, e_{n-1}^{c_{n-1}}\}$ corresponde à *fronteira do corte*. Um *estado global* é um conjunto dos estados locais de cada um dos processos e é determinado pelos estados dos processos na fronteira de um corte. Se um corte $\mathcal{C}$ está contido em um corte $\mathcal{C}'$ ($\mathcal{C} \subset \mathcal{C}'$), dizemos que $\mathcal{C}$ está no passado de $\mathcal{C}'$ e, dualmente, que $\mathcal{C}'$ está no futuro de $\mathcal{C}$.

Nem todo corte corresponde a um estado consistente da computação [12]. Este é o caso, por exemplo, quando incluímos o evento de entrega de uma mensagem sem incluir o seu respectivo evento de envio. Um corte consistente precisa ser coerente com a relação de precedência causal. Se um evento faz parte de um corte consistente, então todos os eventos que o precedem causalmente também devem fazer parte do corte. É interessante notar que podem existir relações de precedência entre dois eventos na fronteira do corte sem que este fato torne o corte inconsistente. Um estado global é consistente se for gerado pela fronteira

de um corte consistente. Portanto, um estado global consistente pode ser visto como uma possível observação do sistema que respeita a relação de precedência causal. Note que um estado consistente observado não precisa ter ocorrido durante a execução real do sistema mas poderia ocorrer em uma outra execução que gera o mesmo resultado final, trocando-se apenas alguns valores não-determinísticos como tempos de execução de eventos e de envio de mensagens [11, 12].

Definição 2.2 ***Corte Consistente*** — *Um corte $\mathcal{C}$ é consistente se, e somente se,*

$$e \in \mathcal{C} \wedge e' \rightarrow e \Rightarrow e' \in \mathcal{C}.$$

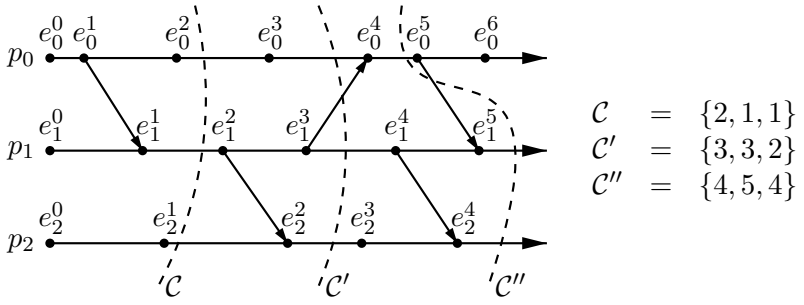

Figura 2.5: Cortes consistentes e inconsistente.

A Figura 2.5 ilustra três cortes $\mathcal{C}$, $\mathcal{C}'$ e $\mathcal{C}''$ por meio de linhas tracejadas que intersectam o diagrama de espaço-tempo. Os eventos que se situam à esquerda de uma linha fazem parte do corte. Neste caso, o critério de consistência de um corte pode ser visto graficamente da seguinte forma: se as setas que cruzam a linha que define o corte possuem sua origem do lado esquerdo e seu fim do lado direito, o corte é consistente, caso contrário ele é inconsistente. Então, temos que os cortes $\mathcal{C}$ e $\mathcal{C}'$ são consistentes e $\mathcal{C}''$ é inconsistente. Além disso, podemos ver que $\mathcal{C}'$ está no futuro de $\mathcal{C}$.

Cortes consistentes definem estados consistentes de uma computação distribuída. Portanto, a avaliação de propriedades distribuídas deve ser baseada neles. Neste contexto, podemos definir uma *propriedade estável* como aquela que uma vez verdadeira, permanecerá verdadeira até o final da execução, a menos que a computação sofra uma intervenção externa [11, 12]. Isto equivale a dizer que se uma propriedade estável é verdadeira em um corte consistente $\mathcal{C}$, será verdadeira em todos os cortes consistentes $\mathcal{C}'$ no futuro de $\mathcal{C}$.

Definição 2.3 ***Propriedade Estável*** — *Uma propriedade $\mathcal{P}$ é estável se, e somente se, uma vez que $\mathcal{P}$ seja válida em um corte consistente $\mathcal{C}$, $\mathcal{P}$ será válida em todo corte consistente $\mathcal{C}'$ tal que $\mathcal{C} \subset \mathcal{C}'$.*

Existem vários exemplos clássicos de propriedades estáveis em computação distribuída como a formação de um impasse[3] e o término da execução de um subconjunto dos processos. Não são propriedades estáveis a igualdade de variáveis alteradas pelos processos ou situações de bloqueio de processos por sua baixa prioridade (problema de inanição[4]).

2.3 *Checkpointing*

A seleção e o armazenamento de *checkpoints* envolvem diversos fatores que vão desde a estrutura interna dos processos da aplicação até a seleção correta de estados de forma a construir novos *checkpoints* globais consistentes, evitando a ocorrência do efeito dominó [29].

2.3.1 Estrutura dos Processos

Em uma aplicação onde um mecanismo de *checkpointing* é empregado, costuma-se estruturar os processos em vários sub-sistemas, cada um com sua função bem definida [5, 13]. Uma estrutura geral que comporta a implementação de todos os protocolos abordados nesta dissertação é apresentada na Figura 2.6.

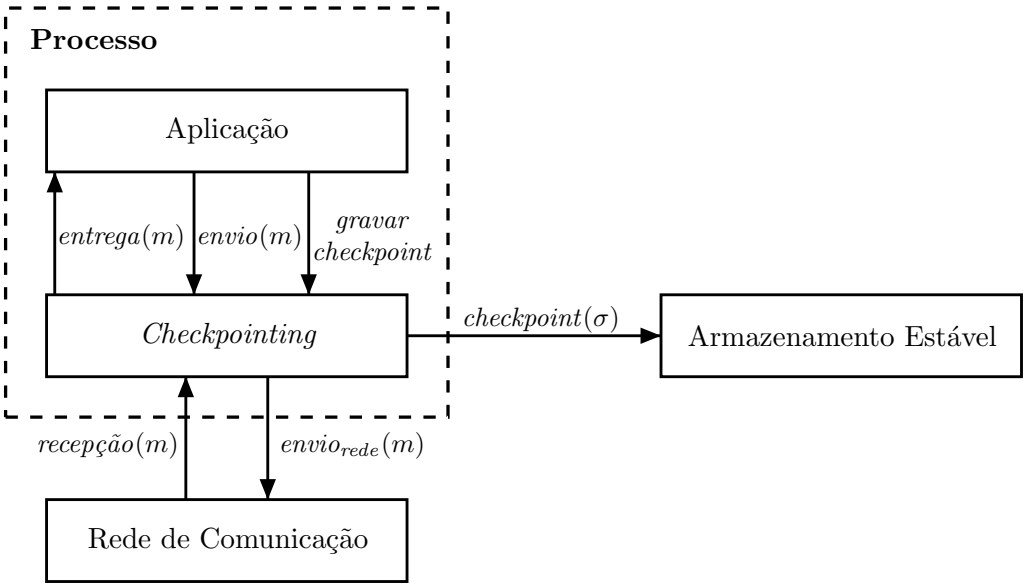

Figura 2.6: Estrutura de um processo.

[3]Em inglês: *deadlock*.
[4]Em inglês: *starvation*.

Em nosso modelo, um processo é um sistema estruturado em dois sub-sistemas: aplicação e *checkpointing*. O primeiro executa as tarefas da aplicação enquanto o segundo intercepta as mensagens recebidas e enviadas pelo primeiro e é responsável pela gravação de *checkpoints*, que pode ocorrer por requisição da aplicação ou do algoritmo de *checkpointing*. Desta maneira, o sub-sistema de *checkpointing* interage com dois outros sistemas externos ao processo: a rede de comunicação, para o envio e a recepção de mensagens através dos canais de comunicação; e o armazenamento estável, para efetuar a gravação dos *checkpoints* de forma que eles não sejam perdidos na ocorrência de uma falha. Todas as interações entre esses sistemas e sub-sistemas ocorrem na forma de eventos, apresentados na Figura 2.6.

Uma mensagem m é *enviada* quando o evento $envio(m)$ é executado. De forma semelhante, ela é *recebida* quando o evento $recepção(m)$ é executado e *entregue* após a execução do evento $entrega(m)$. Portanto, uma mensagem m de p_i para p_j está associada aos eventos $envio(m)$ e $envio_{rede}(m)$ em p_i e $recepção(m)$ e $entrega(m)$ em p_j; e tais eventos estão relacionados pela precedência causal da seguinte forma:

$$envio(m) \rightarrow envio_{rede}(m) \rightarrow recepção(m) \rightarrow entrega(m) \tag{2.4}$$

Um *checkpoint* é gravado quando o evento $checkpoint(\sigma)$ é executado pelo sub-sistema de *checkpointing*, no qual σ representa o estado corrente do processo que deve ser armazenado. Isto ocorre quando o sub-sistema de aplicação executa o evento *gravar checkpoint* ou, dependendo do protocolo de *checkpointing*, após o envio ou a recepção de uma mensagem. Em ambos os casos, existe uma dependência causal entre o evento anterior e o evento $checkpoint(\sigma)$. Especificamente, se um *checkpoint* é gravado após a recepção de uma mensagem m, temos a seguinte precedência causal entre os eventos:

$$envio(m) \rightarrow envio_{rede}(m) \rightarrow recepção(m) \rightarrow checkpoint(\sigma) \rightarrow entrega(m) \tag{2.5}$$

No entanto, como o evento $recepção(m)$ não influencia a execução da aplicação e a entrega da mensagem só ocorre após a gravação do *checkpoint*, podemos assumir que $envio(m) \not\rightarrow checkpoint(\sigma)$. Isto está de acordo com a Definição 2.1, que leva em consideração a entrega de uma mensagem ao invés da sua recepção para criação de uma dependência causal entre os eventos da aplicação. Esta estratégia é utilizada por vários protocolos de *checkpointing*, conforme apresentamos no Capítulo 3.

A inserção do sub-sistema de *checkpointing* entre a aplicação e a rede de comunicação fornece a autonomia e a liberdade necessárias pelos protocolos para a seleção de *checkpoints*. Desta maneira, este sub-sistema pode enviar e receber mensagens de controle, inserir *timestamps* nas mensagens ou bloquear a comunicação de um processo temporariamente, tudo isso de forma totalmente transparente para a aplicação. Além disso, o algoritmo de *checkpointing* pode decidir gravar um *checkpoint* ao receber uma mensagem de controle, depois do envio de uma mensagem da aplicação ou antes de sua entrega.

2.3.2 Padrão de *Checkpoints* e Mensagens

Um *checkpoint* gravado em meio estável é chamado de *checkpoint* estável. O γ-ésimo checkpoint estável gravado pelo processo p_i é representado por s_i^γ. Todo processo p_i começa sua execução pela gravação de um *checkpoint* estável s_i^0. Isto garante a existência de um estado global recuperável, uma vez que o conjunto $\{s_0^0, \ldots, s_{n-1}^0\}$ representa o estado inicial da aplicação distribuída. Tão ou mais importante que este primeiro *checkpoint* é o último *checkpoint* estável de cada processo. Conforme será visto no Capítulo 4, ele é a base para a determinação da linha de recuperação. Pela sua importância, representamos o índice do último *checkpoint* estável do processo p_i por $last\_s(i)$ e o *checkpoint* $s_i^{last\_s(i)}$ por s_i^{last}.

Entretanto, em recuperação por retrocesso nem todos os *checkpoints* são estáveis. Em especial, quando apenas um subconjunto dos processos falha, podemos utilizar o estado corrente dos outros processos como um *checkpoint* passível de ser recuperado, aqui chamado de *checkpoint* volátil. Caso a linha de recuperação calculada para o retrocesso contenha um desses *checkpoints*, o processo em questão não precisa retroceder e pode continuar a sua execução do seu estado corrente, inclusive sem precisar gravar um novo *checkpoint* estável. O *checkpoint* volátil de um processo p_i é representado por v_i. Para simplificar o uso de *checkpoints* estáveis e voláteis nesta dissertação, definimos c_i^γ como um *checkpoint geral* (ou simplesmente *checkpoint*) de acordo com a seguinte regra:

$$c_i^\gamma = \begin{cases} s_i^\gamma & , \quad \gamma \leq last\_s(i); \\ v_i & , \quad \gamma = last\_s(i) + 1. \end{cases} \tag{2.6}$$

O conjunto de todos os *checkpoints* da computação distribuída e suas inter-dependências causais, determinadas pela execução seqüencial dos processos e pela troca de mensagens, forma um *padrão de checkpoints e mensagens*[5] (CCP). Um CCP facilita a visualização de propriedades relativas ao conjunto de *checkpoints* e mensagens representados, sendo um recurso amplamente utilizado em provas e demonstrações relativas a *checkpointing*. Várias são as formas de se representar um CCP, mas a mais comum é por meio de diagramas de espaço-tempo que evidenciem os eventos de gravação de *checkpoints*, entrega e envio de mensagens. A Figura 2.7 mostra um exemplo desta forma de representação de um CCP. Nela podemos identificar os eventos de gravação de *checkpoints* estáveis (quadrados preenchidos), envio e entrega de mensagens (setas); e a representação dos estados correntes dos processos como *checkpoints* voláteis (triângulos preenchidos).

Um *intervalo entre checkpoints* I_i^γ representa o conjunto dos eventos ocorridos em um processo p_i entre os *checkpoints* $c_i^{\gamma-1}$ e c_i^γ, incluindo $c_i^{\gamma-1}$ mas excluindo c_i^γ. Na Figura 2.7, o intervalo I_0^2 é apresentado como um exemplo. A importância dos intervalos entre *checkpoints* vem do fato de que é a ocorrência de eventos de entrega e envio de mensagens em um mesmo intervalo que gera dependências entre *checkpoints* de processos diferentes, conforme será visto nas seções seguintes.

[5]Em inglês: *Checkpoint and Communication Pattern*.

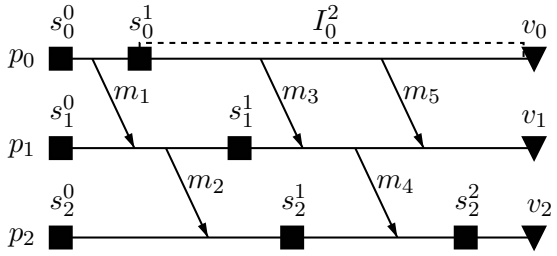

Figura 2.7: Padrão de *checkpoints* e mensagens.

2.3.3 *Checkpoints* Globais Consistentes

Se tomarmos um *checkpoint* de cada processo diferente, formaremos um conjunto de estados dos processos que representam um estado global da computação, chamado de *checkpoint global*. No entanto, assim como ocorre com cortes da computação distribuída, nem todo *checkpoint* global representa um estado consistente da computação. Quando dois *checkpoints* estão relacionados causalmente, eles não podem formar um *checkpoint* global que represente um estado consistente da computação distribuída. Portanto, um *checkpoint* global será consistente quando incluir um *checkpoint* de cada processo da aplicação de forma a não existirem dois *checkpoints* c_a^α e c_b^β tais que $c_a^\alpha \rightarrow c_b^\beta$. Na Figura 2.7, $\{v_0, s_1^1, s_2^1\}$ é um *checkpoint* global consistente enquanto que $\{s_0^0, s_1^1, s_2^1\}$ é inconsistente, uma vez que o corte definido pelo *checkpoint* global inclui o evento $entrega(m_1)$ sem incluir o evento $envio(m_1)$.

Teorema 2.1 *Um* checkpoint *global C representa um estado consistente da computação distribuída se, e somente se,*

$$\forall c_a^\alpha, c_b^\beta \in C : c_a^\alpha \not\rightarrow c_b^\beta$$

Prova: *Suficiência*($\Leftarrow$): Seja C um *checkpoint* global tal que $\forall c_a^\alpha, c_b^\beta \in C : c_a^\alpha \not\rightarrow c_b^\beta$. Suponha, por contradição, que o corte $\mathcal{C}$, definido por C, é inconsistente. Pela Definição 2.2, temos que existem dois eventos $e \in \mathcal{C}$, $e' \notin \mathcal{C}$ tais que $e' \rightarrow e$. Chamemos c_a^α o *checkpoint* em C referente ao processo do evento e' e c_b^β o *checkpoint* em C referente ao processo do evento e. Como e' não pertence ao corte definido por C, temos que $c_a^\alpha \rightarrow e'$. De forma análoga, como e pertence ao corte, ou $e = c_b^\beta$, ou $e \rightarrow c_b^\beta$. Em ambos os casos, pela definição de precedência causal, temos que $c_a^\alpha \rightarrow c_b^\beta$, o que contradiz nossa consideração inicial.

Necessidade($\Rightarrow$): Seja C um *checkpoint* global consistente. Suponha, por contradição, que existam dois *checkpoints* c_a^α e c_b^β em C tais que $c_a^\alpha \rightarrow c_b^\beta$. Como $a \neq b$, existe uma mensagem m cujo evento de envio ocorre no processo p_a após a gravação do *checkpoint* c_a^α e cuja entrega precede causalmente a gravação do *checkpoint* c_b^β, ou seja,

$$c_a^\alpha \rightarrow envio(m) \rightarrow entrega(m) \rightarrow c_b^\beta.$$

Sendo assim, um *checkpoint* global incluindo c_a^α e c_b^β não seria consistente pois o corte representado não conteria o evento $envio(m)$, o que viola a Definição 2.2. □

Podemos definir algumas operações sobre *checkpoints* globais. Neste contexto, duas operações importantes são a sua união e interseção. A união de dois *checkpoints* globais é dada pelo *checkpoint* mais recente, em cada processo, que está presente em um dos dois *checkpoints* globais (Figura 2.8). De forma semelhante, a interseção de dois *checkpoints* globais é dada pelo *checkpoint* menos recente em cada processo (Figura 2.8).

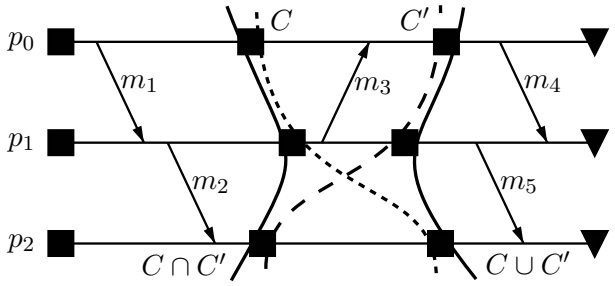

Figura 2.8: União e interseção de *checkpoints* globais.

Definição 2.4 ***União de*** **Checkpoints** ***Globais*** — *A união de dois* checkpoints *globais, C e C' é dada por:*

$$C \cup C' = \bigcup_{i=0}^{n-1} \left\{ c_i^{max(\alpha,\beta)} \mid c_i^{\alpha} \in C \ \wedge \ c_i^{\beta} \in C' \right\}$$

Definição 2.5 ***Interseção de*** **Checkpoints** ***Globais*** — *A interseção de dois* checkpoints *globais, C e C' é dada por:*

$$C \cap C' = \bigcup_{i=0}^{n-1} \left\{ c_i^{min(\alpha,\beta)} \mid c_i^{\alpha} \in C \ \wedge \ c_i^{\beta} \in C' \right\}$$

Uma propriedade interessante de *checkpoints* globais consistentes é o fechamento com relação às operações de união e interseção. A união ou a interseção de dois *checkpoints* globais consistentes resultará sempre em outro *checkpoint* global consistente.

Teorema 2.2 *A união de dois* checkpoints *globais consistentes C e C' é um* checkpoint *global consistente.*

Prova: Suponha, por contradição, que $C \cup C'$ é inconsistente. Portanto, devem existir em $C \cup C'$ pelo menos dois *checkpoints* c_a^{α} e c_b^{β} tais que $c_a^{\alpha} \rightarrow c_b^{\beta}$. Devido a esta dependência causal, sabemos que esses dois *checkpoints* não podem pertencer ao mesmo *checkpoint* global consistente original (C ou C'). Dessa forma, suponha, sem perda de generalidade, que $c_a^{\alpha} \in C$ e $c_b^{\beta} \in C'$; e seja c_a^{ι} a componente do processo p_a em C'. Sendo assim, pela Definição 2.4, $\iota \leq \alpha$ e $c_a^{\iota} \rightarrow c_b^{\beta}$, o que contradiz nossa consideração inicial de que C' é consistente. □

Teorema 2.3 *A interseção de dois* checkpoints *globais consistentes C e C' é um* checkpoint *global consistente.*

Prova: Suponha, por contradição, que $C \cap C'$ é inconsistente. Portanto, devem existir em $C \cap C'$ pelo menos dois *checkpoints* c_a^α e c_b^β tais que $c_a^\alpha \rightarrow c_b^\beta$. Devido a esta dependência causal, sabemos que esses dois *checkpoints* não podem pertencer ao mesmo *checkpoint* global consistente original (C ou C'). Dessa forma, suponha, sem perda de generalidade, que $c_a^\alpha \in C$ e $c_b^\beta \in C'$; e seja c_b^ι a componente do processo p_b em C. Sendo assim, pela Definição 2.5, $\iota \geq \beta$ e $c_a^\alpha \rightarrow c_b^\iota$, o que contradiz nossa consideração inicial de que C é consistente. □

As dependências entre *checkpoints* decorrentes das mensagens trocadas pelos processos dificultam a recuperação por retrocesso de estado. Uma vez que um ou mais processos falhem, estas dependências podem forçar alguns dos processos que não falharam a retroceder para um *checkpoint* anterior. Este procedimento é comumente chamado de *propagação de retrocesso*[6]. Para entender como este mecanismo funciona, considere um cenário onde um processo p_0 envia uma mensagem m para um processo p_1 e esta mensagem é entregue corretamente. Se p_0 não gravar nenhum *checkpoint* após o envio de m e falhar, ele será obrigado a retroceder a um estado anterior ao envio de m. Neste caso, como consideramos uma execução não-determinística dos processos, não se tem garantia de que p_0 executará novamente o evento de envio de m, o que invalidaria o estado em que se encontra o processo p_1. Para resolver este problema é necessário que p_1 também retroceda, voltando a um estado anterior à entrega de m, mesmo que tenha gravado algum *checkpoint* após este evento.

Se não houver nenhuma forma de sincronização entre os processos para a seleção de *checkpoints*, pode-se gerar alguns cenários onde a propagação de retrocesso retrocede a computação para o seu estado inicial mesmo após a gravação de vários *checkpoints* nos diferentes processos, uma anomalia conhecida como efeito dominó [29]. A Figura 2.9 apresenta um exemplo simples de cenário onde este problema ocorre. Inicialmente, o processo p_0 sofre uma falha e tem que ser retrocedido a um *checkpoint* estável, a princípio s_0^2. No entanto, a mensagem m_4 cria uma dependência entre s_0^2 e v_1, de forma que p_1 também deve retroceder. Porém, a mensagem m_3 cria uma dependência entre s_1^1 e s_0^2 de forma que se p_1 retroceder para s_1^1, p_0 deverá retroceder para s_0^1. De forma semelhante, a mensagem m_2 faz com que p_1 retroceda para s_1^0 e, finalmente, a mensagem m_1 força p_0 a retroceder ao *checkpoint* s_0^0. Portanto, neste cenário, a computação retrocede ao seu estado inicial de execução.

O efeito dominó existe devido a uma outra relação de dependência entre *checkpoints* que estende a precedência causal e determina a possibilidade de um *checkpoint* ou conjunto de *checkpoints* fazerem parte de algum *checkpoint* global consistente. As seções seguintes abordam esta relação em maiores detalhes.

[6]Em inglês: *rollback propagation.*

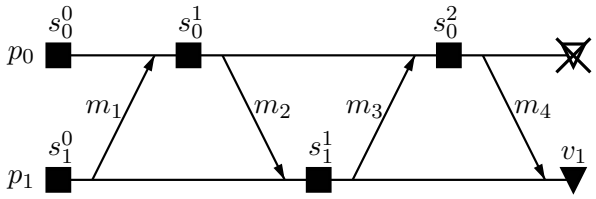

Figura 2.9: Propagação de retrocesso e efeito dominó.

2.3.4 Caminhos em Ziguezague

Apesar de necessária, a ausência de precedência causal entre dois *checkpoints* não é condição suficiente para que eles possam fazer parte de um mesmo *checkpoint* global consistente. Por exemplo, no CCP apresentado na Figura 2.10, os *checkpoints* s_0^1 e s_2^1 não estão relacionados causalmente mas é impossível formar um *checkpoint* global consistente contendo os dois.

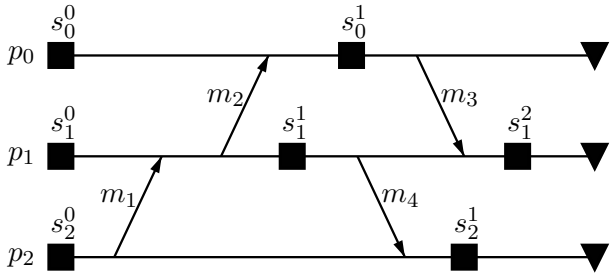

Figura 2.10: Exemplos de caminhos em ziguezague.

A especificação de uma condição necessária e suficiente para a existência de um *checkpoint* global consistente contendo um conjunto de *checkpoints* é devida a Netzer e Xu [28]. Esta condição é descrita a partir de seqüências de mensagens denominadas *caminhos em ziguezague*. Um conjunto S de *checkpoints* de processos diferentes pode fazer parte de algum *checkpoint* global consistente se, e somente se, não existir um caminho em ziguezague entre quaisquer dois *checkpoints* pertencentes ao conjunto S.

Definição 2.6 ***Caminho em Ziguezague*** — *Uma seqüência de mensagens* $[m_1, \ldots, m_k]$ *é um caminho em ziguezague que conecta* c_a^α *a* c_b^β *se, e somente se,*

(i) p_a *envia* m_1 *depois de* c_a^α*;*

(ii) se $m_i, 1 \leq i < k$*, é entregue ao processo* p_c*, então* m_{i+1} *é enviada por* p_c *no mesmo intervalo entre* checkpoints *ou em um intervalo posterior; e*

(iii) m_k *é entregue a* p_b *antes de* c_b^β*.*

Dois tipos de caminhos em ziguezague podem ser identificados: caminhos causais, chamados de *caminhos-C*, e caminhos não-causais, chamados de *caminhos-Z* [5]. Um caminho em ziguezague é causal se a entrega de todas as mensagens, com exceção da última, precede causalmente o evento de envio da próxima mensagem na seqüência. Na Figura 2.10, $[m_1, m_2]$ e $[m_2, m_3]$ são exemplos de caminhos-C enquanto $[m_3, m_4]$ e $[m_4, m_1]$ são exemplos de caminhos-Z. Analisando $[m_3, m_4]$, vemos que m_4 é enviada por p_1 antes de m_3 ser entregue. Porém, a definição de caminho em ziguezague é satisfeita pelo fato da entrega de m_3 e o envio de m_4 ocorrerem no mesmo intervalo entre *checkpoints*.

Como os caminhos-Z não obedecem a relação de precedência causal, pode existir um caminho-Z que conecte um *checkpoint* c_i^γ a si próprio, como o caminho $[m_4, m_1]$ na Figura 2.10. Este tipo especial de caminho-Z é chamado de *ciclo-Z* e impede que o *checkpoint* envolvido tome parte em qualquer *checkpoint* global consistente. Por causa disto, um *checkpoint* envolvido em um ciclo-Z é chamado de *checkpoint inútil*. É a existência de *checkpoints* inúteis que ocasiona o efeito dominó. Portanto, para evitar este problema, deve-se impedir a formação de ciclos-Z no padrão de *checkpoints* e mensagens da computação.

2.3.5 Relação de Z-Precedência entre *Checkpoints*

Se incorporarmos a relação de precedência causal à relação de dependência criada por caminhos em ziguezague podemos construir uma relação de dependência entre *checkpoints* baseada apenas na noção de causalidade [14, 25, 27]. Desta forma temos uma idéia mais geral sobre as dependências entre os *checkpoints*, abstraindo-se das trocas de mensagens. A relação resultante recebe o nome de *Z-precedência* [16].

Definição 2.7 ***Z-precedência*** — *Um* checkpoint c_a^α *Z-precede um* checkpoint c_b^β ($c_a^\alpha \rightsquigarrow c_b^\beta$) *se, e somente se,*

(i) $c_a^\alpha \rightarrow c_b^\beta$, *ou*

(ii) $\exists c_i^\gamma \mid (c_a^\alpha \rightsquigarrow c_i^\gamma) \wedge (c_i^{\gamma-1} \rightsquigarrow c_b^\beta)$.

Apesar da relação de Z-precedência estar definida para um par de *checkpoints*, podemos estender sua definição para conjuntos de *checkpoints* de forma a aumentar a clareza e facilitar a compreensão dos teoremas e provas presentes nesta dissertação.

Definição 2.8 *Seja c um* checkpoint *e sejam R e S dois conjuntos de* checkpoints. *Definimos as seguintes relações:*

- $c \rightsquigarrow S \iff \exists\, c' \in S \mid c \rightsquigarrow c'$;
- $S \rightsquigarrow c \iff \exists\, c' \in S \mid c' \rightsquigarrow c$;
- $S \rightsquigarrow R \iff \exists\, c \in S, c' \in R \mid c \rightsquigarrow c'$;
- $c \not\rightsquigarrow S \Leftrightarrow \neg(c \rightsquigarrow S)$; $S \not\rightsquigarrow c \Leftrightarrow \neg(S \rightsquigarrow c)$; $S \not\rightsquigarrow R \Leftrightarrow \neg(S \rightsquigarrow R)$.

Conforme podemos facilmente constatar, as mesmas propriedades relativas à consistência entre *checkpoints* existentes em caminhos em ziguezague também valem para a relação de Z-precedência. Sendo assim, um conjunto S de *checkpoints* é subconjunto de um *checkpoint* global consistente se, e somente se, $S \not\rightsquigarrow S$; e, portanto, um *checkpoint* c_i^γ é inútil quando $c_i^\gamma \rightsquigarrow c_i^\gamma$ [14, 16, 25].

Teorema 2.4 *Um conjunto de* checkpoints S *pode fazer parte de um mesmo* checkpoint *global consistente se, e somente se,* $S \not\rightsquigarrow S$.

Prova: *Suficiência*($\Leftarrow$): Considere um conjunto S que satisfaça a condição acima, e seja P_S o conjunto de processos que possuem *checkpoints* em S. Formaremos um *checkpoint* global C adicionando a S *checkpoints* dos processos que não estão em P_S da seguinte forma:

$$C = S \cup \bigcup_{p_i \notin P_S} \{c_i^{min(\gamma)} \mid c_i^\gamma \not\rightsquigarrow S\}$$

Note que esta regra de construção está bem definida pois um *checkpoint* volátil é sempre o último evento de cada processo e, portanto, não pode Z-preceder nenhum outro *checkpoint*. Então, pelo critério de minimização de γ, exatamente um *checkpoint* de cada processo não pertencente a P_S é escolhido para compor o conjunto C. Agora vamos supor, por contradição, que C é inconsistente. Neste caso, pelo Teorema 2.1, existem dois *checkpoints* c_a^α e c_b^β em C tais que $c_a^\alpha \rightarrow c_b^\beta$. Existem quatro possibilidades para a origem desses *checkpoints*:

- $c_a^\alpha \in S$, $c_b^\beta \in S$: Pela Definição 2.7, $c_a^\alpha \rightarrow c_b^\beta \Rightarrow c_a^\alpha \rightsquigarrow c_b^\beta$, o que contradiz nossa de hipótese de que $S \not\rightsquigarrow S$.
- $c_a^\alpha \notin S$, $c_b^\beta \in S$: $c_a^\alpha \rightsquigarrow c_b^\beta$ contraria as regras de formação do conjunto C.
- $c_a^\alpha \in S$, $c_b^\beta \notin S$: Como $c_a^\alpha \rightarrow c_b^\beta$, temos que $\beta > 0$, pois o primeiro *checkpoint* de cada processo é o seu primeiro evento e, portanto, não pode ser precedido por nenhum outro *checkpoint*. Então, pela regra de construção do conjunto C, existe um *checkpoint* $c_i^\gamma \in S$ tal que $c_b^{\beta-1} \rightsquigarrow c_i^\gamma$. Como $(c_a^\alpha \rightsquigarrow c_b^\beta) \wedge (c_b^{\beta-1} \rightsquigarrow c_i^\gamma) \Rightarrow c_a^\alpha \rightsquigarrow c_i^\gamma$ e $c_a^\alpha, c_i^\gamma \in S$, temos mais uma vez uma contradição da nossa hipótese inicial de que $S \not\rightsquigarrow S$.
- $c_a^\alpha \notin S$, $c_b^\beta \notin S$: Como no caso anterior, $\beta > 0$ e existe um *checkpoint* $c_i^\gamma \in S$ tal que $c_b^{\beta-1} \rightsquigarrow c_i^\gamma$. Como $(c_a^\alpha \rightsquigarrow c_b^\beta) \wedge (c_b^{\beta-1} \rightsquigarrow c_i^\gamma) \Rightarrow c_a^\alpha \rightsquigarrow c_i^\gamma$, temos uma contradição com relação às regras de formação de C.

Necessidade($\Rightarrow$): Suponha que existem dois *checkpoints* $c_a^\alpha, c_b^\beta \in S$ tais que $c_a^\alpha \rightsquigarrow c_b^\beta$. Pela definição de Z-precedência, temos que tal relação pode ser quebrada em uma série de relações de precedência causal entre pares de *checkpoints*:

$$c_a^\alpha = c_{i0}^{\gamma 0} \quad (c_{i0}^{\gamma 0} \rightarrow c_{i1}^{\gamma 1}), (c_{i1}^{\gamma 1 - 1} \rightarrow c_{i2}^{\gamma 2}), \ldots, (c_{i_{p-1}}^{\gamma_{p-1}-1} \rightarrow c_{i_p}^{\gamma_p}) \quad c_{i_p}^{\gamma_p} = c_b^\beta$$

Provaremos a inconsistência entre c_a^α e c_b^β por indução no número p de precedências.

Hipótese: Se $c_a^\alpha \rightsquigarrow c_b^\beta$ por meio de uma seqüência de $p-1$ relações de precedência causal entre pares, c_a^α e c_b^β não podem fazer parte de um mesmo *checkpoint* global consistente.

Base da Indução: $(p = 1)$ Temos a única relação $c_a^\alpha \rightarrow c_b^\beta$, o que impede, pelo Teorema 2.1, que eles façam parte do mesmo *checkpoint* global consistente.

Passo de Indução: $(p > 1)$ Podemos dividir a Z-precedência $c_a^\alpha \rightsquigarrow c_b^\beta$ em uma Z-precedência $c_a^\alpha \rightsquigarrow c_i^\gamma$ composta por $p-1$ relações de precedência causal e uma relação causal $c_i^{\gamma-1} \rightarrow c_b^\beta$. Pela hipótese de indução, temos que c_a^α não pode pertencer a um mesmo *checkpoint* global consistente que c_i^γ e qualquer *checkpoint* posterior a este em p_i. Além disso, como $c_i^{\gamma-1} \rightarrow c_b^\beta$, c_b^β não pode pertencer a um mesmo *checkpoint* global consistente que $c_i^{\gamma-1}$ ou qualquer *checkpoint* anterior a ele em p_i. Como todos os *checkpoints* em p_i são inconsistentes ou com c_a^α, ou com c_b^β, é impossível formar um *checkpoint* global consistente com ambos. □

Se, no Teorema 2.4, fizermos $S = \{c_i^\gamma\}$, obtemos a caracterização de um *checkpoint* útil.

Corolário 2.1 *Um* checkpoint c_i^γ *pode fazer parte de um* checkpoint *global consistente se, e somente se,* $c_i^\gamma \not\rightsquigarrow c_i^\gamma$.

A relação de Z-precedência não fornece apenas uma condição necessária e suficiente para a construção de *checkpoints* globais consistentes. Mais do que isso, ela estabelece uma relação de dependência entre os *checkpoints* de forma que, se $c_a^\alpha \rightsquigarrow c_b^\beta$, então c_a^α deve ser observado antes de c_b^β, o que significa que todo *checkpoint* global consistente que contém c_a^α, contém um *checkpoint* do processo p_b anterior a c_b^β [16]. Dessa forma nenhum *checkpoint* global consistente contendo c_a^α pode estar no futuro de um *checkpoint* global consistente contendo c_b^β e um monitor externo da computação [11, 12, 15] observará c_a^α antes de c_b^β.

2.3.6 Protocolos para *Checkpointing*

Por questões de autonomia, a própria aplicação deve poder selecionar estados a serem gravados como *checkpoints*, chamados de *checkpoints básicos*. No entanto, sabemos que se nada mais for feito, estes *checkpoints* podem se tornar inúteis e ocasionar o efeito dominó. Isto é evitado por meio de algoritmos que impedem a formação de ciclos-Z no padrão de *checkpoints* e mensagens da computação distribuída. Estes algoritmos são implementados no sub-sistema de *checkpointing* de um processo (Figura 2.6) e se baseiam no envio de mensagens de controle, na propagação de *timestamps* nas mensagens da própria aplicação e na gravação de *checkpoints forçados*, que não são escolhidos pela aplicação mas servem para garantir o avanço da linha de recuperação e a ausência do efeito dominó.

Existem vários tipos de protocolos para *checkpointing*, cada um com suas características positivas e negativas. Eles são classificados de acordo com o grau de autonomia oferecido à

aplicação e a impossibilidade de ocorrência do efeito dominó. Podemos identificar três classes distintas de protocolos: assíncronos, síncronos e quase-síncronos [27]. Na literatura voltada a tolerância a falhas por retrocesso de estado, também encontramos estas mesmas classes com os nomes de protocolos não-coordenados, coordenados e induzidos pela comunicação[7], respectivamente [13].

A abordagem assíncrona oferece o máximo de autonomia possível aos processos que compõem a aplicação com relação à seleção de *checkpoints*. Neste tipo de protocolo, os processos gravam *checkpoints* independentemente, sem qualquer forma de sincronização [9]. Porém, conforme já comentamos, este tipo de comportamento pode levar à ocorrência do efeito dominó. Protocolos síncronos resolvem este problema fazendo com que, sempre que um processo grave um *checkpoint* básico, todos os processos da aplicação se sincronizem e construam um novo *checkpoint* global consistente. No entanto, devido à necessidade de sincronização explícita existente neste método, tais protocolos possuem um custo extra de comunicação devido a mensagens de controle, podem suspender os processos envolvidos durante o período de sincronização e diminuem a autonomia dos processos por exigir um grande número de *checkpoints* forçados.

A abordagem quase-síncrona aparece como um compromisso entre as abordagens síncrona e assíncrona. Nela, os processos ficam livres para gravarem seus *checkpoints* básicos e, eventualmente, são obrigados a gravar *checkpoints* forçados para evitar a ocorrência do efeito dominó, de acordo com o algoritmo empregado. A decisão de quando forçar a gravação de *checkpoints* é baseada em informações de controle que são propagadas como *timestamps* nas mensagens da própria aplicação. Sendo assim, tal classe de protocolos não possui custo extra pelo envio de mensagens de controle ou suspensão dos processos da aplicação.

Falaremos sobre os protocolos para *checkpointing* em maiores detalhes no Capítulo 3.

2.4 Recuperação de Falhas por Retrocesso de Estado

O mecanismo de recuperação é acionado quando um conjunto de processos falhos é identificado. Sua principal função é determinar, por meio deste conjunto e das dependências existentes entre os *checkpoints*, qual será a linha de recuperação a ser utilizada no retrocesso. Uma vez que isto tenha sido determinado, resta coordenar o trabalho de recuperação, de forma a garantir que todos os processos retrocedam de maneira consistente para que a computação possa continuar a sua execução.

[7] Em inglês: *communication-induced*.

2.4.1 Captura das Dependências entre os *Checkpoints*

Conforme vimos na Seção 2.3.3, a relação de precedência causal cria dependências entre os *checkpoints* que impedem que eles façam parte de um mesmo *checkpoint* global consistente. Então, em caso de falha, é necessário capturar estas dependências para poder determinar a linha de recuperação. Existem duas formas de se capturar as dependências entre os *checkpoints* de uma computação distribuída: direta e transitiva.

Na captura por dependências diretas, o índice do intervalo entre *checkpoints* corrente de um processo é anexado a cada mensagem enviada na forma de um *timestamp*. Além disso, durante um intervalo entre *checkpoints*, um processo armazena todas as dependências recebidas e as grava juntamente com o próximo *checkpoint* armazenado [9]. Desta forma, durante o processo de recuperação, esta informação pode ser utilizada para construir um CCP completo da computação distribuída, montado pela união das dependências diretas armazenadas juntamente com os *checkpoints* de todos os processos.

A captura por dependências transitivas utiliza um esquema muito semelhante aos relógios vetoriais vistos na Seção 2.2.2, chamados vetores de dependências[8] (DV) [32, 36]. Cada processo mantém um vetor DV com n elementos, onde a i-ésima entrada representa o valor do último intervalo entre *checkpoints* do processo p_i do qual o evento corrente depende causalmente. Este vetor é propagado juntamente com as mensagens enviadas e é incorporado ao vetor do processo receptor da mensagem que o atualiza tomando para cada entrada o valor máximo entre o vetor recebido na mensagem e o mantido pelo processo. Quando um *checkpoint* é gravado, o vetor corrente é gravado juntamente com ele, de forma a permitir que todas as suas dependências causais possam ser reconstruídas, mesmo na ocorrência de uma falha. Após a gravação de um *checkpoint* por um processo p_i, a sua entrada $DV[i]$ é incrementada para referenciar o novo intervalo entre *checkpoints* criado. Utilizamos $DV(c)$ para referenciar o vetor de dependências do *checkpoint* c e $DV(m)$ para referenciar o vetor de dependências enviado como *timestamp* na mensagem m. A Figura 2.11 ilustra o uso de vetores de dependências, identificando os vetores gravados com os *checkpoints* estáveis e propagados nas mensagens da aplicação.

Os vetores de dependências representam todo o histórico causal de um *checkpoint*. Desta maneira, podemos chegar às seguintes relações entre vetores de dependências e a precedência causal entre *checkpoints* em um padrão de *checkpoints* e mensagens:

$$c_a^\alpha \rightarrow c_b^\beta \iff \alpha < DV(c_b^\beta)[a] \tag{2.7}$$

$$c_a^\alpha \rightarrow c_b^\beta \iff DV(c_a^\alpha)[a] < DV(c_b^\beta)[a] \tag{2.8}$$

Apesar de representar um custo maior com relação aos *timestamps* das mensagens da aplicação e ao armazenamento e atualização de um vetor durante a execução dos processos, a captura por dependências transitivas permite a construção de algoritmos eficientes para a determinação da linha de recuperação e controle do retrocesso [13].

[8]Em inglês: *Dependency Vector*.

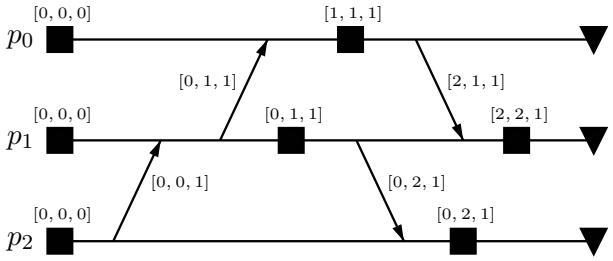

Figura 2.11: Propagação de vetores de dependências.

2.4.2 Mecanismo de Recuperação

Uma sessão de recuperação tem início quando uma falha é percebida. Neste ponto, o sistema de recuperação deve fazer com que a computação seja retrocedida para o *checkpoint* global consistente mais recente, tendo em vista as dependências existentes entre os *checkpoints*. Dependendo da forma como é implementado, o mecanismo de recuperação pode ser classificado como síncrono ou assíncrono [13].

Quando um esquema síncrono de recuperação é utilizado, um processo especial denominado gerente de recuperação é encarregado de controlar o retrocesso dos processos, garantindo a consistência do estado global recuperado e da continuidade da execução. O gerente de retrocesso pode ser implementado por meio de um processo comum da aplicação (como um processo falho já recuperado) utilizando um algoritmo de validação atômica (como o *two-phase commit* [8]) para garantir a atomicidade do retrocesso e impedir a interferência entre vários retrocessos concorrentes [9, 23]. Em uma primeira fase, o gerente de retrocesso obtém as dependências entre os *checkpoints* e avisa os demais processos para interromperem parcial ou totalmente sua execução de forma a não criar dependências novas. Com as dependências adquiridas, o gerente é capaz de determinar a linha de recuperação a ser utilizada pelos processos em função das falhas ocorridas. Em uma segunda fase, ele propaga a linha de recuperação e avisa para os processos efetuarem o retrocesso e recomeçarem sua execução.

Mecanismos de recuperação assíncronos se desenvolvem por meio de um efeito cascata entre os retrocessos [26, 32]. Um processo falho, ao ser reiniciado, envia uma mensagem aos demais processos avisando sobre seu retrocesso ao seu último *checkpoint* estável. Os processos que recebem este aviso e percebem uma dependência com relação a este retrocesso também retrocedem de forma a garantir a consistência do estado global da computação distribuída. Ao retrocederem, estes processos também enviam um aviso aos demais anunciando seu retorno de estado. Este processo continua até que um *checkpoint* global consistente seja recuperado. Nestes casos, a ausência de uma coordenação direta da recuperação e a possibilidade de falhas concorrentes ou falhas durante a sua execução torna os algoritmos mais complicados e, muitas vezes, menos eficientes.

Quando a aplicação distribuída é baseada em canais de comunicação confiáveis, a recuperação de um estado global consistente deve levar em conta as mensagens que se encontram em trânsito, isto é, as mensagens que tenham sido enviadas mas que ainda não tenham sido entregues em tal estado [13]. Ao recomeçar a execução da computação, tais mensagens devem ser reenviadas ou reaplicadas no processo receptor, garantindo assim que elas não sejam perdidas. No entanto, para se fazer isto é necessário gravar as mensagens em meio de armazenamento estável (na forma de *logs*) e promover o retrocesso da computação não apenas para um *checkpoint* global consistente, mas para um *distributed snapshot* consistente, que é um *checkpoint* global consistente onde as mensagens em trânsito estão devidamente armazenadas em meio estável e podem ser recriadas durante a execução [11, 20].

Protocolos síncronos para *checkpointing* podem ser facilmente estendidos para criar *distributed snapshots* consistentes [11, 30, 31]. Em protocolos assíncronos e quase-síncronos a situação é um pouco diferente devido à falta de coordenação explícita entre os processos, existindo poucos trabalhos que abordam este problema [20, 26]. Uma alternativa simples é garantir que todo *checkpoint* global consistente seja um *distributed snapshot* consistente. Isto pode ser realizado gravando-se em meio estável todas as mensagens propagadas pela aplicação. No caso de um retrocesso, todas as mensagens cujo evento de envio não tenha sido retrocedido e cujo evento de entrega não tenha ocorrido ou tenha sido retrocedido deverão ser reaplicadas ao processo receptor. Uma versão otimizada deste algoritmo foi proposta por Manivannan e Singhal [26].

2.4.3 Determinação da Linha de Recuperação

A linha de recuperação corresponde ao estado recuperável mais recente da computação distribuída. Em recuperação de falhas por retrocesso de estado baseada em *checkpointing*, é o *checkpoint* global consistente mais ao futuro dentro do padrão de *checkpoints* e mensagens gerado até o momento da identificação da falha. Obviamente, a linha de recuperação não poderá conter um *checkpoint* volátil de um processo falho pois o conteúdo de sua memória volátil, que corresponde ao seu estado de execução, é perdido. A determinação do *checkpoint* global consistente a ser utilizado é feita com base na quantidade de trabalho perdido, medida pelo número de *checkpoints* retrocedidos.

Definição 2.9 ***Custo de Retrocesso*** — *Dado um* checkpoint *global consistente para o qual a computação pode ser retrocedida após uma falha, seu custo de retrocesso corresponde ao número de* checkpoints *que estão após o estado global representado.*

Definição 2.10 ***Linha de Recuperação*** — *Dados um padrão de* checkpoints *e mensagens e um conjunto* $F \subseteq P$ *de processos falhos, a linha de recuperação* R_F *é o* checkpoint *global consistente que não inclui um* checkpoint *volátil de um processo falho e que minimiza o custo de retrocesso.*

O critério de minimização do custo de retrocesso existente na definição da linha de recuperação garante que ela é única para um determinado CCP e um conjunto F de processos falhos.

Teorema 2.5 *A linha de recuperação R_F criada por um conjunto $F \subseteq P$ de processos falhos em um padrão de* checkpoints *e mensagens é única.*

Prova: Suponha, por contradição, que existem duas linhas de recuperação diferentes R_F e R'_F para um mesmo conjunto F de processos falhos, ambas com o mesmo custo de retrocesso. Como R_F e R'_F são diferentes, deve existir pelo menos um processo p_k para o qual os *checkpoints* desses dois conjuntos diferem. Sendo assim, suponha sem perda de generalidade que $c_k^\alpha \in R_F$, $c_k^\beta \in R'_F$ e que $\alpha < \beta$. Desta forma, pela Definição 2.4, podemos concluir que o custo de retrocesso do *checkpoint* global consistente $R_F \cup R'_F$ é menor do que os custos de R_F e R'_F, o que é uma contradição pois, por definição, R_F e R'_F minimizam o custo de retrocesso. □

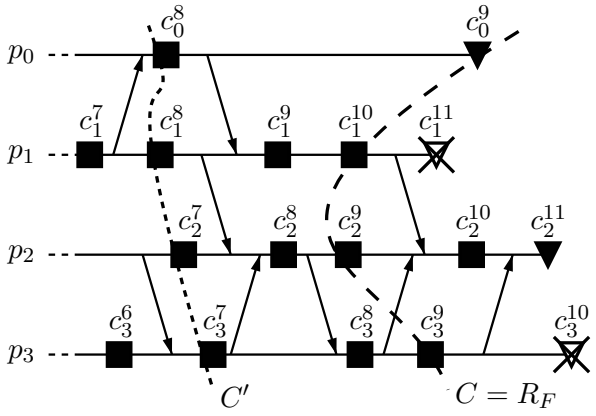

Figura 2.12: CCP após uma falha e sua linha de recuperação.

A Figura 2.12 apresenta um exemplo de padrão de *checkpoints* e mensagens identificado pelo sistema de recuperação depois de uma falha e antes de sua recuperação. A recuperação dar-se-á sobre este CCP, identificando-se a linha de recuperação a ele associada. Note que não é feita nenhuma consideração sobre os instantes de identificação das falhas e de captura dos estados voláteis dos processos não-falhos. Nossa única consideração é a ausência de dependências entre os *checkpoints* voláteis dos processos, o que pode ser facilmente obtido pela suspensão dos processos logo após a captura de seu estado volátil. Podemos identificar pelo menos dois *checkpoint* globais consistentes, C' e C, para os quais a computação poderia retroceder, com custos de retrocesso 11 e 4 respectivamente. Podemos facilmente verificar que C' não corresponde à linha de recuperação. Na verdade, $R_{\{p_1,p_3\}} = C$ neste CCP.

Uma forma simples de se determinar a linha de recuperação é por meio de uma estrutura de dados conhecida como grafo de dependências de retrocesso[9], ou simplesmente *R-graph* [9, 35, 36]. Este grafo apresenta um vértice para cada *checkpoint* existente no padrão de *checkpoints* e mensagens e uma aresta dirigida conecta c_a^α a c_b^β quando:

- $a \neq b$ e existe uma mensagem m enviada no intervalo I_a^α e entregue em I_b^β, ou
- $a = b$ e $\beta = \alpha + 1$.

Esta estrutura recebe este nome pelo fato de que a existência de uma aresta conectando c_a^α a c_b^β significa que o retrocesso do *checkpoint* c_a^α deve forçar o retrocesso do *checkpoint* c_b^β para não deixar a computação em um estado global inconsistente pela entrega de uma mensagem m cujo evento de envio foi retrocedido.

Para determinar a linha de recuperação a partir de um *R-graph* basta realizar uma busca no grafo, marcando todos os vértices para os quais existe um caminho a partir dos vértices que representam os *checkpoints* voláteis dos processos falhos, os quais são perdidos durante a falha e, necessariamente, precisam ser retrocedidos. Por definição, todos os vértices alcançáveis a partir deles no *R-graph* deverão ser retrocedidos também [9]. Depois desta operação, a união dos últimos *checkpoints* não marcados de cada processo corresponde à linha de recuperação da computação distribuída. Como exemplo, considere a Figura 2.13(a), que apresenta o *R-graph* referente ao CCP da Figura 2.12 com os *checkpoints* voláteis dos processos falhos inicialmente marcados. Após o procedimento de busca, obtemos o grafo da Figura 2.13(b), onde os vértices não preenchidos representam aqueles marcados durante a busca. Como resultado obtemos a linha de recuperação da computação R_F, que corresponde ao *checkpoint* global consistente C apresentado na Figura 2.12.

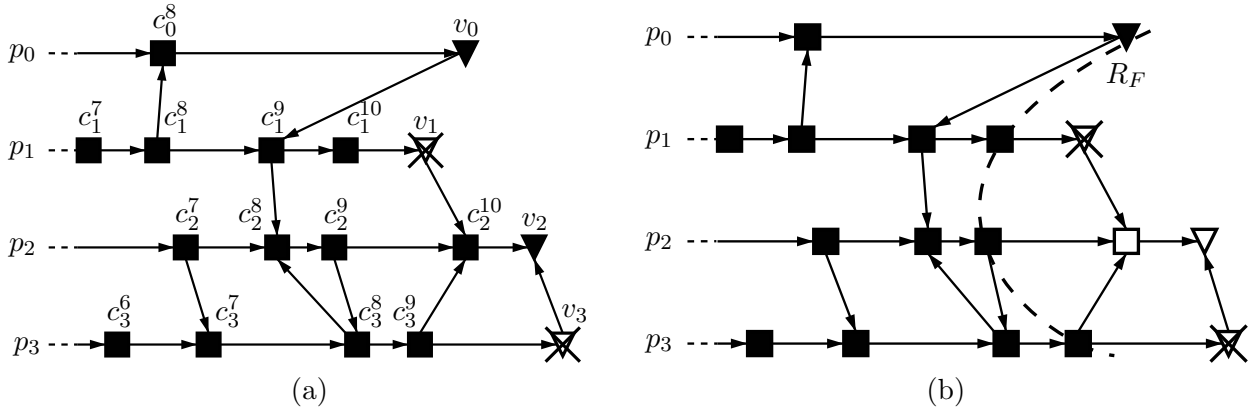

Figura 2.13: Determinação da linha de recuperação através de *R-graphs*.

[9]Em inglês: *rollback-dependency graph*.

Um *R-graph* pode ser construído pela união das dependências diretas entre os *checkpoints* armazenadas durante a execução, conforme foi visto na Seção 2.4.1. Porém, dependendo dos protocolos para *checkpointing* e recuperação utilizados, não é necessário construí-lo explicitamente [36]. Especificamente, quando utilizamos algoritmos de recuperação assíncrona, a busca no grafo é realizada de forma implícita pela propagação de retrocessos na forma de mensagens entre os processos avisando sobre as dependências existentes [32]. De qualquer forma, a importância de *R-graphs* vai além de sua utilização prática. Por fornecer um algoritmo determinístico para o cálculo da linha de recuperação em um CCP, esta estrutura tem uma grande importância teórica, podendo ser utilizada em diversos teoremas e provas sobre *checkpointing* [35, 36, 37, 38].

Capítulo 3

Protocolos para *Checkpointing*

Protocolos para *checkpointing* podem ser classificados como assíncronos, síncronos e quase-síncronos. A abordagem assíncrona oferece o máximo de autonomia na seleção de *checkpoints* pelos processos. Entretanto, alguns dos *checkpoints* selecionados podem ficar impossibilitados de pertencer a um *checkpoint* global consistente, perdendo totalmente sua utilidade. Nos protocolos síncronos, no entanto, os processos sincronizam seus *checkpoints* de forma a sempre construir um novo *checkpoint* global consistente da computação. Esta tarefa de sincronização, porém, envolve a troca de mensagens de controle e, muitas vezes, a suspensão da execução dos processos, o que pode prejudicar o desempenho de toda a aplicação. Protocolos quase-síncronos se apresentam como um híbrido das duas abordagens anteriores, reunindo suas principais vantagens: ausência de mensagens de controle, não suspensão dos processos e inexistência de *checkpoints* inúteis. Isto é conseguido pela quebra das Z-precedências não causais que possam gerar *checkpoints* inúteis. Quando tais Z-precedências são percebidas em um determinado intervalo entre *checkpoints*, um *checkpoint* forçado é gravado pelo protocolo.

Neste capítulo apresentamos as principais características e protocolos para as classes síncrona e quase-síncrona. Os protocolos assíncronos não são abordados pelo fato da ausência de coordenação entre os processos praticamente impedir a existência de variantes dentro desta abordagem. A diferença existente entre os protocolos desta classe reside apenas no modelo de computação e mecanismo de recuperação adotados [9, 22, 32].

3.1 Protocolos Síncronos

Em *checkpointing* síncrono, os processos se coordenam por meio de troca de mensagens para construir um *checkpoint* global consistente da computação sempre que um processo desejar gravar um *checkpoint* local. Esta estratégia simplifica o mecanismo de retrocesso e não sofre de efeito dominó, uma vez que os processos retrocedem, no máximo, ao último *checkpoint* global consistente gravado. Um algoritmo muito simples e prático consiste em bloquear a

comunicação entre os processos enquanto o protocolo executa [30, 31]. A Figura 3.1 apresenta um possível diagrama de execução para um algoritmo deste tipo. Um coordenador externo, que também pode ser implementado como um processo da aplicação, envia uma mensagem a todos os processos avisando-os para gravarem um *checkpoint*. Ao receberem esta mensagem, os processos gravam um *checkpoint* provisório, suspendem as comunicações com os demais processos e enviam uma mensagem de confirmação ao coordenador. Este, ao receber a confirmação de todos os processos, envia-os uma nova mensagem de finalização, avisando-os para transformarem seus *checkpoints* provisórios em permanentes e continuarem sua execução normal.

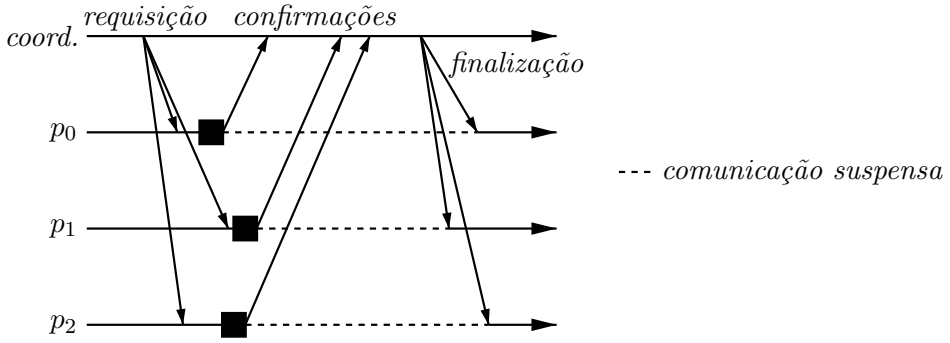

Figura 3.1: Exemplo de protocolo síncrono.

A suspensão da comunicação entre os processos durante a execução do protocolo garante a inexistência de dependências entre os *checkpoints* que estão sendo gravados, o que implica na criação de um *checkpoint* global consistente. No entanto, em muitos casos a própria execução do processo depende do envio ou recebimento de mensagens, o que pode resultar em sua suspensão completa até o protocolo terminar. Isto pode representar um atraso significativo na execução da computação, principalmente se a freqüência de seleção de *checkpoints* for alta, o que não é uma característica desejável em um protocolo para *checkpointing*. A solução para este problema está na utilização de protocolos não-bloqueantes.

Se não houvesse o bloqueio dos processos, um deles poderia enviar uma mensagem a outro após gravar o seu *checkpoint* de forma que esta mensagem pudesse ser entregue ao receptor antes dele gravar o seu *checkpoint*, o que tornaria os *checkpoints* dos dois processos inconsistentes um com o outro. Chandy e Lamport [11] resolveram este problema pela utilização de canais de comunicação FIFO confiáveis e o envio de mensagens de controle contendo uma marca especial por todos os canais de comunicação de um processo quando este grava um *checkpoint*. Dessa forma, sempre que um processo receber uma mensagem dessas proveniente de qualquer canal de comunicação, ele grava o seu *checkpoint*. As mensagens de controle seguintes, provenientes dos outros canais podem ser simplesmente desconsideradas

ou utilizadas para auxiliar na captura do estado dos canais de comunicação [11]. Esta estratégia garante que nenhuma mensagem enviada após a gravação de um *checkpoint* de um processo será entregue ao processo receptor antes da gravação do seu *checkpoint* em uma mesma instância de execução do protocolo. Sendo assim, um *checkpoint* global consistente pode ser construído sem que um processo tenha que ficar bloqueado esperando alguma confirmação ou resposta proveniente de algum outro processo.

O Algoritmo 3.1 apresenta o protocolo síncrono para *checkpointing* não-bloqueante de Chandy e Lamport. Um processo que deseja gravar um *checkpoint* deve fazê-lo e enviar a mensagem de controle com a marca a todos os outros. A variável *instância* é utilizada para que um processo atenda apenas a primeira mensagem de controle referente a cada instância. Como os canais são FIFO, sabe-se que se o valor recebido for maior que aquele mantido no processo, ele certamente terá apenas uma unidade a mais. A corretude deste algoritmo depende do fato dos canais serem FIFO e confiáveis. Se eles não tivessem esta característica, a simples perda ou a entrega fora de ordem de uma mensagem de controle poderia invalidar sua execução. Uma otimização possível para sobrepor esta restrição é enviar os dados da mensagem de controle como um *timestamp* presente em todas as mensagens enviadas pelo processo após a gravação do *checkpoint* [13]. No entanto, esta otimização pode gerar um cenário onde o índice da instância recebido seja duas ou mais unidades maior que o índice atual. Neste caso, o *checkpoint* gravado vale por todas as instâncias neste intervalo e o valor da instância atual passa a ser aquele recebido pela mensagem de controle. Esta técnica é muito semelhante ao protocolo quase-síncrono BCS que será visto na Seção 3.2.3.

Algoritmo 3.1 Protocolo síncrono de Chandy e Lamport em um processo p_i.

Estruturas de Dados

1: **Var**
2: *instância*: *integer*

Inicialização {inicialização realizada em todos os processos}

1: *instância* $\leftarrow 0$

Ao gravar *checkpoint*

1: *instância* $\leftarrow$ *instância* $+1$
2: *envia* (*CKPT*, *instância*) *para todos* {requisição para gravar *checkpoint*}

Ao receber (*CKPT*, *instância_req*)

1: **if** *instância_req* $>$ *instância* **then** {testa se *instância_req* $=$ *instância*$+1$}
2: grava *checkpoint* {evento anterior, o qual envia mensagens de controle}
3: **end if**
4: continua execução normal

Protocolos síncronos requerem a criação de um *checkpoint* global consistente em cada execução sua, o que pode exigir a coordenação de todos os processos e a gravação de novos *checkpoints* em cada um deles, como ocorre nos algoritmos que comentamos. Esta coordenação pode deixar lenta a execução de tais protocolos, principalmente se eles forem bloqueantes. É interessante reduzir o número de processos envolvidos em tal coordenação apenas àqueles que realmente precisam gravar um novo *checkpoint*, ou seja, os processos que se comunicaram direta ou indiretamente com o iniciador da nova instância desde a gravação de seu último *checkpoint*. O algoritmo proposto por Koo e Toueg atinge esta coordenação minimal em uma abordagem de duas fases recursiva, onde um processo coordena a ação daqueles que com ele se comunicaram diretamente no último intervalo entre *checkpoints* [23].

3.2 Protocolos Quase-síncronos

A coordenação necessária entre os processos é a maior desvantagem dos protocolos síncronos. Além disso, por questões de eficiência, várias implementações deste tipo de protocolo utilizam um processo coordenador, o qual decide de maneira unilateral quando o protocolo deve ser executado [30, 31]. Esta estratégia, no entanto, diminui ainda mais a autonomia dos processos quanto à seleção dos seus *checkpoints*. O ideal seria que cada processo pudesse escolher os estados que devem ser gravados em meio estável como é feito em protocolos para *checkpointing* assíncronos. Porém, sabemos que a seleção de *checkpoints* sem coordenação pode gerar *checkpoints* inúteis e levar ao efeito dominó. Na figura 3.2, por exemplo, apenas os *checkpoints* iniciais são úteis e qualquer falha faria a computação retroceder ao seu estado inicial. Protocolos quase-síncronos deixam os processos selecionarem seus *checkpoints* básicos de forma autônoma, mas podem obrigá-los a gravar *checkpoints* forçados na ocorrência de eventos como envio ou recepção de mensagens para evitar a formação de Z-precedências que levem ao efeito dominó.

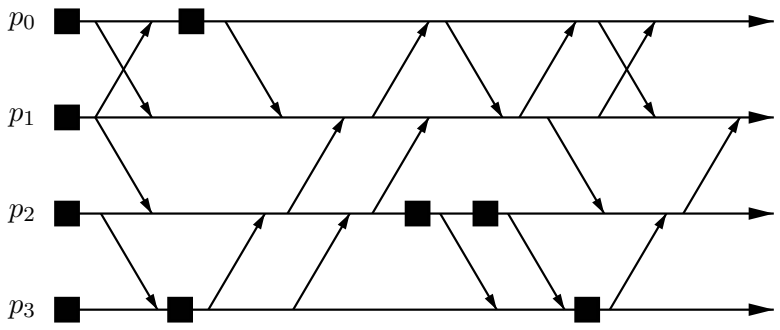

Figura 3.2: *Checkpoints* básicos.

Os protocolos quase-síncronos mais comuns enviam *timestamps* nas mensagens da própria aplicação para auxiliar na seleção dos *checkpoints* forçados de forma a minimizar o seu número. Dependendo dos *timestamps* e das condições utilizadas por estes algoritmos, os padrões de *checkpoints* e mensagens gerados podem ser mais ou menos restritivos com relação aos tipos de Z-precedência permitidos, podendo gerar condições mais fortes do que apenas a ausência de *checkpoints* inúteis [27]. De acordo com as restrições do padrão gerado, podemos identificar pelo menos quatro classes de protocolos quase-síncronos: SZPF(*Strictly Z-Path Free*), ZPF(*Z-Path Free*), ZCF(*Z-Cycle Free*) e PZCF(*Partially Z-Cycle Free*). A primeira classe é a mais restritiva e impede a ocorrência de qualquer caminho-Z. A segunda classe impede que existam Z-precedências não-causais entre os *checkpoints*. A classe ZCF é a menos restritiva dentro daquelas que garantem a ausência de *checkpoints* inúteis, uma vez que os protocolos da classe PZCF apenas realizam algum esforço na tentativa de quebrar os ciclos-Z existentes no padrão, mas não garantem a quebra de todos. A Figura 3.3 mostra o relacionamento entre estas classes de protocolos. Como podemos verificar, as menos restritivas englobam as mais restritivas de forma que as características de uma também são válidas na outra. As seções seguintes apresentam a definição, as características e exemplos de protocolos para cada uma das classes apresentadas.

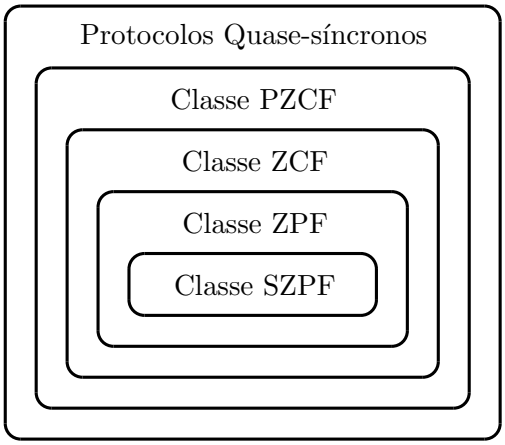

Figura 3.3: Classes de protocolos quase-síncronos.

3.2.1 Classe SZPF

Os protocolos da classe SZPF impedem a formação de qualquer caminho-Z no padrão de *checkpoints* e mensagens da computação. Os padrões gerados por estes protocolos também são chamados de SZPF e são definidos formalmente da seguinte forma:

Definição 3.1 ***Padrões SZPF*** — *Um padrão de* checkpoints *e mensagens é SZPF se, e somente se, ele não possui nenhum caminho-Z.*

Garantir esta propriedade não é nem um pouco complicado, uma vez que ela está diretamente relacionada com a ordem entre os eventos de envio e entrega de mensagens dentro de um intervalo entre *checkpoints*, conforme apresentamos no Teorema 3.1.

Teorema 3.1 *Um padrão de* checkpoints *e mensagens é SZPF se, e somente se, em cada intervalo entre* checkpoints*, todos os eventos de entrega de mensagens ocorrem antes de qualquer evento de envio de mensagem.*

Prova: *Suficiência*($\Leftarrow$): Considere um padrão onde todas as entregas de mensagens ocorrem antes dos envios de mensagens dentro de cada intervalo entre *checkpoints*. Suponha por contradição que este padrão não é SZPF, ou seja, ele possui pelo menos um caminho-Z. No entanto, um caminho-Z apresenta, por definição, pelo menos um par de mensagens m_i e m_{i+1} consecutivas dentro do caminho tais que $entrega(m_i) \not\rightarrow envio(m_{i+1})$, sendo que estes dois eventos ocorrem no mesmo intervalo entre *checkpoints*. Este fato, porém, é uma contradição com a nossa suposição inicial a respeito do padrão de *checkpoints* e mensagens.

Necessidade($\Rightarrow$): Considere um padrão de *checkpoints* e mensagens SZPF. Suponha por contradição que existe um intervalo entre *checkpoints* no qual a entrega de uma mensagem m_a ocorre após o envio de uma mensagem m_b. Neste caso $[m_a, m_b]$ forma um caminho-Z e, portanto, o padrão não pode ser SZPF. □

Em tais padrões, como todos os caminhos em ziguezague são caminhos-C, todas as Z-precedências existentes são causais. Este fato automaticamente impede a existência de *checkpoints* obsoletos uma vez que um evento não pode preceder causalmente a si próprio. Os protocolos desta classe costumam ser muito simples, uma vez que a decisão de forçar um *checkpoint* pode ser baseada apenas no regime de troca de mensagens e não nas relações de dependência entre os *checkpoints*. Por este motivo tais protocolos também são chamados de *baseados em modelo*[1] [13]. Apesar de serem fáceis de desenvolver, os protocolos desta classe apresentam um número excessivo de *checkpoints* forçados, necessários para quebrar todos os caminhos-Z existentes. Isto pode gerar um atraso considerável na execução da computação distribuída e degradar completamente o seu desempenho.

Um dos protocolos mais conhecidos desta classe é o NRAS (*No-Receive-After-Send*) [36], o qual força um *checkpoint* antes de entregar uma mensagem caso já tenha ocorrido um evento de envio dentro do mesmo intervalo. O Algoritmo 3.2 mostra como este protocolo é implementado em um processo p_i. Note que é necessário apenas manter uma variável *booleana* para indicar se um evento de envio já ocorreu no intervalo corrente. Inicialmente, esta variável é inicializada com o valor falso e sempre que uma mensagem for enviada, ela

[1] Em inglês: *model-based*.

Algoritmo 3.2 Protocolo NRAS em um processo p_i.

Estruturas de Dados
1: **Var**
2: *enviou*: *booleano*

Inicialização
1: *enviou* ← *falso*

Ao enviar mensagem m
1: *enviou* ← *verdadeiro*
2: $envia_{rede}(m)$

Ao receber mensagem m
1: **if** *enviou* **then**
2: grava *checkpoint*
3: **end if**
4: *entrega*(m)

Ao gravar *checkpoint*
1: *enviou* ← *falso*

se torna verdadeira. Sendo assim, ao receber uma mensagem, basta testar o conteúdo da variável e se ela for verdadeira, forçar a gravação de um *checkpoint* antes de sua entrega para a aplicação. Além disso, sempre que um *checkpoint* (básico ou forçado) é gravado, esta variável recebe o valor falso pois um novo intervalo entre *checkpoints* acabou de ser criado.

É fácil verificar que o padrão de *checkpoints* e mensagens gerado pelo protocolo NRAS satisfaz a condição do Teorema 3.1 pois nenhuma mensagem pode ser entregue após um evento de envio ter ocorrido em um mesmo intervalo entre *checkpoints*. A Figura 3.4 apresenta o padrão gerado pelo protocolo NRAS para o mesmo CCP da Figura 3.2. Os *checkpoints* forçados estão representados pelos quadrados com um hachurado diferente. Note que agora todos os *checkpoints* são úteis, inclusive os básicos que originalmente não eram.

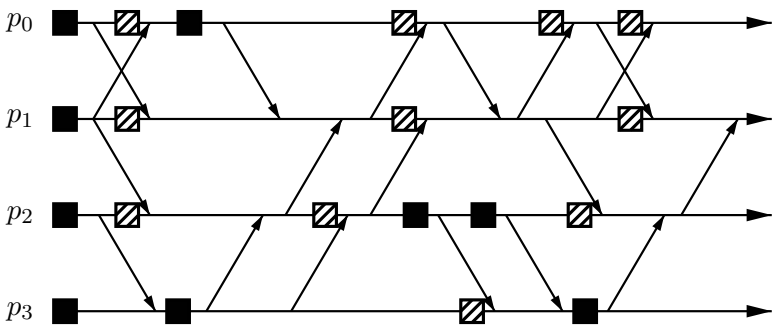

Figura 3.4: Exemplo de padrão gerado pelo protocolo NRAS.

Outro protocolo SZPF que merece ser comentado é o CAS (*Checkpoint-After-Send*) [36], o qual é apresentado no Algoritmo 3.3. Este protocolo induz um processo a gravar um *checkpoint* forçado sempre após o envio de uma mensagem. Isto garante que haverá no máximo um evento de envio por intervalo entre *checkpoints* e que este será o último evento do intervalo, satisfazendo a definição de padrão SZPF. Esta estratégia é muito mais simples que a utilizada pelo NRAS mas gera um número maior de *checkpoints* forçados no padrão de *checkpoints* e mensagens. Na realidade o número de *checkpoints* forçados gerados pelo protocolo CAS corresponde exatamente ao número de mensagens propagadas pela aplicação distribuída.

Algoritmo 3.3 Protocolo CAS em um processo p_i.

Ao enviar mensagem m
1: $envia_{rede}(m)$
2: grava *checkpoint*

A grande vantagem do padrão gerado por este protocolo está no fato de que o último *checkpoint* estável de um processo não precede nenhum *checkpoint* de outro processo pois nenhuma mensagem pode ser enviada depois dele. Se uma mensagem é enviada, um *checkpoint* forçado é gravado e este passará a ser o último *checkpoint* estável do processo. Por causa disso, o retrocesso do estado de um processo ao seu último *checkpoint* estável não força o retrocesso de estado de nenhum outro processo. Desta maneira, ao se recuperar de uma falha um processo precisa apenas retroceder ao seu último *checkpoint* estável, sem se coordenar ou forçar o retrocesso de qualquer outro processo. Isto garante também que nenhum outro *checkpoint* estável além do último será necessário no futuro e, portanto, apenas um *checkpoint* por processo precisa ser mantido ao longo da execução. No entanto, na prática, para garantir que o último *checkpoint* de um processo realmente não possa preceder algum *checkpoint* de outro processo é necessário que as linhas 1 e 2 do Algoritmo 3.3 sejam executadas atomicamente, ou seja, sem a possibilidade de ocorrência de uma falha entre elas ou durante sua execução parcial. Em geral, como o evento de envio de uma mensagem não altera o estado de um processo, pode-se obter este efeito apenas trocando a ordem das duas linhas e enviando a mensagem após a gravação do *checkpoint*.

3.2.2 Classe ZPF e Propriedade RDT

A principal característica dos protocolos da classe SZPF é o fato de todas as Z-precedências serem causais, o que impede a formação de ciclos-Z e permite a construção de algoritmos eficientes para a construção de *checkpoints* globais consistentes [36]. No entanto, para obter esta propriedade não é necessário quebrar todos os caminhos-Z que estejam para se formar. Especificamente, caminhos-Z que estejam duplicados por um caminho-C não precisam

ser quebrados. Os protocolos da classe ZPF, portanto, garantem apenas que todas as dependências entre os *checkpoints* dentro do padrão são causais, sem restrições com relação aos caminhos-Z duplicados causalmente.

Definição 3.2 ***Padrões ZPF*** — *Um padrão de* checkpoints *e mensagens é ZPF se, e somente se, dados dois* checkpoints c_a^α *e* c_b^β *pertencentes ao padrão,*

$$c_a^\alpha \rightsquigarrow c_b^\beta \iff c_a^\alpha \rightarrow c_b^\beta.$$

Wang identificou uma propriedade presente em uma série de protocolos para *checkpointing* que possibilitam a captura *on-line* das dependências entre os *checkpoints* por meio de vetores de dependências [36]. Esta propriedade recebeu o nome de *propriedade RDT* (*Rollback-Dependency Trackability*). Conforme vimos na Seção 2.4.1, vetores de dependências capturam as precedências causais, de forma que esta propriedade se torna equivalente à construção de um padrão ZPF [27]. O nome RDT, no entanto, é utilizado com uma freqüência muito maior do que ZPF na literatura para designar tanto protocolos quanto padrões que possuem esta característica [4, 5, 6, 18, 19, 33, 36]. Sendo assim, utilizaremos ZPF apenas para designar esta classe de protocolos em comparações entre as diferentes classes de protocolos quase-síncronos, e RDT para a designação de uma definição, propriedade, padrão ou protocolo relativo a esta classe.

É fácil verificar que os protocolos RDT, assim como os protocolos SZPF, impedem a ocorrência de *checkpoints* inúteis e evitam, portanto, o efeito dominó. Este fato pode ser verificado pelo Teorema 3.2, apresentado a seguir.

Teorema 3.2 *Todos os* checkpoints *de um padrão RDT são úteis.*

Prova: Vamos supor, por contradição, que exista um *checkpoint* inútil c_i^γ. Sabemos, então, que $c_i^\gamma \rightsquigarrow c_i^\gamma$. No entanto, pela Definição 3.2 temos que $c_i^\gamma \rightarrow c_i^\gamma$, o que é impossível. □

Um protocolo RDT é ótimo se a eliminação de qualquer *checkpoint* forçado resulta em uma relação $c_a^\alpha \rightsquigarrow c_b^\beta$ tal que $c_a^\alpha \not\rightarrow c_b^\beta$. Projetar um protocolo RDT ótimo parece impossível, pois deveria levar em consideração informação futura [27]. A Figura 3.5 apresenta um cenário onde p_1 não tem como saber que não há a necessidade de um *checkpoint* forçado para prevenir a formação da Z-precedência não-causal $c_2^0 \rightsquigarrow c_0^1$, visto que a mensagem m que duplica causalmente esta Z-precedência só será enviada no futuro da execução.

No entanto, um grande esforço tem sido realizado na busca de protocolos que utilizem condições cada vez mais fortes para a tomada de decisão sobre gravar ou não um *checkpoint* forçado. Neste contexto, os resultados mais importantes são aqueles relacionados com a busca de uma caracterização minimal dos padrões de *checkpoints* e mensagens que satisfazem a propriedade RDT [5, 6, 18]. Tal caracterização fornece a condição mais forte possível a ser testada localmente por um processo para verificar a necessidade de gravação de um *checkpoint*

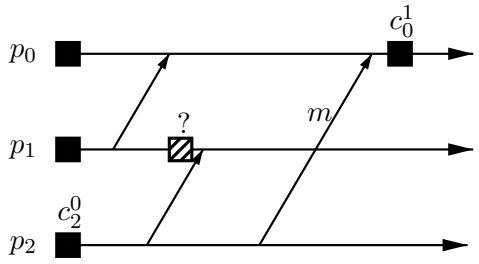

Figura 3.5: Dificuldade de se projetar um protocolo RDT ótimo.

forçado. No entanto, mesmo com a sua descoberta realizada por Garcia e Buzato [18], é impossível construir um protocolo RDT que grave menos *checkpoints* forçados que todos os outros protocolos em todos os cenários de execução possíveis [33]. Isto ocorre basicamente porque a gravação de um *checkpoint* forçado em um determinado momento pode diminuir a necessidade de gravação de outros *checkpoints* forçados no futuro. Nestas condições, um protocolo que utilize uma condição forte poderia deixar de gravar um *checkpoint* forçado que não seja necessário em um determinado momento, mas ter que gravar um ou mais *checkpoints* forçados no futuro por causa disso.

Apresentaremos agora o protocolo RDT chamado FDAS (*Fixed-Dependency-After-Send*), o qual foi proposto juntamente com a definição da propriedade RDT [36]. Este protocolo utiliza um vetor de dependências para capturar as precedências causais entre os *checkpoints*. Quando uma mensagem é enviada, o protocolo evita que o vetor de dependências do processo emissor se altere até o final do intervalo entre *checkpoints* corrente. Se o vetor deve ser alterado pela recepção de uma mensagem com nova informação causal, um *checkpoint* forçado é gravado. Isto garante que as mensagens enviadas não formarão Z-precedências que não possam ser capturadas diretamente pelos vetores de dependências, isto é, que não sejam causais. Ao mesmo tempo, este protocolo evita a gravação de *checkpoints* forçados para quebrar alguns caminhos-Z que já tenham sido duplicados causalmente. Como exemplo, considere a Figura 3.6, onde o processo p_1, ao receber m_3, já sabe que o caminho-Z $[m_3, m_2]$ é duplicado causalmente pelo caminho-C $[m_1, m_2]$ e portanto, pela estratégia do FDAS, não precisa gravar um *checkpoint* forçado para garantir a propriedade RDT.

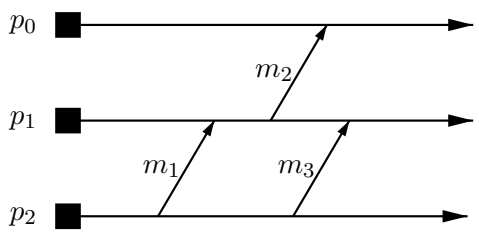

Figura 3.6: Exemplo de caminho-Z que não é quebrado pelo FDAS.

O Algoritmo 3.4 apresenta a estrutura do protocolo FDAS dentro de um processo. É necessário manter durante a sua execução um vetor de dependências e uma variável *booleana* como a utilizada pelo NRAS. As ações executadas durante a inicialização e o envio de uma mensagem são triviais com relação a estas estruturas. Como o vetor de dependências deve ser enviado na forma de um *timestamp* nas mensagens da aplicação, utilizamos $m.DV$ para representar tal estrutura dentro de uma mensagem. Ao receber uma mensagem proveniente de um processo p_k deve-se testar se existe alguma nova precedência causal com relação ao vetor mantido. Neste ponto uma pequena otimização pode ser realizada, comparando-se apenas a entrada $DV[k]$ dos dois vetores (linha 1), uma vez que qualquer precedência causal nova exige a alteração desta entrada no protocolo FDAS [17]. Então, se uma nova precedência causal é recebida e alguma mensagem foi enviada no intervalo entre *checkpoints* corrente, um *checkpoint* forçado é gravado (linha 3). Independentemente do envio de mensagens, o vetor de dependências deve ser atualizado quando uma nova precedência causal é percebida (linhas 5-9). Ao gravar um *checkpoint* básico ou forçado deve-se apenas atualizar a variável *enviou* e a entrada $DV[i]$ do vetor de dependências.

A Figura 3.7 mostra o padrão gerado pelo protocolo FDAS para o mesmo CCP mostrado na Figura 3.2. Note que o número de *checkpoints* forçados é menor do que no cenário de execução do protocolo NRAS. Isto ocorre porque nem todos os caminhos-Z são quebrados. No entanto, o protocolo FDAS não apresenta o melhor desempenho dentre os protocolos RDT com relação ao número de *checkpoints* forçados. Comparações baseadas em simulações mostram que protocolos baseados em condições mais fortes como o BHMR [4] e o RDT-Partner [19] apresentam um desempenho melhor na prática [34].

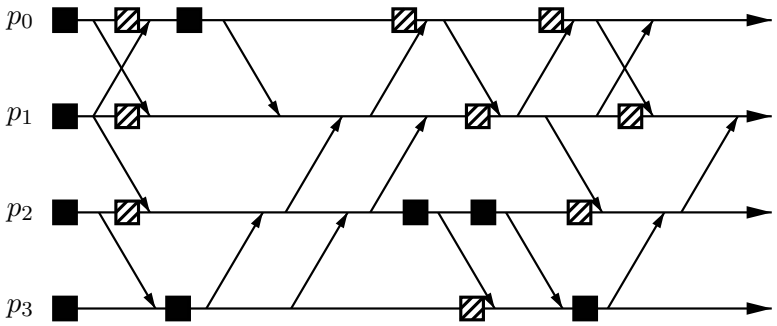

Figura 3.7: Exemplo de padrão gerado pelo protocolo FDAS.

3.2.3 Classe ZCF

Protocolos desta classe livram o padrão de *checkpoints* e mensagem dos ciclos-Z, garantindo apenas a inexistência de *checkpoints* inúteis. Por esta razão, esta é a classe mais fraca que garante o aproveitamento de todos os *checkpoints* selecionados pelos processos em pelo menos um *checkpoint* global consistente.

Algoritmo 3.4 Protocolo FDAS em um processo p_i.

Estruturas de Dados

```
1: Var
2:    enviou: booleano
3:    DV: array[0...n-1] of inteiro
```

Inicialização

```
1: enviou ← falso
2: for i ← 0 to n − 1 do
3:    DV[i] ← 0
4: end for
```

Ao enviar mensagem m

```
1: enviou ← verdadeiro
2: for j ← 0 to n − 1 do
3:    m.DV[j] ← DV[j]
4: end for
5: envia_rede(m)
```

Ao receber mensagem m **de** p_k

```
1: if m.DV[k] > DV[k] then
2:    if enviou then
3:       grava checkpoint
4:    end if
5:    for j ← 0 to n − 1 do
6:       if m.DV[j] > DV[j] then
7:          DV[j] ← m.DV[j]
8:       end if
9:    end for
10: end if
11: entrega(m)
```

Ao gravar *checkpoint*

```
1: enviou ← falso
2: DV[i] ← DV[i] + 1
```

Definição 3.3 ***Padrões ZCF*** — *Um padrão de* checkpoints *e mensagens é ZCF se, e somente se, ele não possui nenhum ciclo-Z.*

Um protocolo ZCF é ótimo se a eliminação de qualquer *checkpoint* forçado resulta em um ciclo-Z. Projetar um protocolo ZCF ótimo parece ser impossível pois os processos deveriam ser capazes de capturar de forma *on-line* Z-precedências não-causais para poder identificar corretamente a formação de um ciclo-Z e, assim, quebrá-lo [27].

Os protocolos ZCF são, em geral, bem simples e eficientes em termos da informação de controle mantida e propagada e são comumente chamados de *baseados em índice*[2] [13]. A razão deste nome decorre da utilização de relógios lógicos (Seção 2.2.2) para controlar a possível formação de ciclos-Z. Em geral isto é garantido por tais protocolos pela seguinte propriedade presente nos padrões de *checkpoints* e mensagens gerados por eles:

$$c \rightsquigarrow c' \Rightarrow LC(c) < LC(c') \tag{3.1}$$

Mostramos no Teorema 3.3 que um padrão com esta propriedade é ZCF.

Teorema 3.3 *Um padrão de* checkpoints *e mensagens que satisfaça a condição apresentada na Equação 3.1 é ZCF.*

Prova: Vamos supor, por contradição que o padrão gerado não é ZCF. Neste caso, existe pelo menos um *checkpoint* c_i^γ tal que $c_i^\gamma \rightsquigarrow c_i^\gamma$. No entanto, pela Equação 3.1, temos que $LC(c_i^\gamma) < LC(c_i^\gamma)$, o que é uma contradição. □

Um dos primeiros protocolos da classe ZCF proposto foi o BCS (Briatico-Ciuffoletti-Simoncini) [10]. Este protocolo é baseado em relógios lógicos e garante a propriedade da Equação 3.1. O relógio mantido pelo processo é incrementado sempre que um *checkpoint* é gravado. A propriedade é garantida pela gravação de um *checkpoint* forçado antes da entrega de uma mensagem que contenha um relógio maior que aquele mantido pelo próprio processo. Fazendo isso, todas as precedências decorrentes de alguma mensagem enviada antes e que continham um valor de relógio menor são quebradas. Algumas otimizações são possíveis em cima da idéia básica do algoritmo. Uma das mais simples consiste em evitar de gravar um *checkpoint* forçado caso nenhuma mensagem tenha sido enviada no intervalo entre *checkpoints*, o que também é feito no NRAS e no FDAS. Esta e outras otimizações foram apresentadas pela primeira vez juntamente com um outro protocolo, chamado BQF [7]. Nesta dissertação, a exemplo de trabalhos anteriores [34], utilizaremos o nome BCS-Aftersend para o protocolo BCS apenas com a otimização comentada acima.

O Algoritmo 3.5 apresenta a descrição do protocolo BCS-Aftersend. Inicialmente, o relógio lógico recebe valor -1; porém, sabemos que a primeira ação de um processo é gravar um *checkpoint* básico, de forma que seu valor passa a ser 0. O relógio corrente de um processo é enviado na forma de um *timestamp* nas mensagens da aplicação. Ao receber uma mensagem com o valor do relógio maior que o corrente (linha 1), o protocolo verifica a variável *enviou* (linha 2) para saber se deve gravar um *checkpoint* forçado. Independentemente da gravação de um *checkpoint* forçado, o valor do relógio corrente é atualizado (linha 5). Ao gravar um *checkpoint* básico ou forçado o relógio é incrementado. A variável *enviou* é atualizada exatamente da mesma que nos protocolos NRAS e FDAS.

[2]Em inglês: *index-based*.

Algoritmo 3.5 Protocolo BCS-Aftersend em um processo p_i.

Estruturas de Dados

1: **Var**
2: *enviou*: *booleano*
3: LC: *inteiro*

Inicialização

1: $enviou \leftarrow falso$
2: $LC \leftarrow -1$

Ao enviar mensagem m

1: $enviou \leftarrow verdadeiro$
2: $m.LC \leftarrow LC$
3: $envia_{rede}(m)$

Ao receber mensagem m

1: **if** $m.LC > LC$ **then**
2: **if** *enviou* **then**
3: grava *checkpoint*
4: **end if**
5: $LC \leftarrow m.LC$
6: **end if**
7: $entrega(m)$

Ao gravar *checkpoint*

1: $enviou \leftarrow falso$
2: $LC \leftarrow LC + 1$

A Figura 3.8 apresenta o padrão gerado pelo protocolo BCS-Aftersend para o CCP mostrado na Figura 3.2. Note que o número de *checkpoints* forçados é menor que o gerado pelos protocolos NRAS e FDAS pois nem todos os caminhos-Z são quebrados, incluindo até mesmo alguns caminhos-Z não duplicados causalmente. Mesmo assim, todos os *checkpoints* são úteis e o efeito dominó é evitado.

3.2.4 Classe PZCF

Os protocolos PZCF não garantem a ausência de ciclos-Z, mas se esforçam para evitá-los [27]. Eles se aproximam dos algoritmos assíncronos em relação à dificuldade de se obter um *checkpoint* global consistente, mas procuram dar algumas garantias estatísticas sobre o impacto da ocorrência de *checkpoints* inúteis.

Definição 3.4 ***Padrões PZCF*** — *Um padrão de* checkpoints *e mensagens é PZCF se nem todos os ciclos-Z são quebrados.*

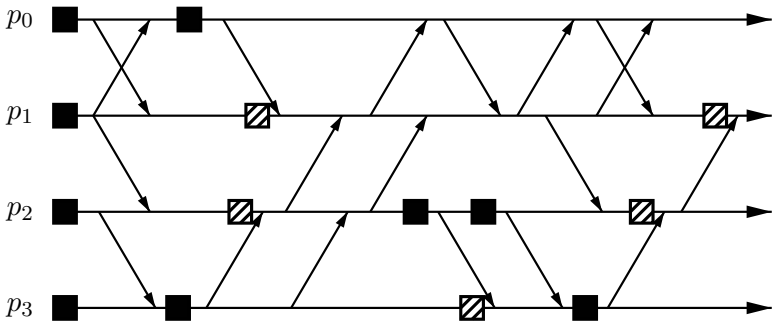

Figura 3.8: Exemplo de padrão gerado pelo protocolo BCS-Aftersend.

Os poucos protocolos da classe PZCF conhecidos diferem bastante com relação às Z-precedências restringidas por eles [27]. Sendo assim, não é possível atribuir ou verificar alguma propriedade global da classe. A principal vantagem deste tipo de protocolo está no menor número de *checkpoints* forçados com relação a todas as outras. Entretanto, a presença de *checkpoints* inúteis pode levar à ocorrência do efeito dominó. Esta desvantagem, de certa forma, inviabiliza sua utilização prática. Por este motivo, não estamos muito interessados em tais protocolos nesta dissertação.

3.3 Comparações e Conclusão

A forma de comparação mais simples entre os protocolos para *checkpointing* refere-se ao número de *checkpoints* induzidos produzidos por eles. Em um extremo temos os protocolos assíncronos que não induzem nenhum *checkpoint* além dos básicos selecionados pelos próprios processos. Em outro extremo temos os protocolos síncronos, onde cada *checkpoint* básico de um processo pode gerar até $n-1$ *checkpoints* forçados, ou seja, um para cada um dos demais processos. O número de *checkpoints* induzidos nos protocolos quase-síncronos é variável e depende, entre outras coisas, da classe de protocolos e da condição testada para gravar um *checkpoint* forçado. Vieira [34] apresenta uma ampla comparação entre protocolos quase-síncronos das diferentes classes existentes com relação a este parâmetro. Segundo seus resultados, temos que o número de *checkpoints* forçados em protocolos SZPF e ZPF está intimamente relacionado com a quantidade de mensagens trocadas pelos processos. Os protocolos da classe ZCF apresentam um número bem menor de *checkpoints* forçados em geral. Especificamente, protocolos que utilizam relógios lógicos podem gerar no máximo $n-1$ *checkpoints* forçados por *checkpoint* básico. Isto ocorre porque o relógio propagado a partir de um *checkpoint* básico só será incrementado pela ocorrência de outro *checkpoint* básico em seu caminho, podendo gerar no máximo um *checkpoint* forçado para cada um

dos demais processos caso seja propagado para todos eles e todos mantenham um relógio de valor menor. Esta característica coloca a classe de protocolos ZCF em um patamar de comparação igual ou melhor que os protocolos síncronos.

Outro parâmetro de comparação importante é o custo extra causado pelo protocolo no mecanismo de comunicação. Os protocolos síncronos exigem a troca de mensagens de controle entre os processos toda vez que o algoritmo for executado, o que pode gerar conflitos no meio de comunicação e causar perda ou atraso na entrega das mensagens da aplicação. Esta característica se torna mais prejudicial à medida que a escala do sistema é aumentada [13]. Protocolos quase-síncronos não trocam mensagens de controle, mas as classes ZPF, ZCF e PZCF enviam dados na forma de *timestamps* nas mensagens da própria aplicação. Protocolos ZPF tendem a utilizar estruturas maiores, principalmente pela propagação de vetores de dependências, enquanto os protocolos ZCF em geral propagam apenas alguns valores inteiros [34]. Os *timestamps* utilizados em protocolos PZCF variam de algoritmo para algoritmo [27].

O tamanho dos *timestamps* utilizados nos protocolos quase-síncronos influencia diretamente na complexidade de tempo das ações executadas pelos protocolos. Protocolos SZPF como o NRAS e o CAS apresentados nos Algoritmos 3.2 e 3.3, respectivamente, possuem complexidade $O(1)$ em todos os seu eventos uma vez que nenhum *timestamp* precisa ser propagado ou tratado. Já protocolos ZPF como o FDAS apresentado no Algoritmo 3.4 geralmente apresentam complexidade $\Omega(n)$ em praticamente todos os eventos em virtude da manutenção do vetor de dependências. Com a otimização que mostramos no FDAS para o evento de recebimento de uma mensagem, ainda conseguimos uma complexidade $O(1)$ quando nenhuma nova precedência causal é percebida. Finalmente, como a maioria dos protocolos ZCF propagam apenas alguns inteiros, tais protocolos executam com complexidade $O(1)$ em todos os eventos tratados, assim como o protocolo BCS-Aftersend mostrado no Algoritmo 3.5.

Protocolos síncronos favorecem o mecanismo de recuperação de falhas. Como cada execução do protocolo gera um *checkpoint* global consistente em meio estável, na pior das hipóteses o retrocesso levará a computação para o *checkpoint* global gerado pela última execução completa do protocolo [30, 31]. Algum esforço pode ser realizado para não retroceder os processos que não precisem [23]. Protocolos assíncronos e quase-síncronos necessitam de um mecanismo de recuperação mais sofisticado, capaz de capturar as Z-precedências entre os *checkpoints* e determinar a linha de recuperação a ser utilizada. Neste caso, os protocolos da classe ZPF levam vantagem uma vez que todas as Z-precedências existentes são causais e podem ser facilmente capturadas. A facilidade de construir a linha de recuperação foi a primeira grande motivação para a utilização dos protocolos desta classe [36].

Recentemente descobriu-se também que os protocolos para *checkpointing* podem ser classificados de acordo com a distância de retrocesso [1]. O melhor protocolo segundo esta

métrica é o CAS, uma vez que um processo pode se recuperar de uma falha sem retroceder nenhum outro processo. Além disto, foi provado que os protocolos síncronos e os quase-síncronos da classe ZPF apresentam um retrocesso máximo $n-1$ vezes menor que os protocolos ZCF, o que vem a se somar nas vantagens da utilização de protocolos ZPF ao invés de ZCF em recuperação por retrocesso. Como os protocolos PZCF são suscetíveis ao efeito dominó, eles não apresentam limite para a distância de retrocesso.

Uma preocupação existente com relação ao mecanismo de *checkpointing* é o espaço necessário em meio de armazenamento estável para guardar os *checkpoints* gravados. O problema de liberar espaço em meio estável eliminando os *checkpoints* que não podem ser mais utilizados (obsoletos) é chamado de coleta de lixo. Em geral, protocolos síncronos não precisam armazenar mais do que dois *checkpoints* por processo durante a execução da computação [23]. No entanto, para os protocolos quase-síncronos não era conhecida uma forma de limitar o número de *checkpoints* utilizados sem sincronizar os processos, o que é uma contradição com a definição deste tipo de protocolo. Este fator chegou a ser apresentado como uma importante desvantagem para a utilização destes protocolos na prática [2]. O problema de coleta de lixo é abordado nos dois capítulos seguintes e no Capítulo 5 apresentamos um protocolo de coleta de lixo para a classe ZPF que garante um limite para o número de *checkpoints* mantidos em meio de armazenamento estável sem a necessidade de sincronização entre os processos.

Por fim, acreditamos que a escolha por uma determinada classe de protocolo para *checkpointing* deve envolver um estudo das vantagens de cada uma delas para o modelo de computação utilizado e propriedades desejadas para o sistema de recuperação. Conforme vimos neste capítulo, cada classe possui suas vantagens e desvantagens com relação a estes parâmetros de forma que não existe uma classe ou protocolo superior em todos os aspectos.

Capítulo 4

Coleta de Lixo

À medida que a computação progride, novos *checkpoints* são selecionados e novas linhas de recuperação são formadas para os possíveis conjuntos de processos falhos. Este processo acaba tornando os *checkpoints* estáveis mais antigos obsoletos. Desta maneira, tais *checkpoints* devem ser descartados para liberar espaço em meio de armazenamento estável. Este problema é conhecido como coleta de lixo.

O primeiro passo para desenvolver um algoritmo de coleta de lixo é definir formalmente o que é um *checkpoint* obsoleto. Depois de ter este conceito definido, é possível desenvolver algoritmos capazes de identificar e eliminar este tipo de *checkpoint* baseado em condições suficientes. Dizemos que um algoritmo para coleta de lixo é ótimo quando ele é capaz de identificar e eliminar todos os *checkpoints* obsoletos de um CCP. Entretanto, o desenvolvimento de tal algoritmo está ligado à obtenção de uma caracterização dos *checkpoints* obsoletos, ou seja, uma condição necessária e suficiente para sua identificação.

4.1 Checkpoints Obsoletos

O conceito de *checkpoint* obsoleto é válido apenas para *checkpoints* estáveis pois está diretamente associado à coleta de lixo. Um *checkpoint* se torna obsoleto quando não for mais útil ao sistema de recuperação de falhas, isto é, quando não puder mais fazer parte de uma linha de recuperação no futuro da execução da computação distribuída.

Definição 4.1 ***Checkpoint Obsoleto*** — *Um* checkpoint *estável é obsoleto se, e somente se, ele não pode tomar parte em qualquer linha de recuperação presente ou futura, mesmo depois de um ou mais retrocessos.*

As próximas seções exploram esta definição, apresentando alguns algoritmos conhecidos para coleta de lixo e uma caracterização do conjunto de *checkpoints* obsoletos em um padrão de *checkpoints* e mensagens, a qual pode ser utilizada para a realização de uma coleta de lixo ótima.

4.2 Coleta de Lixo para *Checkpointing* Síncrono

Uma condição suficiente bastante simples para um *checkpoint* ser considerado obsoleto é estar no passado de um *checkpoint* global consistente composto apenas por *checkpoints* estáveis. Este *checkpoint* global funciona como uma barreira que impede os processos de retrocederem para qualquer *checkpoint* anterior a ela.

Teorema 4.1 *Um* checkpoint *é obsoleto se estiver no passado de um* checkpoint *global consistente C formado por* checkpoints *estáveis.*

Prova: Para provar este teorema é necessário apenas mostrar que nenhuma linha de recuperação pode conter um *checkpoint* que esteja no passado de C. Dessa forma, nenhum *checkpoint* de C pode ser retrocedido e a barreira persistirá mesmo depois de um ou vários retrocessos. Vamos supor, por contradição, que exista uma linha de recuperação R_F contendo um *checkpoint* anterior a C. Neste caso, $R_F \cup C$ teria um custo de retrocesso menor que R_F, o que é uma contradição com a definição de linha de recuperação (Figura 4.1). □

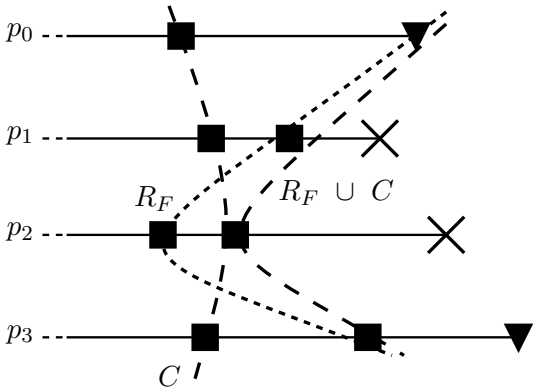

Figura 4.1: Barreira de *checkpoints* estáveis.

Esta condição é a base para o desenvolvimento de um algoritmo eficiente de coleta de lixo para protocolos de *checkpointing* síncronos, uma vez que eles constroem um *checkpoint* global consistente completo da computação ao final de sua execução. Dessa forma, todos os *checkpoints* anteriores ao novo *checkpoint* global criado podem ser eliminados. Se os *checkpoints* são selecionados por meio de execuções sucessivas de um protocolo síncrono, ao completar uma execução podemos eliminar os *checkpoints* criados na sessão anterior, mantendo apenas um *checkpoint* armazenado por processo durante o período entre duas execuções do protocolo para *checkpointing*.

Quando o protocolo está sendo executado pode ser necessário manter dois *checkpoints* por processo. Considere a Figura 4.2(a), a qual apresenta um CCP durante a execução de um protocolo para *checkpointing* síncrono. Por simplicidade, são mostradas apenas as mensagens da própria aplicação distribuída, abstraindo-se das mensagens de controle enviadas pelo protocolo utilizado. Na figura podemos ver um *checkpoint* global consistente C_1, gravado durante a execução anterior do protocolo para *checkpointing*. Além disso, existem dois novos *checkpoints* nos processos p_0 e p_1 que já foram gravados durante a execução atual do protocolo. Note que não é possível apagar os *checkpoints* anteriores destes processos porque C_2 ainda não está completo. Se o processo p_2 falhasse antes do protocolo terminar, a linha de recuperação seria formada pelos *checkpoints* de C_1 pois o último *checkpoint* estável de p_2 Z-precede os novos *checkpoints* gravados para compor C_2.

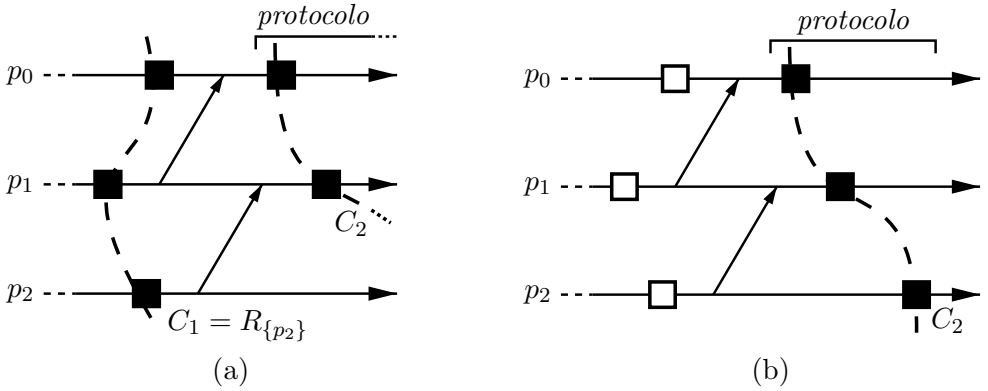

Figura 4.2: Coleta de lixo para protocolos síncronos.

Somente quando o protocolo termina, gerando o *checkpoint* global consistente C_2, é que os *checkpoints* que compõem C_1 podem ser coletados, liberando o espaço no meio de armazenamento estável, conforme pode ser visto na Figura 4.2(b) onde os *checkpoints* obsoletos estão representados por quadrados não preenchidos. Podemos concluir que é necessário manter os *checkpoints* anteriores até o final da execução de um protocolo síncrono para *checkpointing* pois só então teremos formado a barreira de *checkpoints* estáveis. Entretanto, alguns protocolos, especialmente os não-bloqueantes [11], não avisam a todos os processos quando o algoritmo terminou a sua execução, isto é, quando todos os *checkpoints* já foram gravados. Nestes casos se faz necessário um passo extra de sincronização de forma que os processos se avisem uns aos outros que eles já gravaram seu estado. Ao receber a notificação de término de todos os outros processos, um processo pode eliminar o seu *checkpoint* anterior.

A única complicação com relação ao limite de dois *checkpoints* por processos é a possibilidade de execução concorrente de duas ou mais instâncias do protocolo para *checkpointing* (por exemplo, quando dois processos iniciam o protocolo ao mesmo tempo), pois neste caso

podem haver dependências entre os *checkpoints* das diferentes instâncias e mais do que dois *checkpoints* teriam de ser armazenados. No entanto, isto pode ser resolvido pela utilização de um único processo capaz de invocar o protocolo [30, 31] ou pela utilização de um protocolo capaz de identificar e cancelar execuções concorrentes [23].

4.3 Coleta de Lixo Ingênua

A linha de recuperação mais antiga para a qual a computação poderia retroceder é aquela obtida por uma falha de todos os processos ($F = P$) pois ela incorpora os retrocessos causados pela falha de cada um dos processos da aplicação. Precisamente, a linha de recuperação referente a uma falha de todos os processos não contém nenhum *checkpoint* volátil e corresponde ao *checkpoint* global consistente mais recente composto somente por *checkpoints* estáveis. Desta forma, pelo Teorema 4.1, ela pode ser utilizada para identificar e eliminar *checkpoints* obsoletos por meio do que chamamos de coleta de lixo ingênua [9, 22, 32]. A Figura 4.3 apresenta a linha de recuperação R_P para um padrão de *checkpoints* e mensagens. Os *checkpoints* c_1^7 e c_3^6 são obsoletos, uma vez que estão no passado de R_P.

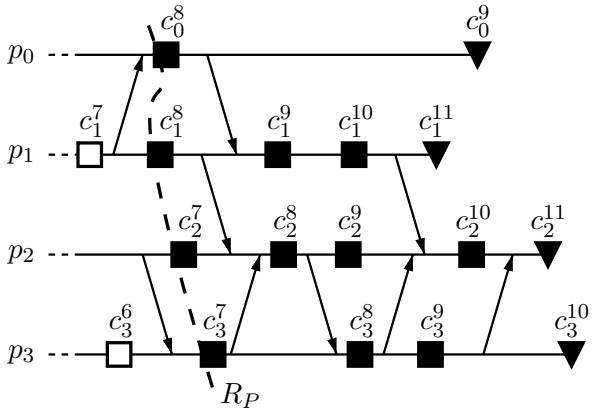

Figura 4.3: Exemplo de coleta de lixo ingênua.

Este método para coleta de lixo pode ser utilizado com qualquer protocolo para *checkpointing* assíncrono ou quase-síncrono. Uma implementação simples segue os passos do algoritmo de recuperação síncrono descrito na Seção 2.4.2 [9]. Em uma primeira fase, o coordenador coleta informações sobre as dependências existentes entre os *checkpoints* dos diferentes processos e monta uma fotografia do padrão de *checkpoints* e mensagens da computação distribuída, possivelmente utilizando um *R-graph*. Depois de fazer isto, basta calcular localmente a linha de recuperação R_P e propagá-la para os demais processos de forma

que eles possam eliminar todos os *checkpoints* anteriores a ela. Obviamente, todos os *checkpoints* eliminados não precisam aparecer nos CCPs futuros da computação e podem ter suas informações de dependências eliminadas também. Assim, tais *checkpoints* não estarão presentes nos *R-graphs* futuros a serem utilizados para retrocesso ou coleta de lixo.

A coleta de lixo ingênua pode ser facilitada pela presença de um monitor da computação distribuída [16]. Neste caso, à medida que novos *checkpoints* são selecionados e suas informações são enviadas ao monitor, este pode construir progressivamente a linha de recuperação para uma falha total dos processos e propagar esta informação aos demais processos para que estes possam eliminar os *checkpoints* mais antigos [15, 22]. Além disso, como a linha de recuperação é calculada de forma progressiva, o monitor pode avisar um processo apenas quando a sua componente seja alterada, economizando mensagens de controle extras.

Apesar de simples, a coleta de lixo ingênua apresenta dois grandes problemas. O primeiro é que o cálculo da linha de recuperação para uma falha de todos os processos requer o conhecimento de informação global do sistema. Isto quer dizer que um processo não consegue realizá-lo sem o recebimento de mensagens dos outros processos. Como exemplo considere a Figura 4.4(a), na qual apenas uma mensagem é trocada no início da execução. Neste caso, o processo p_0 não tem como saber se o processo p_1 gravou algum outro *checkpoint* depois de c_1^0 e, portanto, não pode eliminar nenhum de seus *checkpoints* apesar de todos exceto o último serem obsoletos. Sendo assim, a coleta de lixo ingênua necessita que mensagens de controle sejam trocadas entre os processos além das mensagens da própria aplicação, o que vai totalmente de encontro à principal vantagem dos protocolos assíncronos e quase-síncronos, que é a ausência deste tipo de mensagem.

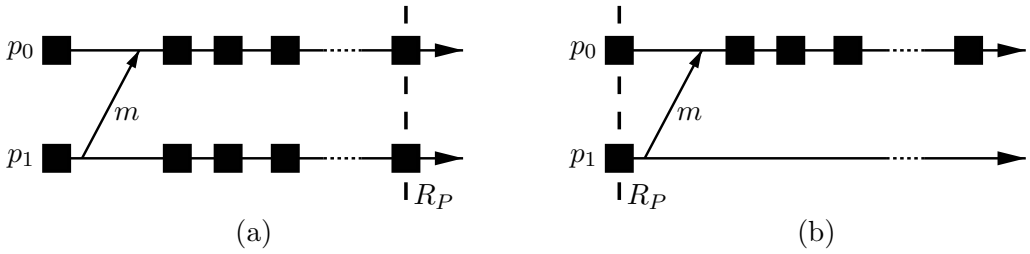

Figura 4.4: Problemas da coleta de lixo ingênua.

O outro problema refere-se ao número máximo de *checkpoints* não coletados, isto é, a quantidade máxima de *checkpoints* que é mantida em meio de armazenamento estável durante a execução da computação. A existência de um limite é importante para sistemas onde a memória estável é escassa ou possui um custo elevado, de forma que não possa ser desperdiçada. Na Figura 4.4(b) temos um exemplo em que todos os *checkpoints* do processo p_0 seriam preservados pela coleta ingênua, não importando quantos eles sejam. Note que a

ausência de um *checkpoint* do processo p_1 posterior ao envio da mensagem m impede que a barreira de *checkpoints* estáveis avance.

Além disso, podemos facilmente verificar que a coleta de lixo ingênua não elimina todos os *checkpoints* obsoletos. Na figura 4.4(b), todos os *checkpoints* de p_0 posteriores à entrega de m com exceção do último são obsoletos. A ausência de eventos de comunicação entre os *checkpoints* implica na possibilidade de eles serem substituídos pelo último em um *checkpoint* global consistente, o que minimizaria o custo de retrocesso de qualquer linha de recuperação que os contivesse. Além disso, nada impede que existam *checkpoints* inúteis após R_P, os quais são obsoletos por definição. Sendo assim, devemos buscar uma identificação completa dos *checkpoints* obsoletos de forma a obter um algoritmo ótimo para coleta de lixo.

4.4 Caracterização dos *Checkpoints* Obsoletos

Para melhorar os mecanismos para coleta de lixo é necessário um melhor entendimento sobre a formação dos *checkpoints* obsoletos. Se conseguirmos entender o processo pelo qual um *checkpoint* se torna obsoleto, conseguiremos desenvolver algoritmos capazes de identificar e eliminar todos eles, independente de estarem no passado ou no futuro de um *checkpoint* global consistente formado por *checkpoints* estáveis.

4.4.1 Linha de Recuperação Baseada em Z-Precedência

Uma outra forma de se determinar a linha de recuperação da computação é por meio da relação de Z-precedência. Neste caso, ao invés de utilizarmos as dependências de retrocesso entre os *checkpoints* (caminho em um *R-graph*), utilizamos as dependências de visualização existentes entre eles (Z-precedência). As duas formas são equivalentes para padrões de *checkpoints* e mensagens gerais com todas as formas possíveis de dependências entre *checkpoints*. No entanto, protocolos para *checkpointing* quase-síncronos são classificados de acordo com as formas de dependências permitidas em um CCP da computação, conforme apresentamos no Capítulo 3. Neste caso, a utilização de uma estrutura mais abstrata como um *R-graph* pode impedir a visualização de algumas propriedades com relação à linha de recuperação que são específicas de alguns protocolos onde nem todas as dependências são permitidas.

Começaremos nossa caracterização da linha de recuperação baseada em Z-precedência pelo Teorema 4.2, que mostra como determinar a linha de recuperação para um único processo falho. Informalmente, a linha de recuperação é composta pelo último *checkpoint* de cada processo, volátil ou não, que não é Z-precedido pelo último *checkpoint* estável do processo falho.

Teorema 4.2 *Dados um padrão de* checkpoints *e mensagens e um processo falho p_f, a linha de recuperação $R_{\{p_f\}}$ é determinada por:*

$$R_{\{p_f\}} = \bigcup_{i=0}^{n-1} \{c_i^{\max(k)} | s_f^{last} \not\rightsquigarrow c_i^k\}$$

Prova: Três coisas precisam ser evidenciadas a respeito do conjunto $R_{\{p_f\}}$: ele está bem definido, corresponde a um *checkpoint* global consistente, e minimiza o custo de retrocesso. A primeira parte é a mais simples de ser verificada, uma vez que o primeiro *checkpoint* de cada processo, s_i^0, não pode ser Z-precedido por nenhum outro *checkpoint*. Sendo assim, pela condição de maximização da variável k para cada processo, podemos facilmente verificar que $R_{\{p_f\}}$ é composto por exatamente um *checkpoint* de cada processo $p_i \in P$.

Agora vamos supor, por contradição, que $R_{\{p_f\}}$ é inconsistente. Neste caso, existem dois *checkpoints*, c_a^α e c_b^β, tais que $c_a^\alpha \rightarrow c_b^\beta$. Esta precedência causal implica que c_a^α não é um *checkpoint* volátil e, portanto, existe um *checkpoint* $c_a^{\alpha+1}$ que é Z-precedido por s_f^{last}. Entretanto, pela definição de Z-precedência, se $s_f^{last} \rightsquigarrow c_a^{\alpha+1}$ e $c_a^\alpha \rightarrow c_b^\beta$, então $s_f^{last} \rightsquigarrow c_b^\beta$, o que contradiz a definição do conjunto $R_{\{p_f\}}$.

Finalmente, vamos supor que $R_{\{p_f\}}$ não minimiza o custo de retrocesso. Sendo assim, existe um outro *checkpoint* global consistente $R'_{\{p_f\}}$ que possui pelo menos um *checkpoint* $c_e^{\epsilon'}$ tal que, dado o único *checkpoint* $c_e^\epsilon \in R_{\{p_f\}}$ do processo p_e, $\epsilon' > \epsilon$. Pela definição de $R_{\{p_f\}}$, $s_f^{last} \rightsquigarrow c_e^{\epsilon'}$, caso contrário $c_e^{\epsilon'}$ teria sido escolhido para compor $R_{\{p_f\}}$. Como p_f é um processo falho, ele precisa ser retrocedido para um de seus *checkpoints* estáveis. Porém, se $s_f^{last} \rightsquigarrow c_e^{\epsilon'}$ então todos os *checkpoints* estáveis de p_f Z-precedem $c_e^{\epsilon'}$ e, portanto, não podem fazer parte de um mesmo *checkpoint* global consistente que ele. Isto contradiz nossa suposição sobre a existência de $R'_{\{p_f\}}$. □

Depois de entendermos como a linha de recuperação é determinada para um único processo falho fica mais fácil de estender o raciocínio para qualquer conjunto $F \subseteq P$ de processos falhos. Neste caso, queremos os *checkpoints* mais recentes de cada processo que não violem a regra para composição da linha de recuperação para qualquer um dos processos presentes no conjunto F. Isto pode ser obtido pela operação de interseção entre *checkpoints* globais apresentada na Definição 2.5.

Teorema 4.3 *Dados um padrão de* checkpoints *e mensagens e um conjunto de processos falhos F, a linha de recuperação R_F é determinada por:*

$$R_F = \bigcap_{p_f \in F} R_{\{p_f\}}$$

Prova: Pelo Teorema 2.3, temos que R_F é um *checkpoint* global consistente. Além disso, pela definição de interseção de *checkpoints* globais sabemos que R_F não inclui o *checkpoint*

volátil de nenhum processo falho uma vez que $R_{\{p_f\}}$ não inclui v_f. Sendo assim, é necessário apenas provar que ele minimiza o custo de retrocesso. Vamos supor, por contradição, que R_F não minimiza o custo de retrocesso. Neste caso, existe um outro *checkpoint* global consistente R'_F que possui pelo menos um *checkpoint* $c_e^{\epsilon'}$ tal que, dado o único *checkpoint* $c_e^{\epsilon} \in R_F$ do processo p_e, $\epsilon' > \epsilon$. Pela definição de R_F, temos que $c_e^{\epsilon} \in R_{\{p_f\}}$ para algum processo $p_f \in F$ e, portanto, $s_f^{last} \rightsquigarrow c_e^{\epsilon'}$, o que impede que qualquer *checkpoint* estável de p_f faça parte de R'_F. Como p_f é um processo falho e precisa ser retrocedido, temos uma contradição com relação à existência de R'_F. □

Se juntarmos o Teorema 4.3 com o Teorema 4.2 obtemos uma caracterização da linha de recuperação diretamente baseada na Z-precedência com relação aos últimos *checkpoints* estáveis dos processos falhos. De uma forma geral, a linha de recuperação é composta pelo *checkpoint* mais recente de cada processo, volátil ou não, que não é Z-precedido pelo último *checkpoint* estável de algum processo falho.

Corolário 4.1 *Dados um padrão de* checkpoints *e mensagens e um conjunto de processos falhos F, a linha de recuperação R_F é determinada por:*

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

Prova: Pelos Teorema 4.2 e 4.3, e pela definição de interseção de *checkpoints* globais consistentes (Definição 2.5). □

A Figura 4.5 exemplifica o uso destes teoremas e corolário. O padrão de *checkpoints* e mensagens completo utilizado como exemplo é apresentado na Figura 4.5(a). A Figura 4.5(b) mostra a linha de recuperação $R_{\{p_0\}}$, construída com base no Teorema 4.2. Note que $s_0^{last} = c_0^8$, de forma que devemos procurar pelo último *checkpoint* de cada processo que não é Z-precedido por este *checkpoint*. Naturalmente, $c_0^8 \rightarrow c_0^9$; além disso, $c_0^8 \rightarrow c_1^9$ pelo envio de uma mensagem, e como $c_1^8 \rightarrow c_2^8$ e $c_1^8 \rightarrow c_3^8$, formam-se as Z-precedências $c_0^8 \rightsquigarrow c_2^8$ e $c_0^8 \rightsquigarrow c_3^8$. Reunindo todas estas precedências observamos que $c_0^8 \rightsquigarrow \{c_0^9, c_1^9, c_2^8, c_3^8\}$ e, como $c_0^8 \not\rightsquigarrow \{c_0^8, c_1^8, c_2^7, c_3^7\}$, concluímos que a linha de recuperação $R_{\{p_0\}}$ corresponde ao conjunto $\{c_0^8, c_1^8, c_2^7, c_3^7\}$. As Figuras 4.5(c), 4.5(d) e 4.5(e) apresentam, respectivamente, as linhas de recuperação $R_{\{p_1\}}$, $R_{\{p_2\}}$ e $R_{\{p_3\}}$ utilizando o mesmo raciocínio. Já a Figura 4.5(f) mostra um exemplo de composição de linha de recuperação para a falha conjunta dos processos p_1 e p_3. Conforme descrito pelo Teorema 4.3, temos que $R_{\{p_1,p_3\}} = R_{\{p_1\}} \cap R_{\{p_3\}}$. Como podemos verificar, este mesmo conjunto pode ser obtido pela aplicação do Corolário 4.1.

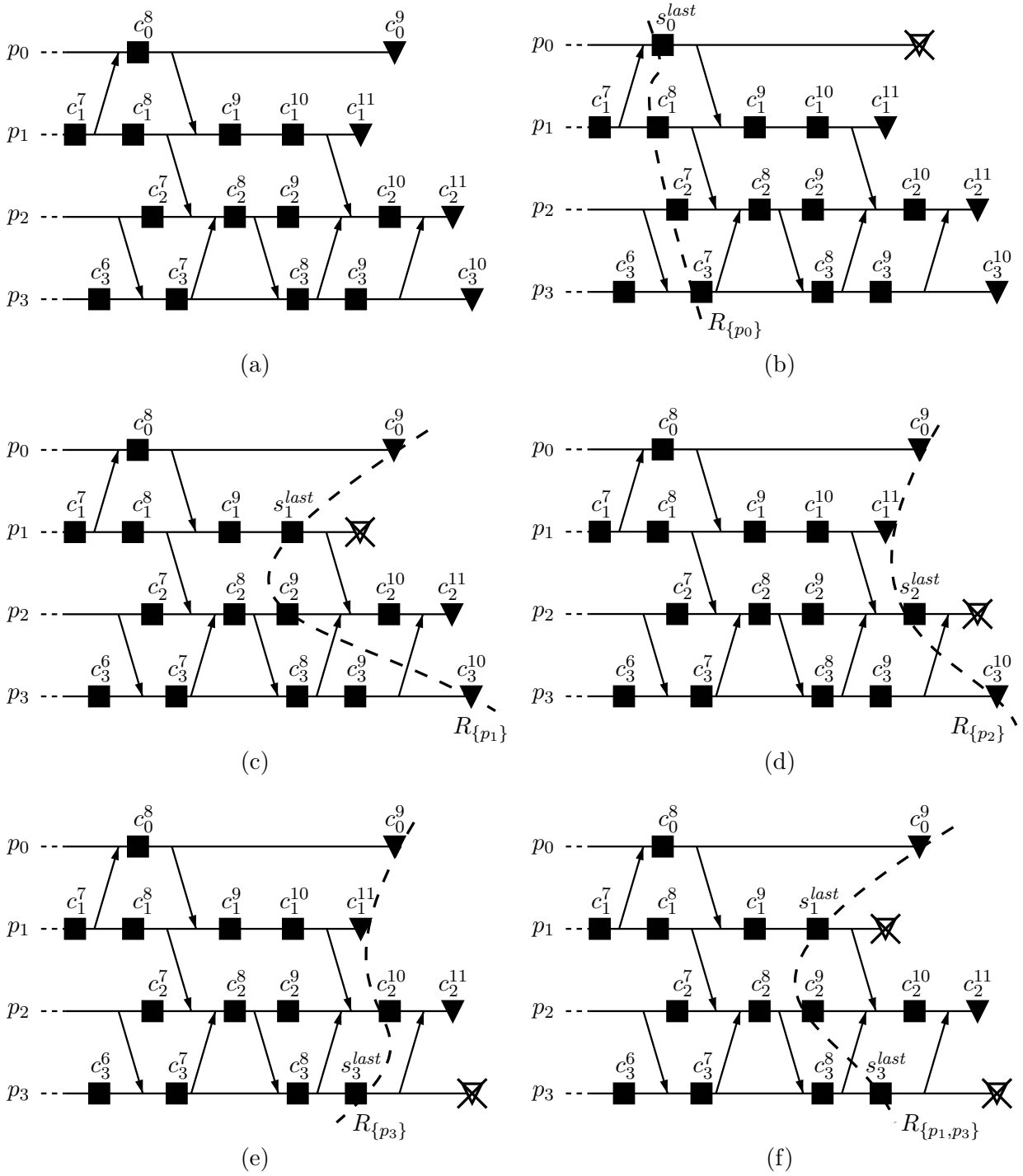

Figura 4.5: Exemplos de determinação da linha de recuperação.

Esta nossa caracterização da linha de recuperação pode ser adaptada aos tipos de precedências permitidas pelo protocolo para *checkpointing* utilizado. Conforme foi apresentado no Capítulo 3, as diferentes classes de protocolos quase-síncronos restringem os tipos de Z-precedências permitidos dentro do padrão gerado. Portanto, teremos regras diferentes para a construção da linha de recuperação em cada uma dessas classes. Nas seções seguintes, apresentamos uma caracterização dos *checkpoints* obsoletos baseada em nossa caracterização de linha de recuperação. Desta maneira, também é possível adaptá-la para as precedências permitidas pelos protocolo para *checkpointing*, o que pode facilitar o desenvolvimento de algoritmos para coleta de lixo.

4.4.2 *Checkpoints* Desnecessários

O Teorema 4.3 fornece uma maneira de facilmente identificar todos os *checkpoints* que fazem parte de alguma linha de recuperação para qualquer conjunto F de processos falhos, conforme podemos verificar no Corolário 4.2.

Corolário 4.2 *Um* checkpoint c_i^γ *faz parte de uma linha de recuperação* R_F *para um conjunto* F *qualquer de processos falhos se, e somente se,* c_i^γ *pertence a uma das* n *linhas de recuperação para falhas unitárias de processos* $R_{\{p_f\}}$.

Prova: Pelo Teorema 4.3 e a definição de interseção de *checkpoints* globais consistentes (Definição 2.5). □

Neste contexto, verificamos que uma condição necessária para um *checkpoint* ser obsoleto é não fazer parte de qualquer linha de recuperação R_F. Definimos um *checkpoint* com esta propriedade em um determinado instante da execução como um *checkpoint* desnecessário naquele instante. No entanto, pelo menos a princípio, esta condição não exime o *checkpoint* de tomar parte em uma linha de recuperação qualquer em outro momento da execução.

Definição 4.2 ***Checkpoint desnecessário*** — *Um* checkpoint *estável* s_i^γ *é desnecessário em um corte consistente* $\mathcal{C}$ *(aqui chamado* $\mathcal{C}$*-desnecessário) se, e somente se, ele pertence ao corte e não faz parte de nenhuma linha de recuperação possível no padrão de* checkpoints *e mensagens definido por ele.*

Podemos utilizar o Corolário 4.2 para identificar todos os *checkpoints* desnecessários em um padrão de *checkpoints* e mensagens definido por um corte $\mathcal{C}$ na execução da computação distribuída. Um *checkpoint* será desnecessário quando não fizer parte de nenhuma das n linhas de recuperação referentes às falhas unitárias dos processos, conforme apresentamos no Corolário 4.3.

Corolário 4.3 *Um* checkpoint *estável s_i^γ é desnecessário em um corte consistente $\mathcal{C}$ se, e somente se, s_i^γ não pertence a uma das n linhas de recuperação para falhas unitárias de processos $R_{\{p_f\}}$.*

Prova: Pela Definição 4.2 e pelo Corolário 4.2. □

A Figura 4.6 apresenta todos os *checkpoints* desnecessários no padrão de *checkpoints* e mensagens da Figura 4.5(a), representados pelos quadrados não preenchidos. Podemos verificar que tais *checkpoints* não são utilizados em nenhuma das linhas de recuperação para falhas unitárias apresentadas na Figura 4.5 e, portanto, não pertencem a nenhuma linha de recuperação para qualquer conjunto F de processos falhos. Além disso, verificamos que o *checkpoint* v_2 também não faz parte de nenhuma linha de recuperação neste CCP; porém, o conceito de *checkpoint* desnecessário se aplica apenas para *checkpoints* estáveis, de forma que v_2 não pode ser considerado.

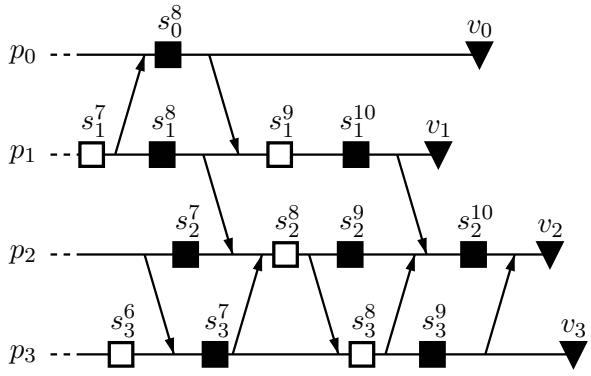

Figura 4.6: *Checkpoints* desnecessários.

4.4.3 Estabilidade da Propriedade

A propriedade de um *checkpoint* ser desnecessário possui um papel importante na caracterização dos *checkpoints* obsoletos, sendo que sua maior importância está no fato de ser uma propriedade estável, conforme será demonstrado no Teorema 4.4. Antes, porém, apresentaremos uma importante característica dos *checkpoints* desnecessários que é a grande razão da estabilidade desta propriedade. Pelo Teorema 4.2, temos que um *checkpoint* estável s_e^ϵ fará parte de uma linha de recuperação $R_{\{p_f\}}$ quando $s_f^{last} \rightsquigarrow c_e^{\epsilon+1} \wedge s_f^{last} \not\rightsquigarrow s_e^\epsilon$, pois neste caso s_e^ϵ será o último *checkpoint* do processo p_e que não é Z-precedido por s_f^{last}. O Lema 4.1 demonstra que todas as Z-precedências deste tipo, capazes de fazer um *checkpoint* $\mathcal{C}$-desnecessário pertencer a uma linha de recuperação presente ou futura, estão completamente contidas e fechadas no corte $\mathcal{C}$.

Lema 4.1 *Se um* checkpoint s_e^ϵ *é* $\mathcal{C}$*-desnecessário, então toda Z-precedência* $s_i^\gamma \rightsquigarrow c_e^{\epsilon+1}$ *tal que* $s_i^\gamma \not\rightsquigarrow s_e^\epsilon$ *pertencerá inteiramente ao corte* $\mathcal{C}$*, juntamente com o* checkpoint $s_i^{\gamma+1}$.

Prova: Se s_e^ϵ for um *checkpoint* inútil ($s_e^\epsilon \rightsquigarrow s_e^\epsilon$) a prova é trivial pois toda precedência $s_i^\gamma \rightsquigarrow c_e^{\epsilon+1}$ implicará na precedência $s_i^\gamma \rightsquigarrow s_e^\epsilon$, o que não satisfaz as condições impostas pelo lema. Consideremos então o caso em que s_e^ϵ é um *checkpoint* útil. Como s_e^ϵ é $\mathcal{C}$-desnecessário, pelo Teorema 4.2 podemos concluir que s_e^ϵ não é o último *checkpoint* estável do processo p_e dentro do corte $\mathcal{C}$, caso contrário ele tomaria parte na linha de recuperação $R_{\{p_e\}}$. Desta forma, temos que $c_e^{\epsilon+1}$ é estável e pertence ao corte $\mathcal{C}$. Além disto, pela definição de Z-precedência, podemos quebrar a relação $s_i^\gamma \rightsquigarrow s_e^{\epsilon+1}$ em uma série de relações de precedência causal entre pares de *checkpoints* da seguinte forma:

$$s_i^\gamma = c_{i_0}^{\gamma_0} \quad (c_{i_0}^{\gamma_0} \rightarrow c_{i_1}^{\gamma_1}), (c_{i_1}^{\gamma_1 - 1} \rightarrow c_{i_2}^{\gamma_2}), \ldots, (c_{i_{p-1}}^{\gamma_{p-1}-1} \rightarrow c_{i_p}^{\gamma_p}) \quad c_{i_p}^{\gamma_p} = s_e^{\epsilon+1}$$

Provaremos por indução no número p de precedências que a Z-precedência $s_i^\gamma \rightsquigarrow s_e^{\epsilon+1}$ e o *checkpoint* $s_i^{\gamma+1}$ devem estar totalmente contidos no corte $\mathcal{C}$.

Hipótese: Se $s_i^\gamma \not\rightsquigarrow s_e^\epsilon$ e $s_i^\gamma \rightsquigarrow s_e^{\epsilon+1}$ por meio de uma seqüência de $p-1$ relações de precedência causal entre pares, então a relação $s_i^\gamma \rightsquigarrow s_e^{\epsilon+1}$ e o *checkpoint* $s_i^{\gamma+1}$ estão totalmente contidos no corte $\mathcal{C}$.

Base da Indução: ($p = 1$) Temos a única relação $s_i^\gamma \rightarrow s_e^{\epsilon+1}$. Como $s_e^{\epsilon+1} \in \mathcal{C}$, pela definição de corte consistente temos que todos os eventos que o precedem causalmente também devem pertencer a $\mathcal{C}$. Sendo assim, a relação $s_i^\gamma \rightarrow s_e^{\epsilon+1}$ deve estar totalmente contida em $\mathcal{C}$, conforme é mostrado na Figura 4.7(a). Como $s_i^\gamma \not\rightsquigarrow s_e^\epsilon$, se s_i^γ fosse o último *checkpoint* estável do processo p_i em $\mathcal{C}$, pelo Teorema 4.2, teríamos que $s_e^\epsilon \in R_{\{p_i\}}$ no corte $\mathcal{C}$, o que é uma contradição com fato de s_e^ϵ ser $\mathcal{C}$-desnecessário. Desta forma, também concluímos que $s_i^{\gamma+1} \in \mathcal{C}$, conforme é mostrado na Figura 4.7(b).

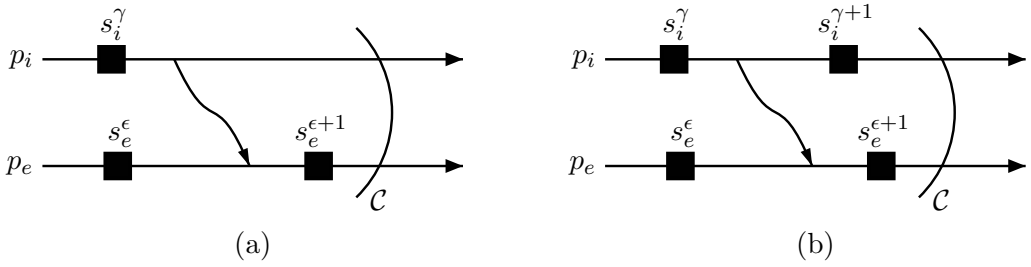

Figura 4.7: Base da indução.

Passo de Indução: ($p > 1$) Podemos dividir a Z-precedência $s_i^\gamma \rightsquigarrow s_e^{\epsilon+1}$ em duas partes: uma dependência causal $s_i^\gamma \rightarrow c_j^{\iota+1}$ e uma Z-precedência $s_j^\iota \rightsquigarrow s_e^{\epsilon+1}$ composta por $p-1$ relações de

precedência causal. Sabemos que $s_j^{\iota} \not\rightsquigarrow s_e^{\epsilon}$, caso contrário, pela definição de Z-precedência, teríamos que $s_i^{\gamma} \rightsquigarrow s_e^{\epsilon}$. Desta maneira, a Hipótese de Indução garante que a Z-precedência $s_j^{\iota} \rightsquigarrow s_e^{\epsilon+1}$ e o *checkpoint* $s_j^{\iota+1}$ estão totalmente contidos no corte $\mathcal{C}$. Além disso, pela definição de corte consistente, temos que todas as precedências causais de $s_j^{\iota+1}$ devem pertencer a $\mathcal{C}$, incluindo a relação $s_i^{\gamma} \rightarrow s_j^{\iota+1}$. Isto prova que a relação $s_i^{\gamma} \rightsquigarrow s_e^{\epsilon+1}$ está totalmente contida no corte $\mathcal{C}$, conforme é mostrado na Figura 4.8(a) onde a curva em formato de Z representa uma relação de Z-precedência. Para provar que $s_i^{\gamma+1}$ também pertence ao corte $\mathcal{C}$ basta utilizar o mesmo raciocínio empregado na base da indução. Como $s_i^{\gamma} \not\rightsquigarrow s_e^{\epsilon}$, se s_i^{γ} fosse o último *checkpoint* estável do processo p_i em $\mathcal{C}$, pelo Teorema 4.2, teríamos que $s_e^{\epsilon} \in R_{\{p_i\}}$ no corte $\mathcal{C}$, o que é uma contradição com fato de s_e^{ϵ} ser $\mathcal{C}$-desnecessário. Desta forma provamos que $s_i^{\gamma+1} \in \mathcal{C}$, conforme é mostrado na Figura 4.8(b). □

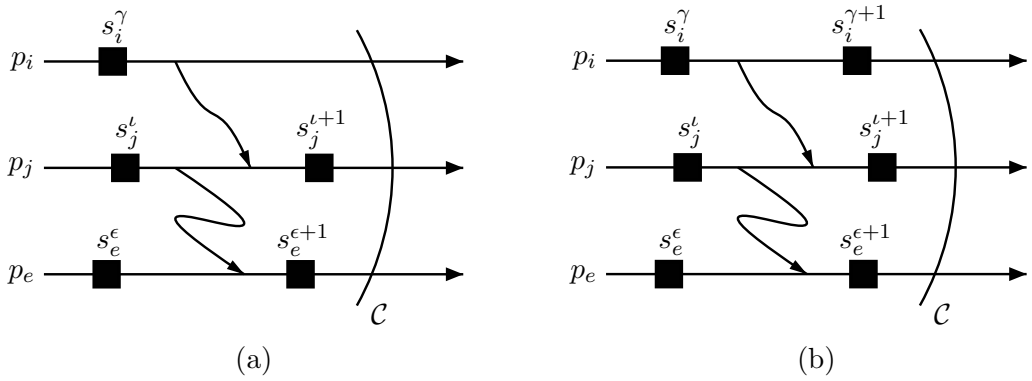

Figura 4.8: Passo da indução.

Com base no Lema 4.1 fica muito fácil provar que um *checkpoint* desnecessário permanecerá desnecessário durante um período de execução normal da computação.

Teorema 4.4 *Se um* checkpoint s_e^{ϵ} *é* $\mathcal{C}$*-desnecessário, ele também será* $\mathcal{L}$*-desnecessário para todo corte consistente* $\mathcal{L}$ *no futuro de* $\mathcal{C}$ *(*$\mathcal{C} \subset \mathcal{L}$*).*

Prova: Vamos supor, por contradição, que s_e^{ϵ} é $\mathcal{C}$-desnecessário e que existe um corte consistente $\mathcal{L}$ no futuro de $\mathcal{C}$ ($\mathcal{C} \subset \mathcal{L}$) no qual s_e^{ϵ} faz parte de uma linha de recuperação. Desta maneira, sabemos pelo Corolário 4.2 que existe um processo p_f tal que $s_f^{last} \rightsquigarrow c_e^{\epsilon+1}$ e $s_f^{last} \not\rightsquigarrow s_e^{\epsilon}$ dentro do corte $\mathcal{L}$. Acontece que o Lema 4.1 demonstra que, como s_e^{ϵ} é $\mathcal{C}$-desnecessário, a Z-precedência $s_f^{last} \rightsquigarrow s_e^{\epsilon+1}$ deve estar totalmente contida no corte $\mathcal{C}$. Mais do que isto, o próximo *checkpoint* estável de p_f após s_f^{last} também deveria pertencer a $\mathcal{C}$ o que representa uma contradição pois s_f^{last} é, por definição, o último *checkpoint* estável de p_f no corte $\mathcal{L}$ conforme apresentamos na Figura 4.9. □

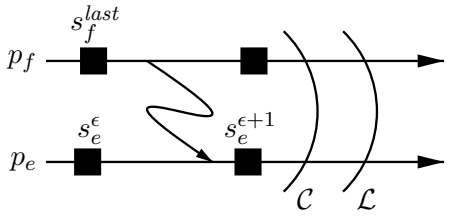

Figura 4.9: Contradição com a definição de s_f^{last}.

4.4.4 *Checkpoints* Desnecessários após um Retrocesso

Os *checkpoints* desnecessários, além de permanecerem nesta condição durante todo um período de execução normal, se mantêm neste estado mesmo depois de um retrocesso. Na realidade, após um retrocesso, um *checkpoint* que seja desnecessário ou será retrocedido e apagado, ou continuará desnecessário. A principal razão deste fato é que todas as Z-precedências dos *checkpoints* estáveis que não são retrocedidos são mantidas durante a recuperação de uma falha.

Lema 4.2 *Se um* checkpoint *estável* s_e^{ϵ} *é Z-precedido por um outro* checkpoint *estável* s_i^{γ} *(*$s_i^{\gamma} \rightsquigarrow s_e^{\epsilon}$*) em um corte* $\mathcal{C}$*, então a Z-precedência* $s_i^{\gamma} \rightsquigarrow s_e^{\epsilon}$ *será mantida caso haja um retrocesso para qualquer linha de recuperação* R_F *em* $\mathcal{C}$ *que não elimine o* checkpoint s_e^{ϵ}.

Prova: Vamos supor, por contradição, que exista uma Z-precedência $s_i^{\gamma} \rightsquigarrow s_e^{\epsilon}$ no corte $\mathcal{C}$ que não é mantida após o retrocesso a uma linha de recuperação R_F que não elimine o *checkpoint* s_e^{ϵ}. Para a Z-precedência ter sido eliminada durante o retrocesso, é necessário que um *checkpoint* c_j^{ι} que faça parte da Z-precedência ($c_j^{\iota} = s_i^{\gamma}$ ou $s_i^{\gamma} \rightsquigarrow c_j^{\iota} \wedge c_j^{\iota-1} \rightsquigarrow s_e^{\epsilon}$) tenha sido retrocedido durante o processo. Portanto, pelos Teoremas 4.2 e 4.3, sabemos que existe um processo $p_f \in F$ tal que $s_f^{last} \rightsquigarrow c_j^{\iota}$. No entanto, como $s_f^{last} \rightsquigarrow c_j^{\iota} \wedge c_j^{\iota-1} \rightsquigarrow s_e^{\epsilon}$, pela definição da relação de Z-precedência podemos verificar que $s_f^{last} \rightsquigarrow s_e^{\epsilon}$, o que implicaria no retrocesso de s_e^{ϵ}. Isto é uma contradição pois definimos que s_e^{ϵ} não era retrocedido. □

São as Z-precedências de um *checkpoint* estável e de seu sucessor que o tornam desnecessário. Se tais precedências são mantidas em um retrocesso, podemos verificar que esta propriedade também se mantém, conforme é apresentado no Teorema 4.5.

Teorema 4.5 *Se um* checkpoint *estável* s_e^{ϵ} *é* $\mathcal{C}$*-desnecessário, ele será retrocedido ou continuará desnecessário após a recuperação de uma falha a partir do corte* $\mathcal{C}$.

Prova: Seja $\mathcal{C}'$ o corte consistente mínimo que contém o *checkpoint* s_e^{ϵ}. Desta forma, sabemos que todos os cortes consistentes que contêm s_e^{ϵ} estão no futuro de $\mathcal{C}'$. Vamos supor,

por contradição, que existe uma linha de recuperação R_F em $\mathcal{C}$ que força a computação a retroceder a um estado consistente no qual s_e^ϵ existe e não é desnecessário, ou seja, é utilizado em alguma linha de recuperação. Como s_e^ϵ é $\mathcal{C}$-desnecessário, conclui-se que R_F não o contém. Portanto, o corte $\mathcal{R}$ definido por R_F deve estar no futuro de $\mathcal{C}'$ e no passado de $\mathcal{C}$ ($\mathcal{C}' \subset \mathcal{R} \subset \mathcal{C}$). Além disso, como s_e^ϵ não é $\mathcal{R}$-desnecessário, pelo Teorema 4.2 e pelo Corolário 4.2 deve haver um processo p_f tal que $s_f^{last} \rightsquigarrow s_e^{\epsilon+1} \wedge s_f^{last} \not\rightsquigarrow s_e^\epsilon$ dentro do corte $\mathcal{R}$. A presença da precedência $s_f^{last} \rightsquigarrow s_e^{\epsilon+1}$ no corte $\mathcal{R}$ implica que o *checkpoint* do processo p_f em R_F é volátil, conforme é mostrado na Figura 4.10; e se ele é volátil, representa o estado do processo p_f no corte $\mathcal{C}$. Também verificamos, pelo Lema 4.2, que $s_f^{last} \not\rightsquigarrow s_e^\epsilon$ no corte $\mathcal{C}$, caso contrário esta precedência também estaria presente no corte $\mathcal{R}$. No entanto, se $s_f^{last} \rightsquigarrow s_e^{\epsilon+1} \wedge s_f^{last} \not\rightsquigarrow s_e^\epsilon$ no corte $\mathcal{C}$, então $s_e^\epsilon \in R_{\{p_f\}}$ e portanto não é $\mathcal{C}$-desnecessário, o que é uma contradição com nossa definição original. □

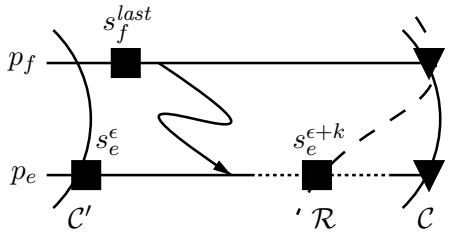

Figura 4.10: Checkpoints desnecessários em um retrocesso.

4.4.5 Relação entre *Checkpoints* Desnecessários e Obsoletos

As seções anteriores apresentaram vários resultados que nos permitem facilmente caracterizar os *checkpoints* obsoletos. Na realidade, o fato de um *checkpoint* desnecessário permanecer desnecessário durante um período de execução normal associado com o fato de que esta propriedade não pode ser perdida durante um retrocesso cria uma relação de equivalência entre *checkpoints* desnecessários e *checkpoints* obsoletos.

Teorema 4.6 *Um* checkpoint *estável s_e^ϵ é obsoleto em um corte consistente $\mathcal{C}$ se, e somente se, ele for $\mathcal{C}$-desnecessário.*

Prova: *Suficiência*($\Leftarrow$): Pelo Teorema 4.4 verificamos que um *checkpoint* estável que se torna desnecessário permanecerá desnecessário durante todo um período de execução normal. Além disso, o Teorema 4.5 nos mostra que um *checkpoint* desnecessário ou é retrocedido ou permanece desnecessário após a recuperação de uma falha. Desta maneira, podemos concluir que um *checkpoint* desnecessário s_e^ϵ permanecerá desnecessário até o final da execução da

computação ou será eliminado durante um retrocesso; portanto s_e^ϵ não poderá fazer parte de nenhuma linha de recuperação futura, sendo considerado obsoleto.

Necessidade($\Rightarrow$): Se um *checkpoint* estável s_e^ϵ não é desnecessário, ele está presente em pelo menos uma linha de recuperação unitária $R_{\{p_f\}}$ em $\mathcal{C}$. Sendo assim, se p_f falhar, s_e^ϵ será utilizado na linha de recuperação, de onde podemos concluir que ele não é obsoleto. □

O Teorema 4.6 pode ser reescrito de uma forma mais direta, baseada apenas nas Z-precedências existentes entre um *checkpoint* e seu sucessor, conforme apresentamos no Corolário 4.4.

Corolário 4.4 *Um* checkpoint *estável s_e^ϵ é obsoleto em um corte consistente $\mathcal{C}$ se, e somente se, não existe um processo p_f tal que $s_f^{last} \rightsquigarrow c_e^{\epsilon+1} \ \wedge \ s_f^{last} \not\rightsquigarrow s_e^\epsilon$ em $\mathcal{C}$.*

Prova: Pelo Corolário 4.3 e pelo Teorema 4.2. □

Como exemplo considere a Figura 4.11, onde todos os *checkpoints* obsoletos estão representados pelos quadrados não preenchidos. Note que a caracterização abrange todas as condições suficientes apresentadas anteriormente. Verificamos que os *checkpoints* s_1^7 e s_3^6 estão antes de um *checkpoint* global consistente composto por *checkpoints* estáveis: $\{s_0^8, s_1^8, s_2^7, s_3^7\}$; além disso, o *checkpoint* s_2^8 é inútil e, portanto, obsoleto, e o intervalo entre *checkpoints* I_1^{10} não apresenta eventos de comunicação, o que torna o *checkpoint* s_1^9 obsoleto. Além de todos estes *checkpoints* identificados por condições suficientes comentadas anteriormente, também identificamos o *checkpoint* s_3^8 como obsoleto.

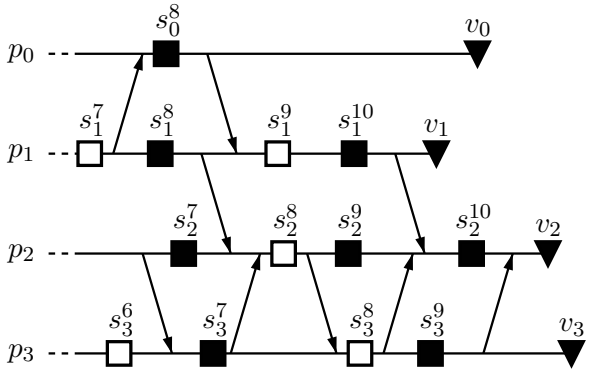

Figura 4.11: *Checkpoints* obsoletos.

4.5 Coleta de Lixo Ótima

Uma vez que obtivemos uma caracterização dos *checkpoints* obsoletos dentro de um padrão de *checkpoints* e mensagens, torna-se possível desenvolver um algoritmo ótimo para coleta

de lixo, capaz de eliminar todos os *checkpoints* obsoletos existentes. Basta capturar o CCP da computação e identificar todos os *checkpoints* estáveis que fazem parte de alguma linha de recuperação $R_{\{p_f\}}$; todos os outros *checkpoints* estáveis serão obsoletos.

4.5.1 Descrição do Algoritmo

Uma idéia muito simples para um algoritmo ótimo para coleta de lixo consiste em uma pequena modificação no algoritmo de coleta ingênua apresentado na Seção 4.3. Durante a primeira fase, o coordenador coleta informações sobre as dependências existentes entre os *checkpoints* dos diferentes processos e monta uma fotografia do padrão de *checkpoints* e mensagens da computação distribuída, possivelmente utilizando um *R-graph*. Depois de fazer isto, basta calcular localmente as n linhas de recuperação para falhas unitárias $R_{\{p_f\}}$ e identificar os *checkpoints* estáveis a serem utilizados em cada uma delas. Finalmente, o coordenador deve enviar a todos os processos o conjunto de *checkpoints* identificados, os quais devem ser mantidos. Ao receber este conjunto, um processo deve identificar e eliminar todos os *checkpoints* locais que não pertencem a ele. A idéia de utilização de tal algoritmo consiste em deixar a aplicação e o protocolo de *checkpointing* executarem normalmente, até que um número excessivo de *checkpoints* armazenados seja percebido. Neste momento, a coleta de lixo é invocada pelo coordenador e todos os *checkpoints* obsoletos são coletados.

O Algoritmo 4.1 apresenta uma versão do algoritmo comentado acima. Assumimos que todos os processos armazenam as dependências de seus *checkpoints* locais em um conjunto DEP_i também armazenado localmente. Este conjunto pode representar as dependências de forma direta ou transitiva conforme foi visto na Seção 2.4.1. De posse de todas as dependências entre os *checkpoints* da computação distribuída, o coordenador consegue facilmente construir um *R-graph* do padrão de *checkpoints* e mensagens. Bhargava e Lian [9] apresentam uma forma eficiente de manter e construir dependências diretas e *R-graphs* da computação.

Um cuidado extra deve ser tomado durante a eliminação de um *checkpoint* caso um mecanismo de captura de dependências diretas esteja sendo utilizado. Se o *checkpoint* obsoleto for anterior a uma linha de recuperação composta por *checkpoints* estáveis, todas as suas dependências podem ser eliminadas sem problemas. No entanto, se o *checkpoint* não satisfizer a condição para coleta ingênua, suas dependências diretas devem ser repassadas para o próximo *checkpoint* que não tenha sido coletado. Além disso, as dependências devem ser modificadas para referirem ao último *checkpoint* não-obsoleto antes do envio da mensagem, caso a dependência original refira-se a um *checkpoint* obsoleto. Todas essas modificações podem ser realizadas localmente uma vez que o conjunto *N_OBSOL* contém todos os *checkpoints* não-obsoletos de todos os processos. Note que a utilização de captura por dependências transitivas não requer este cuidado extra pois, neste caso, um *checkpoint* armazena também uma estrutura que representa todas as suas dependências causais, e não apenas as do último intervalo.

Algoritmo 4.1 Coleta de lixo ótima.

Coordenador

1: *envia REQ para todos* {requisição de dependências}
2: $DEP \leftarrow \emptyset$ {conjunto de *checkpoints* e suas dependências}
3: **for** $i \leftarrow 0$ **to** $n-1$ **do**
4: *recebe* DEP_i *de* p_i {recebe informações de dependências de p_i}
5: $DEP \leftarrow DEP \cup DEP_i$ {e as acumula em DEP}
6: **end for**
7: cria *R-graph RG* a partir de DEP
8: $N\_OBSOL \leftarrow \emptyset$ {conjunto de *checkpoints* não obsoletos}
9: **for** $i \leftarrow 0$ **to** $n-1$ **do**
10: calcula $R_{\{p_i\}}$ a partir de RG {linhas de recuperação para falhas unitárias}
11: $N\_OBSOL \leftarrow N\_OBSOL \cup R_{\{p_i\}}$ {representam *checkpoints* não obsoletos}
12: **end for**
13: retira *checkpoints* voláteis de N_OBSOL
14: *envia N_OBSOL para todos* {propaga informações a todos os processos}

Processo p_i ao receber *REQ*

1: *envia* DEP_i *para coordenador* {envia dependências locais ao coordenador}
2: *bloqueia até* receber N_OBSOL do coordenador {e fica bloqueado aguardando resposta}
3: **for all** s_i^γ armazenado **do**
4: **if** $s_i^\gamma \notin N\_OBSOL$ **then**
5: elimina s_i^γ {coleta todos os *checkpoints* obsoletos}
6: **end if**
7: **end for**
8: continua execução normal

Por fim, o motivo pelo qual o conjunto identificado e propagado pelo coordenador é o de *checkpoints* não-obsoletos é que este conjunto apresenta um limite máximo para o seu tamanho, conforme veremos na próxima seção. Sendo assim, para um número elevado de *checkpoints* existente estaremos economizando no tamanho máximo das informações propagadas se usarmos os *checkpoints* utilizáveis ao invés dos obsoletos.

4.5.2 Limite Máximo de *Checkpoints* Mantidos

Conforme podemos facilmente verificar, esta abordagem possui algumas semelhanças com a coleta de lixo ingênua apresentada anteriormente. Sua principal desvantagem é a necessidade de sincronização e mensagens de controle. Porém, diferentemente da coleta ingênua, o algoritmo ótimo possui um limite máximo para número de *checkpoints* não coletados durante a sua execução. Desta forma, pode-se desenvolver um sistema com limite para a quantidade máxima necessária de armazenamento estável.

Sabemos que os *checkpoints* não-obsoletos são aqueles que pertencem a pelo menos uma das linhas de recuperação para falhas unitárias dos processos. No pior caso, cada processo

poderá ter que manter um *checkpoint* estável para cada uma destas linhas de recuperação, o que resulta em um limite de n *checkpoints* por processo. Como existem n processos, temos um limite máximo de n^2 *checkpoints* não-obsoletos. No entanto, este cenário global pessimista não pode ocorrer na prática o que faz diminuir aproximadamente pela metade este limite máximo global conforme é apresentado no Teorema 4.7.

Teorema 4.7 *O número máximo de* checkpoints *não-obsoletos em um padrão de* checkpoints *e mensagens é menor ou igual a* $n(n+1)/2$.

Prova: Provaremos este teorema por indução no número de processos da seguinte forma:

Hipótese: O número máximo de *checkpoints* não-obsoletos em um CCP com $n-1$ processos é menor ou igual a $(n-1)n/2$.

Base da Indução: $(n = 1)$ Com apenas um processo, somente o último *checkpoint* gravado não é obsoleto, então temos que verificar a seguinte inequação:

$$\begin{array}{rcl} 1 & \leq & 1(2)/2 \\ 1 & \leq & 1 \end{array} \tag{4.1}$$

Passo de Indução: $(n > 1)$ Considere um CCP com $n-1$ processos $(p_0, \ldots, p_{n-2})$. Pela hipótese de indução, sabemos que o número máximo de *checkpoints* não-obsoletos é menor ou igual a $(n-1)n/2$. Ao acrescentarmos um novo processo p_{n-1} certamente teremos mais um *checkpoint* não-obsoleto, correspondente ao *checkpoint* de p_{n-1} em $R_{\{p_{n-1}\}}$. Além disso, para cada processo p_i $(i < n-1)$, temos a possibilidade de inserção de um *checkpoint* estável de p_{n-1} em $R_{\{p_i\}}$ e de um *checkpoint* estável de p_i em $R_{\{p_{n-1}\}}$, o que resulta em mais dois *checkpoints* não-obsoletos para cada dupla (p_{n-1}, p_i). Na verdade, o limite de n^2 é obtido quando as duas opções são consideradas juntamente. No entanto, podemos verificar que isto é impossível de acontecer.

Sejam s_{n-1}^{α} o *checkpoint* de p_{n-1} em $R_{\{p_{n-1}\}}$ e s_i^{β} o *checkpoint* de p_i em $R_{\{p_i\}}$. Pelo Teorema 4.2, verificamos que estes são os últimos *checkpoints* estáveis úteis destes processos. Um *checkpoint* estável de p_{n-1} em $R_{\{p_i\}}$ certamente estaria antes de s_{n-1}^{α} e, da mesma forma, um *checkpoint* estável de p_i em $R_{\{p_{n-1}\}}$ estaria antes de s_i^{β}, conforme é mostrado na Figura 4.12. Neste caso porém, também pelo Teorema 4.2, teríamos que $s_{n-1}^{last} \rightsquigarrow s_i^{last} \rightsquigarrow s_{n-1}^{\alpha}$, o que é uma contradição pois $s_{n-1}^{last} \not\rightsquigarrow s_{n-1}^{\alpha}$. Sendo assim, não é possível aplicar as duas considerações ao mesmo tempo. Portanto, para cada dupla (p_{n-1}, p_i) ou podemos incluir um *checkpoint* estável de p_{n-1} em $R_{\{p_i\}}$ ou um *checkpoint* estável de p_i em $R_{\{p_{n-1}\}}$, o que resulta em no máximo um *checkpoint* não-obsoleto para cada uma das $n-1$ duplas existentes. Ao somarmos os valores obtidos, verificamos o seguinte limite para o número de *checkpoints* obsoletos em um CCP com n processos:

$$\begin{array}{rcl} & \leq & (n-1)n/2 + 1 + (n-1) \\ & \leq & n(n+1)/2 \end{array} \tag{4.2}$$

□

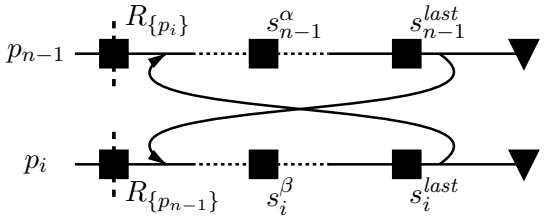

Figura 4.12: Ocorrência das duas condições simultaneamente.

Além disso, o limite apresentado é justo pois é possível construir um padrão de *checkpoints* e mensagens que o atinja, conforme é demonstrado no Teorema 4.8.

Teorema 4.8 *Existe um padrão de* checkpoints *e mensagens para um conjunto de n processos com* $n(n+1)/2$ checkpoints *não-obsoletos.*

Prova: Considere o padrão de *checkpoints* e mensagens definido recursivamente da seguinte maneira:

Base: (p_0) O processo p_0 apresenta um único *checkpoint* estável e envia uma única mensagem para p_1 imediatamente após sua gravação.

Passo Recursivo: $(p_i,\ i > 0)$ O processo p_i entrega as mensagens de p_{i-1} na mesma ordem em que elas foram enviadas. Além disso, um *checkpoint* é gravado imediatamente antes e uma mensagem é enviada para p_{i+1} imediatamente depois da entrega de cada mensagem proveniente de p_{i-1}. Um último *checkpoint* é gravado após todas as mensagens de p_{i-1} terem sido entregues e imediatamente depois uma nova mensagem é enviada para p_{i+1}.

Fechamento: Um processo, *checkpoint* ou mensagem pertence ao CCP somente se puder ser obtido a partir do caso base por meio de um número finito de aplicações do passo recursivo.

A Figura 4.13 apresenta um exemplo deste padrão para 4 processos. Conforme podemos facilmente verificar pelo Teorema 4.2, a linha de recuperação $R_{\{p_i\}}$ é composta pelo *checkpoint* volátil dos processos $\{p_0, \ldots, p_{i-1}\}$ e pelo $(i+1)$-ésimo *checkpoint* estável dos processos $\{p_i, \ldots, p_{n-1}\}$, ou seja,

$$R_{\{p_i\}} = \bigcup_{j=0}^{i-1} \{\ v_j\ \} \ \cup \ \bigcup_{j=i}^{n-1} \{\ s_j^{i+1}\ \}. \tag{4.3}$$

Desta forma, todos os *checkpoints* estáveis do CCP são não-obsoletos. Pela definição recursiva, verificamos que cada processo p_i apresenta $i+1$ *checkpoints* estáveis e, portanto, a quantidade total é dada por $\sum_{i=1}^{n} i = n(n+1)/2$, conforme queríamos demonstrar. Além disto, note que este padrão é SZPF, a classe mais restrita de padrões de *checkpoints* e mensagens conforme foi visto no Capítulo 3, de forma que o limite estabelecido vale para todos os protocolos para *checkpointing* assíncronos e quase-síncronos. □

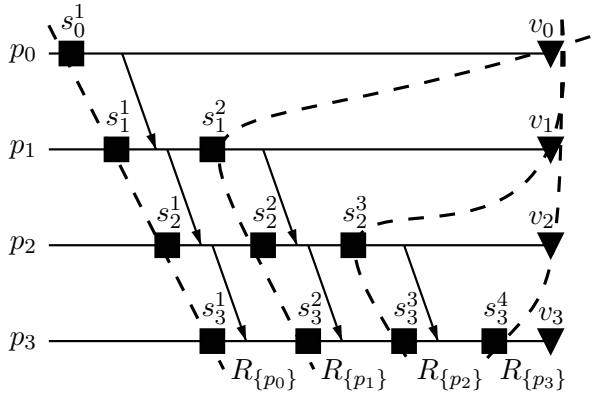

Figura 4.13: Pior caso para o número de *checkpoints* não-obsoletos.

Portanto, como todos os *checkpoints* obsoletos são coletados durante a execução do algoritmo ótimo, concluímos que o número de *checkpoints* mantidos após a coleta é, no máximo, $n(n+1)/2$. Note porém, que o limite máximo de *checkpoints* mantidos em um único processo continua igual a n, a exemplo do processo p_3 da Figura 4.13. Sendo assim, é necessário que cada processo tenha espaço em meio estável para armazenar pelo menos estes n *checkpoints*. Mais do que isto, para minimizar as perdas devido à sincronização realizada pela coleta de lixo, é importante ter um limite para armazenamento algumas vezes maior do que n *checkpoints*, de forma a oferecer uma boa folga para a aplicação executar antes de invocar a coleta de lixo novamente. Um processo pode acioná-la quando seu armazenamento estável estiver muito utilizado com a garantia de que restarão no máximo n *checkpoints* armazenados após a sua execução.

4.5.3 Análise e Conclusão

Assim como a coleta ingênua, o algoritmo ótimo utiliza informações globais referentes ao padrão de *checkpoints* e mensagens da computação e requer, portanto, a troca de mensagens de controle. Especificamente, para a versão que apresentamos há um total de $3n$ mensagens trocadas. Esta característica não é desejável em protocolos para *checkpointing* assíncronos e quase-síncronos. No entanto, dentro de nosso conhecimento, o único algoritmo para coleta de lixo que não se utiliza de mensagens de controle é aquele que apresentamos no Capítulo 5 e, portanto, a presença de tais mensagens nunca chegou a ser colocada como uma característica negativa para a coleta de lixo ótima pois também estava presente nos outros algoritmos existentes. Na realidade, Manivannan sugere uma coleta de lixo que não se utiliza de mensagens de controle mas depende fortemente da existência de sincronismo de relógio entre os processos e de uma freqüência bem definida para a seleção dos *checkpoints* básicos [26], condições que

consideramos muito acima das fornecidas pelo modelo usualmente adotado dentro de *checkpointing* e recuperação de falhas por retrocesso de estado, o que dificultaria sua utilização na prática.

A mensagem *REQ* enviada pelo coordenador no início do algoritmo tem tamanho constante. No entanto, o tamanho das mensagens DEP_i e *N_OBSOL* depende das estruturas de dados enviadas em seu conteúdo. Na seção anterior, vimos que o conjunto de *checkpoints* não-obsoletos possui um tamanho máximo de $n(n+1)/2$, o que limita o tamanho da mensagem *N_OBSOL*. Porém, o tamanho da mensagem DEP_i depende do número de *checkpoints* e dependências criados desde a última coleta de lixo, o que não pode ser limitado pois depende da execução da aplicação. Sendo assim, se o volume de informações for elevado, pode não ser possível enviar toda a informação em uma única mensagem (caso o seu tamanho máximo seja limitado), o que aumentaria ainda mais o número de mensagens de controle.

O algoritmo utilizado pelo coordenador para o cálculo da linha de recuperação consiste de uma busca simples sobre um *R-graph* e pode ser implementada de forma a executar em tempo $O(n + m)$, onde n representa o número de processos e m representa o número de mensagens no CCP. Sendo assim, como são executadas n buscas, temos uma complexidade total de tempo $O(n(n+m))$. A situação se complica com o fato da aplicação inteira ter que ficar bloqueada durante a coleta de lixo. Esta sincronização é o maior dos problemas do algoritmo ótimo. Em sistemas onde o meio de armazenamento estável não possui uma latência para gravação muito alta, pode ser mais eficiente executar um protocolo para *checkpointing* síncrono periodicamente ao invés do algoritmo ótimo para coleta de lixo, com a vantagem de manter apenas n *checkpoints* armazenados no final da sua execução.

Algumas estratégias podem ser adotadas para evitar o bloqueio dos processos causado pela coleta de lixo ótima, como a utilização de um protocolo não-bloqueante [11] para capturar de forma consistente as dependências entre os *checkpoints* ou o desbloqueio antecipado dos processos, logo após o coordenador receber as suas dependências mas antes do cálculo do conjunto *N_OBSOL*. Porém, o custo dessas estratégias é um aumento considerável no número de mensagens de controle propagadas. Além disso, nestes casos, deve-se tomar o cuidado de não eliminar os *checkpoints* que eventualmente tenham sido gravados após um processo ter enviado suas informações de coleta de lixo para o coordenador pois tais *checkpoints* não fazem parte da fotografia do sistema utilizada para a identificação dos *checkpoints* obsoletos.

O algoritmo ótimo foi apresentado originalmente por Wang, Chung e Fuchs [37, 38]. Entretanto, apesar do algoritmo final para coleta de lixo ser praticamente o mesmo, a abordagem original caracterizava a linha de recuperação e os *checkpoints* obsoletos baseada em estruturas de dados gerais para armazenamento de dependências entre *checkpoints*, como *R-graphs*, e não em Z-precedência conforme fazemos nesta dissertação. A principal vantagem de nossa abordagem é um melhor entendimento das condições que levam um *checkpoint* a fazer parte de uma linha de recuperação ou se tornar obsoleto dentro de um padrão de *checkpoints*

e mensagens baseado nas precedências existentes. Como tais precedências são restringidas pelos protocolos para *checkpointing* quase-síncronos, conforme vimos no Capítulo 3, acreditamos que nossa abordagem possa ser utilizada para desenvolver algoritmos de coleta de lixo eficientes para as diversas classes de protocolos existentes.

Capítulo 5

Coleta de Lixo para Padrões RDT

Padrões de *checkpoints* e mensagens que satisfazem a propriedade RDT apresentam apenas dependências causais entre os seus *checkpoints*, ou seja, estão livres dos caminhos-Z que impedem a captura das dependências durante a execução da computação. Esta característica simplifica a determinação da linha de recuperação, o que pode facilitar o mecanismo de recuperação por retrocesso [36] ou mesmo a coleta de lixo, conforme apresentaremos neste capítulo.

Primeiramente, mostramos como é possível simplificar a determinação da linha de recuperação em padrões RDT e as vantagens que esta simplificação proporciona aos algoritmos de coleta ingênua e ótima em tais padrões. Na seqüência, exploramos a caracterização dos *checkpoints* obsoletos apresentada no Capítulo 4 e relaxamos nossas considerações sobre o conhecimento do estado global da computação de forma a criar uma nova condição suficiente para coleta de lixo.

Mesmo não identificando todos os *checkpoints* obsoletos, esta nova condição apresenta um limite máximo para o número de *checkpoints* não identificados. Desta maneira, conseguimos desenvolver um algoritmo para coleta de lixo em padrões RDT com características inéditas, tais como coleta *on-line* durante a execução da computação, ausência de mensagens de controle e limite máximo de n *checkpoints* armazenados por processo em qualquer momento da execução da computação. Este novo algoritmo foi batizado com o nome de RDT-LGC[1].

5.1 Linha de Recuperação em Padrões RDT

O fato de só existirem Z-precedências causais em um padrão de *checkpoints* e mensagens que satisfaça a propriedade RDT permite simplificar o Teorema 4.2, que caracteriza uma a linha de recuperação na falha de um único processo p_f. Obtemos, então, o Teorema 5.1.

[1]LGC é o acrônimo de *Local Garbage Collection*.

Teorema 5.1 *Dados um padrão de* checkpoints *e mensagens RDT e um processo falho p_f, a linha de recuperação $R_{\{p_f\}}$ é determinada por:*

$$R_{\{p_f\}} = \bigcup_{i=0}^{n-1} \{c_i^{\max(k)} | s_f^{last} \not\rightarrow c_i^k\}$$

Prova: Pelo Teorema 4.2 e a Definição 3.2. □

Uma outra característica importante sobre a linha de recuperação e que se vale não só para padrões RDT, como também para qualquer padrão ZCF é o fato de que o último *checkpoint* estável de um processo p_f sempre pertence a $R_{\{p_f\}}$, conforme apresentamos no Teorema 5.2.

Teorema 5.2 *Dados um padrão de* checkpoints *e mensagens ZCF e um processo falho p_f, a linha de recuperação $R_{\{p_f\}}$ contém o* checkpoint *estável s_f^{last}.*

Prova: O Teorema 4.2 explica que o *checkpoint* de p_f que faz parte da linha de recuperação $R_{\{p_f\}}$ é o *checkpoint* c_f^ι mais recente tal que $s_f^{last} \not\rightsquigarrow c_f^\iota$. Sabemos que $s_f^{last} \rightarrow v_f$ e que $s_f^{last} \not\rightsquigarrow s_f^{last}$ pela definição de padrão ZCF. Sendo assim, s_f^{last} é o *checkpoint* mais recente de p_f que não é Z-precedido por s_f^{last} e, portanto, pertence a $R_{\{p_f\}}$. □

5.2 Coleta de Lixo Ingênua para Padrões RDT

A coleta de lixo ingênua pode ser otimizada para protocolos quase-síncronos que satisfazem a propriedade RDT, de forma que não seja necessário coordenar ou suspender os processos durante a sua execução. De acordo com a caracterização de linha de recuperação unitária para padrões RDT descrita no Teorema 5.1 e a caracterização de linha de recuperação geral apresentada no Teorema 4.3, temos que o *checkpoint* de um processo que faz parte de R_P em um padrão RDT é aquele mais recente que não depende causalmente de nenhum *checkpoint* s_f^{last}. Como as dependências causais podem ser capturadas durante a execução por meio de relógios vetoriais ou vetores de dependências, se um processo souber o índice do último *checkpoint* estável s_f^{last} de todos os processos, ele poderá executar a coleta de lixo ingênua de maneira autônoma, eliminando todos os *checkpoints* anteriores àquele necessário para compor a linha de recuperação de uma falha total.

Considere inicialmente que cada processo tem acesso a um vetor UC (Último *Checkpoint*) de n elementos, um para cada processo da aplicação, onde cada elemento contém o índice do último *checkpoint* estável daquele processo ($UC[j] = last\_s(j)$). Além disso, cada processo deve manter e propagar um vetor de dependências DV, de forma que cada *checkpoint* tenha um vetor DV associado, contendo o índice do último intervalo entre *checkpoints* de cada processo do qual depende causalmente. Conforme já dissemos, o *checkpoint* de um processo

p_i que faz parte da linha de recuperação de uma falha total é o *checkpoint* c_i^γ mais recente que não depende causalmente de nenhum *checkpoint* s_f^{last} para todos os valores $0 \leq f \leq n$. Traduzindo esta condição em vetores de dependências, procuramos pelo *checkpoint* c_i^γ tal que $UC[f] \geq DV(c_i^\gamma)[f]$ para todos os valores de f. Depois de identificá-lo, basta eliminar todos os anteriores a ele. A Figura 5.1 apresenta um exemplo de como este algoritmo funciona. Os quadrados não-preenchidos indicam os *checkpoints* obsoletos. Note que a linha de recuperação R_P pode ser identificada por meio dos vetores de dependências dos *checkpoints* conforme descrevemos acima.

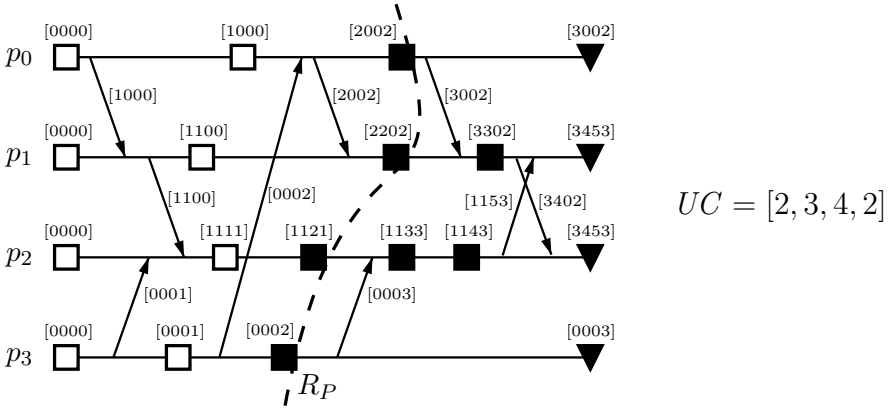

Figura 5.1: Coleta de lixo ingênua para padrões RDT.

Como é impossível manter um vetor UC com informações de todos os processos, cada processo p_i deve manter seu próprio vetor UC_i. Neste caso, pode acontecer de p_i demorar mais a receber a informação de que um processo p_j gravou um novo *checkpoint* estável. Se isso acontecer, p_i apenas demorará mais para avançar com o algoritmo de coleta de lixo, sem eliminar um *checkpoint* que não seja obsoleto, uma vez que cada entrada do vetor UC_i será atualizada ao longo do tempo de forma monotonicamente crescente. A atualização do vetor UC_i pode ser feita de várias maneiras. Um processo pode avisar os outros toda vez que gravar um *checkpoint* ou os processos podem trocar seus vetores periodicamente. Este algoritmo foi apresentado originalmente no contexto de *message logging* otimista [32] mas, conforme apresentamos, aplica-se muito bem a protocolos quase-síncronos RDT.

5.3 Coleta de Lixo Ótima para Padrões RDT

Assim como a determinação da linha de recuperação, a caracterização de *checkpoints* obsoletos (Corolário 4.4) pode ser simplificada em padrões que satisfaçam a propriedade RDT. Em

tais padrões um processo p_i precisa manter apenas o *checkpoint* estável mais recente que não é precedido causalmente por s_f^{last} para todo processo p_f tal que $s_f^{last} \rightarrow v_i$. Todos os outros *checkpoints* estáveis de p_i são obsoletos e podem ser eliminados, conforme apresentamos no Corolário 5.1. Claramente podemos ver que o *checkpoint* s_i^{last} de p_i não é obsoleto uma vez que $s_i^{last} \rightarrow v_i \wedge s_i^{last} \not\rightarrow s_i^{last}$.

Corolário 5.1 *Um* checkpoint s_e^{ϵ} *é obsoleto em um padrão de* checkpoints *e mensagens que satisfaz a propriedade RDT se, e somente se, não existe um processo* p_f *tal que*

$$s_f^{last} \rightarrow c_e^{\epsilon+1} \ \wedge \ s_f^{last} \not\rightarrow s_e^{\epsilon}.$$

Prova: A prova deriva diretamente do Corolário 4.4 e da definição da propriedade RDT. □

Se os processos mantiverem e propagarem um vetor de dependências DV podemos reescrever o Corolário 5.1 de outra maneira, com sua condição baseada nos vetores de dependências atribuídos aos *checkpoints* e o índice do último *checkpoint* estável de cada processo $p_f \in P$, identificado pela função $last\_s(f)$. Com esta nova condição, se os processos tiverem acesso aos valores da função $last\_s(f)$, eles poderão realizar a coleta de lixo localmente, sem que a identificação dos *checkpoints* obsoletos tenha que ser feita em um coordenador externo.

Corolário 5.2 *Um* checkpoint s_e^{ϵ} *é obsoleto em um padrão de* checkpoints *e mensagens que satisfaz a propriedade RDT se, e somente se, não existe um processo* p_f *tal que*

$$last\_s(f) < DV(c_e^{\epsilon+1})[f] \ \wedge \ last\_s(f) \geq DV(s_e^{\epsilon})[f].$$

Prova: Pelo Corolário 5.1 e a definição de vetores de dependências. □

Infelizmente, a função $last\_s(f)$ representa uma informação global da computação distribuída e é necessário sincronizar os processos de alguma maneira para obtê-la de forma consistente. Sendo assim, o Algoritmo 5.1 apresenta um possível algoritmo de coleta de lixo ótima para padrões RDT. Inicialmente, o coordenador envia requisições para todos os processos, pedindo a eles o índice de seu último *checkpoint* estável. Cada processo responde a esta requisição e se mantém bloqueado para manter a informação consistente. Ao receber a resposta de todos os processos o coordenador constrói um vetor UC, onde $UC[j] = last\_s(j)$, e o propaga a todos os processos da aplicação. Finalmente, com esta informação global, cada processo pode realizar a coleta de lixo ótima com base na condição apresentada no Corolário 5.2.

O Algoritmo 5.1 é muito semelhante ao Algoritmo 4.1 mas possui algumas vantagens importantes. Primeiramente, a informação enviada pelos processos ao coordenador consiste apenas de um inteiro. Na realidade, para um processo p_i o valor enviado corresponde a

Algoritmo 5.1 Coleta de lixo ótima para padrões RDT.

Coordenador

1: *envia REQ para todos* {requisição do índice do último *checkpoint*}
2: **for** $i \leftarrow 0$ **to** $n-1$ **do**
3: *recebe last_s*(i) *de* p_i {recebe índice do último *checkpoint* de p_i}
4: $UC[i] \leftarrow$ *last_s*(i) {e atribui ao índice i do vetor UC}
5: **end for**
6: *envia UC para todos* {propaga UC para todos os processos}

Processo p_i ao receber *REQ*

1: *envia last_s*(i) *para coordenador* {envia índice do último *checkpoint* ao coordenador}
2: *bloqueia até* receber UC do coordenador {e fica bloqueado aguardando resposta}
3: **for all** s_i^γ armazenado **do**
4: **if** $\nexists f \mid (UC[f] < DV(c_i^{\gamma+1})[f] \ \wedge \ UC[f] \geq DV(s_i^\gamma)[f])$ **then**
5: elimina s_i^γ {coleta todos os *checkpoints* obsoletos}
6: **end if**
7: **end for**
8: continua execução normal

$DV[i]-1$. Outra importante vantagem é a determinação dos *checkpoints* obsoletos localmente por parte dos processos da aplicação. Desta forma, não há um bloqueio global da computação durante a execução desta tarefa, o que tende a acelerar o processo. Finalmente, a complexidade de tempo da coleta ficou dividida entre os vários processos. Se o limite para o número de *checkpoints* armazenados for kn, para uma constante qualquer k, temos que um processo coleta todos os seus *checkpoints* obsoletos em tempo $O(n^2)$, uma vez que a linha 4 executa em tempo $O(n)$.

A Figura 5.2 apresenta um CCP que satisfaz a propriedade RDT. Para cada *checkpoint* apresentamos, além de sua identificação, o conteúdo do seu vetor de dependências. Se o Algoritmo 5.1 fosse aplicado sobre este padrão, teríamos que $UC = [0, 3, 3]$ e os *checkpoints* representados pelos quadrados não preenchidos seriam coletados, conforme a condição apresentada no Corolário 5.2. Note que os conjuntos de *checkpoints* estáveis $\{s_0^0, s_1^0, s_2^0\}$, $\{s_1^3, s_2^3\}$ e $\{s_2^3\}$ compõem $R_{\{p_0\}}$, $R_{\{p_1\}}$ e $R_{\{p_2\}}$ respectivamente e, por este motivo, não devem ser eliminados pelo algoritmo.

5.4 Condição Suficiente para Coleta em Padrões RDT

Podemos relaxar um pouco a condição necessária e suficiente obtida na seção anterior de forma a conseguir uma condição apenas suficiente, mas com vantagens semelhantes às da coleta ótima. Seja *last_k*$_i(j)$ o índice do último *checkpoint* estável do processo p_j conhecido pelo processo p_i, isto é, o último *checkpoint* de p_j que precede causalmente o *checkpoint* volátil

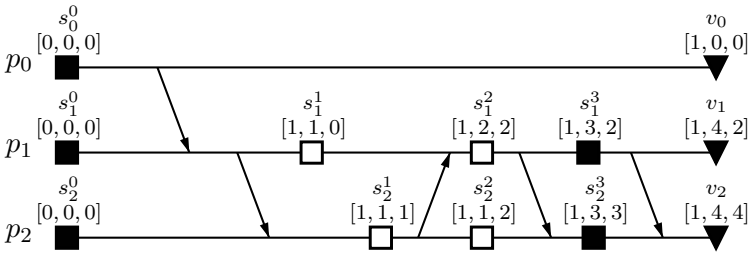

Figura 5.2: Checkpoints obsoletos em um padrão RDT.

atual de p_i. Caso nenhum *checkpoint* de p_j preceda causalmente o *checkpoint* volátil atual de p_i, $last\_k_i(j)$ possui valor -1. Podemos verificar facilmente que $last\_k_i(j) \leq last\_s(j)$. Sendo assim, se $last\_k_i(j) < last\_s(j)$, pelo Teorema 5.1, sabemos que $v_i \in R_{\{p_j\}}$; caso contrário ($last\_k_i(j) = last\_s(j)$) o último *checkpoint* estável de p_i que não seja causalmente precedido por $s_j^{last\_k_i(j)}$ pertencerá à linha de recuperação $R_{\{p_j\}}$ e não poderá ser coletado. Portanto, se deixarmos de coletar apenas o último *checkpoint* estável do processo p_i não precedido causalmente pelo *checkpoint* $s_j^{last\_k_i(j)}$ para todo processo p_j tal que $last\_k_i(j) \geq 0$, não eliminaremos nenhum *checkpoint* que não seja obsoleto, como podemos verificar no Teorema 5.3.

Teorema 5.3 *Um* checkpoint s_e^ϵ *é obsoleto em um padrão de* checkpoints *e mensagens que satisfaz a propriedade RDT se não existe um processo* p_f *tal que*

$$last\_k_e(f) \geq 0 \ \wedge \ s_f^{last\_k_e(f)} \rightarrow c_e^{\epsilon+1} \ \wedge \ s_f^{last\_k_e(f)} \not\rightarrow s_e^\epsilon$$

Prova: Vamos supor, por contradição, que exista um *checkpoint* estável s_e^ϵ que satisfaça esta condição mas não seja obsoleto. Pelo Corolário 5.1, existe um processo p_f tal que $s_f^{last} \rightarrow c_e^{\epsilon+1} \wedge s_f^{last} \not\rightarrow s_e^\epsilon$. Como $s_f^{last} \rightarrow c_e^{\epsilon+1}$, p_e conhece s_f^{last} e $last\_s(f) = last\_k_e(f)$. Portanto, $last\_k_e(f) \geq 0 \ \wedge \ s_f^{last\_k_e(f)} \rightarrow c_e^{\epsilon+1} \ \wedge \ s_f^{last\_k_e(f)} \not\rightarrow s_e^\epsilon$, o que contradiz nossa hipótese inicial. □

Em outras palavras, esta condição diz que um processo p_e pode manter apenas o *checkpoint* estável anterior à criação da última precedência causal proveniente de um processo p_f para todos os processos da aplicação. Desta maneira, um processo que realize a coleta de lixo baseada nesta nova condição suficiente manterá em meio estável no máximo n *checkpoints*, uma vez que, na pior das hipóteses, poderá ser obrigado a reter um *checkpoint* estável diferente para cada processo existente na aplicação.

Além disso, se os processos mantiverem e propagarem um vetor de dependências DV, o valor da função $last\_k_e(f)$ pode ser obtido por $DV(v_e)[f] - 1$. Sendo assim, da mesma forma que fizemos com o Corolário 5.1 na seção anterior, podemos reescrever o Teorema 5.3 com sua condição baseada apenas nos vetores de dependências atribuídos aos *checkpoints* locais de p_e conforme apresentamos no Corolário 5.3.

Corolário 5.3 *Um* checkpoint s_e^ϵ *é obsoleto em um padrão de* checkpoints *e mensagens que satisfaz a propriedade RDT se não existe um processo* p_f *tal que*

$$DV(v_e)[f] > 0 \ \wedge \ DV(v_e)[f] \leq DV(c_e^{\epsilon+1})[f] \ \wedge \ DV(v_e)[f] > DV(s_e^\epsilon)[f].$$

Prova: Pelo Teorema 5.3 e a definição de vetores de dependências. □

Podemos simplificar ainda mais a condição apresentada no Corolário 5.3 se levarmos em consideração algumas propriedades relativas à inicialização e propagação de um vetor de dependências, conforme apresentamos no Corolário 5.4.

Corolário 5.4 *Um* checkpoint s_e^ϵ *é obsoleto em um padrão de* checkpoints *e mensagens que satisfaz a propriedade RDT se não existe um processo* p_f *tal que*

$$DV(v_e)[f] = DV(c_e^{\epsilon+1})[f] \ \wedge \ DV(v_e)[f] > DV(s_e^\epsilon)[f].$$

Prova: Como uma entrada $DV[i]$ é atualizada de forma monotonicamente crescente dentro de um processo, é impossível que tenhamos $DV(v_e)[f] < DV(c_e^{\epsilon+1})[f]$, o que explica a substituição do $\leq$ por $=$ na segunda condição do Corolário 5.3. Além disso, como todas as entradas de DV são inicializadas com valor 0, se a primeira condição do Corolário 5.3 ($DV(v_e)[f] > 0$) não for verdadeira, a última ($DV(v_e)[f] > DV(s_e^\epsilon)[f]$) também não será, de forma que a primeira se torna supérflua e pode ser retirada. □

Note que a condição apresentada no Corolário 5.4 é baseada totalmente em dados locais do processo p_e. Desta maneira, cada processo pode realizar coleta de lixo localmente, sem precisar sincronizar ou mesmo trocar mensagens de controle com os demais e, acima de tudo, com a certeza de que no máximo n *checkpoints* restarão armazenados. Este limite máximo é ótimo por processo mas gera um limite máximo global de n^2 *checkpoints* que é maior que o limite ótimo global apresentado na Seção 4.5.2.

O Algoritmo 5.2 utiliza esta condição para realizar coleta de lixo localmente. Note que não há troca de mensagens de controle ou qualquer outra forma de sincronização. O processo apenas pára temporariamente sua execução para testar a condição do Corolário 5.4 em todos os seus *checkpoints* estáveis armazenados. A linha 2 executa em tempo $O(n)$ o que implica em uma complexidade de tempo $O(n^2)$ se o limite para o número de *checkpoints* armazenados for kn, para uma constante qualquer k.

A Figura 5.3 representa o mesmo padrão RDT apresentado anteriormente na Figura 5.2. Se o Algoritmo 5.2 fosse executado em todos os processos para este padrão, todos os *checkpoints* representados por quadrados não preenchidos seriam considerados obsoletos e coletados. Note que a condição é suficiente mas não necessária, uma vez que o *checkpoint* s_1^1 é obsoleto mas não foi identificado pelo algoritmo.

Algoritmo 5.2 Coleta de lixo local para padrões RDT.

Processo p_i ao identificar uso excessivo do meio estável

1: **for all** s_i^γ armazenado **do**
2: **if** $\nexists f \mid (DV(v_i)[f] = DV(c_i^{\gamma+1})[f] \ \wedge \ DV(v_i)[f] > DV(s_i^\gamma)[f])$ **then**
3: elimina s_i^γ {*checkpoint* obsoleto}
4: **end if**
5: **end for**
6: continua execução normal

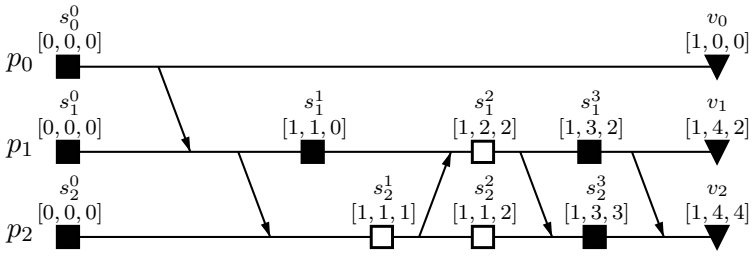

Figura 5.3: Coleta de lixo local em um padrão RDT.

5.5 Descrição do Algoritmo RDT-LGC

É possível, ainda, melhorar o desempenho da coleta de lixo local se os *checkpoints* forem coletados no exato momento em que se tornam obsoletos pela condição apresentada no Corolário 5.4. Nesta seção apresentamos um algoritmo com esta propriedade, batizado de RDT-LGC.

5.5.1 Estruturas de Dados

Sabemos que todos os *checkpoints* que não satisfaçam a condição imposta pelo Teorema 5.3 devem ser mantidos pelo algoritmo de coleta de lixo. Além disso, sabemos que um processo p_f poderá forçar a retenção de no máximo um *checkpoint* estável de um processo p_i: aquele que talvez faça parte de $R_{\{p_f\}}$ se p_f falhar. Sendo assim, podemos manter em p_i, para cada processo $p_f \in P$, uma referência ao *checkpoint* estável mantido por causa de p_f. Isto pode ser feito por meio de um vetor CM (*Checkpoints* Mantidos) de n entradas armazenado por p_i, onde $CM[f]$ indica o *checkpoint* de p_i que não satisfaz a condição do Teorema 5.3 por causa de p_f, ou seja, o *checkpoint* s_i^γ tal que $s_f^{last\_k_i(f)} \rightarrow c_i^{\gamma+1} \ \wedge \ s_f^{last\_k_i(f)} \not\rightarrow s_i^\gamma$. Dessa maneira, quando um *checkpoint* deixar de ser referenciado por todas as entradas de CM ele poderá ser coletado, de acordo com o Teorema 5.3. O Algoritmo 5.3 apresenta as estruturas e procedimentos auxiliares para a manutenção do vetor CM.

Algoritmo 5.3 Estruturas de dados e procedimentos utilizados pelo RDT-LGC.

Estruturas de Dados

```
1: Type
2:    BCC: record of                          {Bloco de Controle do Checkpoint}
3:             IND: inteiro                   {índice local}
4:             CR: inteiro                    {contador de referências}
5:          end
6: Var
7:    CM: array[0...n-1] of ↑BCC              {Ponteiros para BCCs dos checkpoints mantidos}
8:    DV: array[0...n-1] of inteiro           {Vetor de Dependências}
```

Procedure inicializa() {inicializa vetores CM e DV}

```
1: for i ← 0 to n-1 do
2:    CM[i] ← Null
3:    DV[i] ← 0
4: end for
```

Procedure libera(*i*:*inteiro*) {libera $CM[i]$}

```
1: if CM[i] ≠ Null then
2:    CM[i]↑.CR ← CM[i]↑.CR−1                 {decrementa o contador de referências}
3:    if CM[i]↑.CR = 0 then                   {se não possui mais nenhuma referência}
4:       elimina checkpoint CM[i]↑.IND        {tornou-se obsoleto e precisa ser eliminado}
5:       desaloca CM[i]                       {desaloca ponteiro}
6:    end if
7:    CM[i] ← Null
8: end if
```

Procedure liga(*i*:*inteiro*, *j*:*inteiro*) {faz $CM[i]$ apontar para $CM[j]$}

```
1: CM[i] ← CM[j]
2: CM[i]↑.CR ← CM[i]↑.CR+1                    {incrementa o contador de referências}
```

Procedure novo_ckpt(*i*:*inteiro*, *ind*:*inteiro*) {cria um novo ponteiro para $CM[i]$}

```
1: CM[i] ← novo BCC                           {aloca espaço para o ponteiro}
2: CM[i]↑.IND ← ind                           {atribui o índice do checkpoint e}
3: CM[i]↑.CR ← 1                              {inicializa o contador de referências}
```

Um processo p_f pode não forçar a retenção de nenhum *checkpoint* estável de um processo p_i (caso $last\_k_i(f) < 0$). Além disso, várias entradas de CM podem referenciar o mesmo *checkpoint*. Para resolver estes problemas e aumentar a eficiência do algoritmo, utilizamos ponteiros para estruturas que representam um *checkpoint* mantido, chamadas de Blocos de Controle de *Checkpoints*(BCC). Um BCC pode ser usado para armazenar qualquer dado volátil de um *checkpoint*. Para o algoritmo RDT-LGC, no entanto, se faz necessário apenas seu índice e um contador de referências, o qual representa o número de entradas $CM[f]$ que apontam para o BCC em questão.

O procedimento **libera** é utilizado para liberar uma entrada $CM[i]$, isto é, apagar a referência desta entrada a um BCC existente. Este procedimento primeiramente decrementa o valor do contador de referências do BCC apontado e, caso o contador chegue a zero, automaticamente elimina o *checkpoint* do meio de armazenamento estável e desaloca o espaço de memória utilizado pelo BCC. O procedimento **liga** faz com que uma entrada $CM[i]$ aponte para o mesmo BCC apontado por uma entrada $CM[j]$ e automaticamente incrementa o contador de referências deste BCC. Finalmente, o procedimento **novo_ckpt** cria um novo BCC referente ao *checkpoint* com índice de valor *ind* e faz a entrada $CM[i]$ apontar para ele. Este procedimento é utilizado apenas quando um novo *checkpoint* é gravado pelo processo.

5.5.2 Período de Execução Normal

De posse das estruturas de dados e procedimentos apresentados na seção anterior fica fácil desenvolver um algoritmo capaz de manter armazenados apenas os *checkpoints* que não satisfazem a condição apresentada no Corolário 5.4 durante toda a execução de um processo e não apenas após a execução de um determinado procedimento como é feito no Algoritmo 5.2. Note que a condição do Corolário 5.4 é baseada totalmente nos vetores de dependências dos *checkpoints* estáveis e do *checkpoint* volátil de cada processo. Sendo assim, o conjunto de *checkpoints* a serem mantidos só pode ser alterado quando um novo *checkpoint* é gravado ou quando o vetor de dependências volátil do processo se altera, o que só pode ocorrer na recepção de uma mensagem. Desta maneira, podemos executar procedimentos especiais nestes eventos de forma a manter a consistência das estruturas apresentadas na seção anterior e garantir que os *checkpoints* sejam coletados tão logo satisfaçam a condição imposta pelo Corolário 5.4, conforme é apresentado no Algoritmo 5.4.

Ao enviar uma mensagem, o vetor de dependências é propagado na forma de um *time-stamp* de forma a permitir a captura de novas dependências causais. Quando um processo p_i recebe uma mensagem ele atualiza o seu vetor de dependências baseado no vetor recebido. Sendo assim, se uma entrada for alterada, é necessário modificar as estruturas mantidas pelo algoritmo RDT-LGC, uma vez que algum *checkpoint* pode ter se tornado obsoleto. Já sabemos que uma entrada $DV[j]$ é alterada quando uma nova dependência causal do processo p_j é percebida. Neste caso após a alteração temos, pelo próprio algoritmo de propagação do vetor de dependências, que

$$DV(v_i)[j] = DV(v_i)[j] \ \wedge \ DV(v_i)[j] > DV(s_i^{last})[j]. \tag{5.1}$$

Portanto, o processo p_j está impedindo que o *checkpoint* s_i^{last} seja considerado obsoleto pelo Corolário 5.4. Devemos, então, liberar a antiga entrada $CM[j]$ (linha 4) e fazê-la apontar para o BCC do *checkpoint* s_i^{last}. Conforme veremos na seqüência, sabemos que $CM[i]$ sempre aponta para o BCC de s_i^{last}. Então, para atualizar corretamente $CM[j]$, basta fazê-lo apontar para o mesmo BCC apontado por $CM[i]$ (linha 5).

Algoritmo 5.4 RDT-LGC durante um período de execução normal em um processo p_i.

Inicialização

1: inicializa() {Inicializa CM e DV}

Ao enviar mensagem m

1: **for** $j \leftarrow 0$ **to** $n-1$ **do**
2: $m.DV[j] \leftarrow DV[j]$ {envia DV em m como um *timestamp*}
3: **end for**
4: $envia_{rede}(m)$

Ao receber mensagem m

1: **for** $j \leftarrow 0$ **to** $n-1$ **do**
2: **if** $m.DV[j] > DV[j]$ **then** {*nova dependência causal*}
3: $DV[j] \leftarrow m.DV[j]$ {atualiza DV}
4: libera(j) {libera $CM[j]$}
5: liga(j,i) {liga $CM[j]$ com o último *checkpoint* estável}
6: **end if**
7: **end for**
8: $entrega(m)$

Ao gravar *checkpoint*

1: armazena DV junto com o *checkpoint* {para recuperar durante um retrocesso}
2: libera(i) {libera $CM[i]$}
3: newckpt(i,$DV[i]$) {liga $CM[i]$ com o novo *checkpoint*}
4: $DV[i] \leftarrow DV[i]+1$ {atualiza DV}

Por definição, a entrada $DV[i]$ é alterada quando um novo *checkpoint* é gravado. Após a gravação do *checkpoint* e atualização de $DV[i]$, temos que

$$DV(v_i)[i] = DV(v_i)[i] \ \wedge \ DV(v_i)[i] > DV(s_i^{last})[i]. \tag{5.2}$$

Desta forma, o processo p_i está impedindo que o *checkpoint* s_i^{last} seja considerado obsoleto pelo Corolário 5.4. Assim como no caso anterior, devemos liberar a antiga entrada $CM[i]$ (linha 2) e fazê-la apontar para o BCC do *checkpoint* s_i^{last}. Como s_i^{last} acabou de ser criado ele não possui um BCC e podemos criar um utilizando o procedimento **novo_ckpt**, onde o índice do *checkpoint* é a própria entrada $DV[i]$ antes da atualização. Note que desta maneira $CM[i]$ sempre irá apontar para o BCC de s_i^{last} conforme havíamos comentado no parágrafo anterior.

A Figura 5.4 apresenta um exemplo de execução do algoritmo RDT-LGC. Para cada evento detalhado, apresentamos os valores dos vetores DV e CM, um acima do outro, respectivamente. No vetor CM, por simplicidade, mostramos apenas o índice local do *checkpoint* referente a cada BCC e representamos um ponteiro nulo por um *. Neste caso, duas entradas

de mesmo valor no vetor *CM* representam duas entradas apontando para o mesmo BCC. Os quadrados não preenchidos representam os *checkpoints* obsoletos identificados e coletados durante a execução do algoritmo. Note que são os mesmos *checkpoints* apresentados na Figura 5.3. Isto se deve ao fato da mesma condição suficiente estar sendo testada pelos dois algoritmos. A diferença do RDT-LGC é que a coleta é feita de forma *on-line*, durante a execução da computação e, desta forma, um *checkpoint* é coletado tão logo satisfaça a condição do Corolário 5.4 e o limite de n *checkpoints* armazenados por processo nunca é ultrapassado.

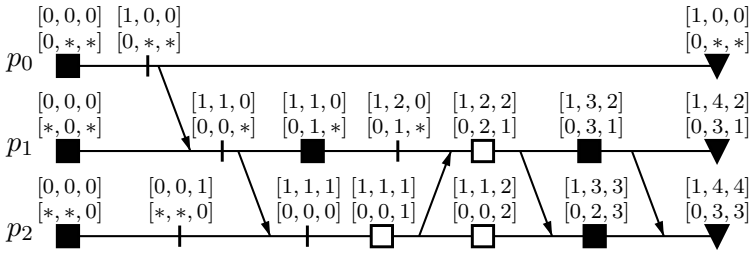

Figura 5.4: Execução do algoritmo RDT-LGC.

5.5.3 Recuperação de uma Falha

Vimos na Seção 2.4.2 que o retrocesso para recuperação de uma falha resulta de uma ação coordenada entre os processos que pode ser classificada como síncrona ou assíncrona. Quando um algoritmo de recuperação síncrono é utilizado, não é errado supor que os processos recebam informações completas sobre o *checkpoint* global consistente para o qual a computação distribuída irá retroceder. Assumindo esta hipótese é possível realizar uma coleta de lixo ótima durante o retrocesso sem prejudicar a consistência das estruturas utilizadas pelo algoritmo RDT-LGC. Vamos supor inicialmente que informações globais sobre a linha de recuperação são repassadas aos processos durante a sessão de recuperação na forma de um vetor de últimos intervalos *UI*. Este vetor é análogo ao vetor *UC* descrito nas Seções 4.3 e 5.3, apenas representando os últimos intervalos entre *checkpoints* ao invés dos últimos *checkpoints* de cada processo. Na verdade, temos que $UI[j] = UC[j] + 1$. A Figura 5.5 apresenta um exemplo de vetor *UI*. A razão de utilizarmos este vetor é a sua identificação mais direta com o vetor de dependências.

Um processo p_i que esteja em um período de execução normal, de posse do vetor *UI*, pode realizar uma coleta de lixo ótima em seus *checkpoints* utilizando as mesmas estruturas do algoritmo RDT-LGC baseando-se na condição apresentada no Corolário 5.1, conforme é apresentado no Algoritmo 5.5. Neste caso, quando $DV[j] < UI[j]$, temos que $s_j^{last} \not\rightarrow v_i$ e, portanto, nenhum *checkpoint* de p_i deve ser retido por causa de p_j e a entrada $CM[j]$

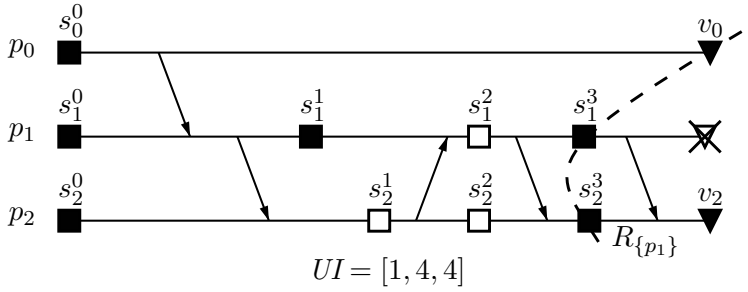

Figura 5.5: Exemplo de vetor *UI* em um retrocesso.

Algoritmo 5.5 Coleta ótima com as estruturas do RDT-LGC.

Procedure coleta_ótima(*UI*: array[$0 \ldots n-1$] **of *inteiro*)**
1: **for** $j \leftarrow 0$ **to** $n-1$ **do**
2: **if** $DV[j] < UI[j]$ **then**
3: libera(j) {libera entradas obsoletas}
4: **end if**
5: **end for**

pode ser liberada. Desta maneira, um processo que utilize o seu *checkpoint* volátil em uma linha de recuperação, como p_0 na Figura 5.5, não precisa retroceder, mas pode utilizar o procedimento **coleta_ótima** para eliminar todos os seus *checkpoints* obsoletos.

Diferentemente, processos que precisem retroceder a um *checkpoint* estável devem executar o Algoritmo 5.6. Neste caso, além do vetor *UI*, o gerente de retrocesso informa ao processo p_i o índice do *checkpoint* estável para o qual ele deve retroceder, aqui chamado índice de retrocesso, ou *IR*. Logo após o procedimento de recuperação ser iniciado, p_i elimina todos os *checkpoints* posteriores à linha de recuperação e reconstrói o seu vetor de dependências a partir do vetor de s_i^{IR} (linhas 4-6). Se p_i corresponde a um processo falho, é necessário criar os BCCs dos *checkpoints* armazenados em meio estável (linha 7). Este passo não é necessário se p_i não falhou pois pois todos os BCCs já estão criados na memória. Neste caso, entretanto, é necessário zerar os seus contadores de referências. Depois disso, para cada processo p_f o algoritmo procura o *checkpoint* de p_i que deve ser mantido em meio estável por causa de p_f utilizando como base para a busca a propriedade do Corolário 5.2 (linha 9). Note que isto só é possível porque os vetores de dependências são armazenados juntamente com os *checkpoints*. Se a busca for bem sucedida, faz-se $CM[f]$ apontar para o BCC do *checkpoint* encontrado (linhas 11-12); caso contrário a entrada $CM[f]$ é simplesmente invalidada pois nenhum *checkpoint* estável de p_i precisa ser mantido por causa de p_f. Ao final deste passo, o vetor *CM* encontra-se corretamente inicializado. Sendo assim, basta eliminar cada *checkpoint* cujo BCC não é referenciado (linhas 17-20) pois tais *checkpoints* são obsoletos segundo o Corolário 5.2. A Figura 5.6 apresenta o mesmo cenário da Figura 5.5 após a execução do Algoritmo 5.6.

Algoritmo 5.6 RDT-LGC durante um retrocesso em um processo p_i.

```
 1: Input
 2:    UI: array[0...n−1] of inteiro                    {vetor de últimos intervalos}
 3:    IR: inteiro                                      {índice da componente de p_i em R_F}
 4: elimina checkpoints s_i^γ | γ > IR                  {elimina checkpoints que estão após R_F}
 5: DV ← DV(s_i^IR)                                     {recupera DV do checkpoint armazenado}
 6: DV[i] ← DV[i] + 1                                   {novo checkpoint volátil}
 7: cria um novo BCC para cada checkpoint s_i^γ armazenado
 8: for f ← 0 to n − 1 do
 9:    encontra γ | (UI[f] = DV(c_i^{γ+1})[f] ∧ UI[f] > DV(s_i^γ)[f])   {condição do Corolário 5.2}
10:    if encontrou then
11:       CM[f] ← BCC de s_i^γ                         {atribui CM[f] corretamente e}
12:       CM[f]↑.CR ← CM[f]↑.CR + 1                    {incrementa o contador de referências}
13:    else
14:       CM[f] ← Null
15:    end if
16: end for
17: for all {BCC | CR = 0} do
18:    elimina checkpoint representado                  {elimina checkpoints obsoletos}
19:    desaloca BCC
20: end for
```

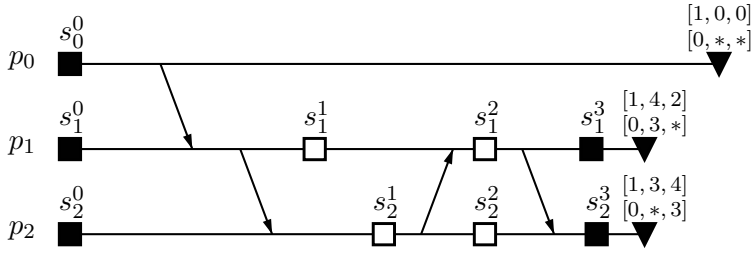

Figura 5.6: Cenário após execução do retrocesso.

Caso o mecanismo de retrocesso seja assíncrono, não há a propagação de informação global sobre a linha de recuperação (vetor *UI*). Neste caso, um processo que não precise retroceder também não precisa alterar as estruturas de dados utilizadas pelo RDT-LGC, uma vez que seu estado volátil é consistente com o estado dos processos que precisam ser retrocedidos. Já os processos que precisam retroceder podem executar o próprio Algoritmo 5.6 substituindo *UI* por *DV* na linha 9. Neste caso, tanto a coleta de lixo quanto a inicialização de *CM* serão feitas com base no Corolário 5.4 ao invés do Corolário 5.2, o que também está correto apesar de não coletar todos os *checkpoints* obsoletos. Note que esta modificação mantém a coerência entre a condição utilizada para a coleta durante um período de execução normal e durante o retrocesso. A Figura 5.7 apresenta o mesmo cenário da Figura 5.5 após a execução do Algoritmo 5.6 com esta modificação.

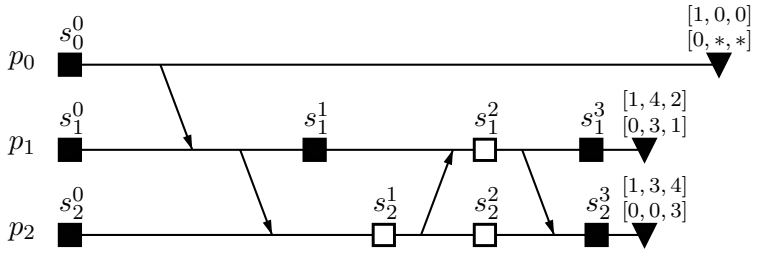

Figura 5.7: Cenário após execução do retrocesso utilizando apenas informação local.

Finalmente, o Algoritmo 5.5 não precisa ser usado apenas durante a recuperação de uma falha. Sempre que houver alguma sincronização entre os processos e um vetor *UI* possa ser construído e propagado entre eles, um processo p_i que o receba poderá executar o procedimento **coleta_ótima** localmente com a garantia de eliminar todos os *checkpoints* obsoletos após a sua execução. Note que desta maneira, assim como no retrocesso dos processos, estamos alternando entre a utilização de uma condição apenas suficiente e outra necessária e suficiente para a realização da coleta de lixo, o que poderia comprometer a corretude do algoritmo. Na próxima seção apresentamos uma prova de que isso não ocorre uma vez que existe uma propriedade invariante do algoritmo que garante a sua corretude.

5.6 Prova de Corretude

Esta seção apresenta uma prova formal da corretude do algoritmo RDT-LGC. A prova é baseada em uma propriedade invariante do algoritmo que ajuda, inclusive, a compreender o seu funcionamento. Para facilitar o entendimento, dividimos esta seção em três partes. A primeira subseção apenas apresenta a propriedade e demonstra a sua importância, enquanto as duas subseções seguintes comprovam que sua veracidade é mantida ao longo da execução da computação.

5.6.1 Propriedade Invariante do Algoritmo

A corretude do protocolo RDT-LGC deriva da seguinte propriedade invariante mantida durante a sua execução para cada processo p_i pertencente à aplicação distribuída:

$$s_f^{last} \rightarrow c_i^{\gamma+1} \wedge s_f^{last} \not\rightarrow s_i^{\gamma} \Rightarrow CM_i[f] = BCC \ de \ s_i^{\gamma} \tag{5.3}$$

O que esta propriedade quer dizer é que todos os *checkpoints* não-obsoletos de um processo são referenciados por pelo menos uma entrada de seu vetor CM. Além disso, por análise do algoritmo RDT-LGC sabemos que nenhum *checkpoint* referenciado por uma entrada do vetor

CM pode ser coletado. Desta forma, temos uma garantia de que nenhum *checkpoint* que não seja obsoleto é eliminado pelo algoritmo e portanto ele está correto, conforme apresentamos no Teorema 5.4.

Teorema 5.4 *Se a propriedade apresentada na Equação 5.3 se mantém durante a sua execução, então todo o* checkpoint *estável eliminado pelo algoritmo RDT-LGC é obsoleto.*

Prova: Sabemos pelo Corolário 5.1 que um *checkpoint* s_i^γ é não-obsoleto se, e somente se, existe um processo p_f tal que $s_f^{last} \rightarrow c_i^{\gamma+1} \wedge \ s_f^{last} \not\rightarrow s_i^\gamma$. Portanto, se a invariante for verdadeira, temos que a entrada $CM[f]$ em p_i aponta para o BCC de s_i^γ. Analisando o algoritmo RDT-LGC vemos que um *checkpoint* só é coletado quando nenhuma entrada do vetor *CM* aponta para o seu BCC, isto é, quando seu contador de referências está zerado. Sendo assim, podemos concluir que nenhum *checkpoint* não-obsoleto pode ser eliminado pelo algoritmo. □

Desta maneira, precisamos apenas provar que a propriedade apresentada na Equação 5.3 realmente se mantém durante a execução do algoritmo. Após a inicialização das variáveis sua veracidade é trivialmente verificada uma vez que não existe nenhum *checkpoint* gravado e, portanto, nenhum *checkpoint* obsoleto. Resta apenas provar que a propriedade se mantém nos períodos de execução normal e nas sessões de recuperação, onde podem ocorrer retrocessos. As duas seções seguintes abordam estes dois casos, respectivamente.

5.6.2 Período de Execução Normal

Para verificar se a propriedade se mantém durante um período de execução normal devemos analisar todos os pontos onde as variáveis presentes na Equação 5.3 podem ser alteradas. O termo $(s_f^{last} \rightarrow c_i^{\gamma+1} \wedge s_f^{last} \not\rightarrow s_i^\gamma)$ só é alterado quando a dependência $s_f^{last} \rightarrow c_i^{\gamma+1}$ é criada ou quando p_f grava um novo *checkpoint* estável que sobrepõe o s_f^{last} anterior. O termo $(CM[f] = BCC\ de\ s_i^\gamma)$ pode ser alterado pelo Algoritmo 5.4 ao executar as linhas 3-5 do evento **receber mensagem** que correspondem à identificação de uma nova precedência causal proveniente de p_f, ou ao executar as linhas 2 e 3 do evento **gravar *checkpoint***. Além disso, conforme foi colocado anteriormente, um processo pode executar o procedimento **coleta_ótima** (Algoritmo 5.5) sempre que o vetor *UI* estiver disponível. Sendo assim, como há interseções entre alguns dos eventos mencionados, precisamos analisar apenas os seguintes:

- identificação de uma nova precedência causal de p_f em p_i.
- gravação de um novo *checkpoint* em p_i.
- gravação de um novo *checkpoint* em p_f.
- execução do procedimento **coleta_ótima** em p_i.

Os lemas seguintes abordam a veracidade da invariante em cada um desses eventos separadamente.

Lema 5.1 *A Equação 5.3 se mantém verdadeira após a identificação de uma nova precedência causal em p_i proveniente de um processo p_f.*

Prova: Quando o RDT-LGC recebe uma nova precedência causal relativa a um *checkpoint* s_f^ι de um processo p_f, ele atualiza a entrada $CM[f]$ fazendo-a referenciar o último *checkpoint* estável de p_i. Se $s_f^\iota = s_f^{last}$, a atualização simplesmente mantém a propriedade invariante. Caso contrário ($s_f^\iota \neq s_f^{last}$), a atualização não compromete a propriedade pois neste caso sabemos que s_f^{last} é posterior a s_f^ι e não precede causalmente nenhum *checkpoint* de p_i, sem poder satisfazer o termo da esquerda na Equação 5.3. □

Lema 5.2 *A Equação 5.3 se mantém verdadeira após a gravação de um novo* checkpoint *no processo p_i.*

Prova: Quando o processo p_i grava um novo *checkpoint*, o RDT-LGC faz com que a entrada $CM[i]$ referencie o novo *checkpoint* gravado. Esta alteração mantém a invariante uma vez que após a gravação de s_i^{last} temos que $s_i^{last} \rightarrow v_i \wedge s_i^{last} \not\rightarrow s_i^{last}$. □

Lema 5.3 *A Equação 5.3 se mantém verdadeira no processo p_i após a gravação de um novo* checkpoint *em um processo p_f.*

Prova: Neste caso, uma possível relação ($s_f^{last} \rightarrow c_i^{\gamma+1} \wedge s_f^{last} \not\rightarrow s_i^\gamma$) pode ter sido quebrada. Porém, o novo *checkpoint* s_f^{last} acabou de ser criado e não precede nenhum *checkpoint* de p_i sem poder satisfazer o termo da esquerda na Equação 5.3 e tornando-a trivialmente verdadeira independente do valor da entrada $CM[f]$. □

Lema 5.4 *A Equação 5.3 se mantém verdadeira no processo p_i após a execução do procedimento* ***coleta_ótima****.*

Prova: O procedimento **coleta_ótima** utiliza informação global da computação e libera uma entrada $CM[f]$ somente se não existir um *checkpoint* s_i^γ que satisfaça o termo da esquerda na Equação 5.3, mantendo-a verdadeira. □

Uma vez que a propriedade é satisfeita após todos os eventos mencionados conclui-se que ela se mantém verdadeira por todo um período de execução normal da computação, conforme apresentamos no teorema seguinte.

Teorema 5.5 *A propriedade representada pela Equação 5.3 se mantém verdadeira em um processo p_i durante um período de execução normal.*

Prova: Pela aplicação conjunta dos Lemas 5.1, 5.2, 5.3 e 5.4. □

5.6.3 Recuperação de uma Falha

Durante a recuperação de uma falha, um processo p_i pode ser obrigado a retroceder ou não. Se ele não precisar retroceder, seu estado volátil é consistente com a linha de recuperação e nenhuma precedência $s_j^{last} \rightarrow c_i^{\gamma+1}$ pode ter sido quebrada, de forma que a invariante permanece verdadeira. Além disso, caso o procedimento **coleta_ótima** seja executado, já sabemos pelo Lema 5.4 que a propriedade se mantém verdadeira.

Diferentemente, para os processos que retrocedem o RDT-LGC executa um algoritmo simples (Algoritmo 5.6) e local ao processo. Desta maneira fica fácil demonstrar que a propriedade também se mantém verdadeira nesta ocasião, conforme apresentamos no seguinte teorema:

Teorema 5.6 *A propriedade representada pela Equação 5.3 se mantém verdadeira em um processo p_i durante a realização de um retrocesso.*

Prova: Como o Algoritmo 5.6 utiliza informação global, sempre que é encontrado um *checkpoint* s_i^{γ} que satisfaça a condição da propriedade invariante para um processo p_f, a entrada $CM[f]$ é atribuída corretamente. Caso tal *checkpoint* não exista para um determinado processo p_f, a entrada $CM[f]$ é anulada. Este procedimento obviamente garante a manutenção da propriedade invariante. □

Se não houver acesso ao vetor UI durante a recuperação, a utilização do vetor DV não prejudica a corretude do algoritmo. Neste caso, se houver um processo p_f e um *checkpoint* s_i^{γ} em p_i tais que $s_f^{last} \rightarrow c_i^{\gamma+1} \wedge s_f^{last} \not\rightarrow s_i^{\gamma}$, então temos que a entrada $DV[f]$ do vetor mantido por p_i é igual a $UI[f]$ e a atribuição feita pelo algoritmo se manterá correta com relação à propriedade invariante.

5.7 Análise e Conclusão

Esta seção apresenta uma análise do desempenho do algoritmo RDT-LGC com relação a diferentes aspectos como número de *checkpoints* mantidos, complexidades de espaço e tempo e possíveis otimizações em sua implementação, incluindo a implementação conjunta com protocolos para *checkpointing* que satisfaçam a propriedade RDT. Após esta análise, concluímos que o RDT-LGC oferece o limite ótimo para o número de *checkpoints* armazenados por processo sem aumentar as complexidades de tempo ou espaço dos protocolos RDT mais eficientes e, principalmente, utilizando-se dos mesmos *timestamps* nas mensagens da aplicação, sem mensagens de controle adicionais.

5.7.1 Limite Máximo de *Checkpoints* Mantidos

Pode ser facilmente verificado que o RDT-LGC mantém no máximo n *checkpoints* não coletados em cada processo, uma vez que um *checkpoint* estável só é mantido se seu BCC for referenciado por uma das n entradas do vetor CM. Este limite por processo gera um limite global de no máximo n^2 *checkpoints* não coletados pelo algoritmo. Pode-se pensar que é possível reduzir este limite de maneira semelhante ao que foi feito na Seção 4.5.2. No entanto, como o RDT-LGC leva em consideração apenas o passado conhecido de cada processo, tal redução não é possível.

Considere como exemplo o cenário apresentado na Figura 5.8, onde os *checkpoints* coletados pelo algoritmo são representados por quadrados não preenchidos. Neste cenário, cada processo p_i recebe no intervalo I_i^j $(i \neq j-1)$ pela primeira vez a única informação conhecida proveniente do processo p_{j-1}. Isto faz com que o seu *checkpoint* s_i^{j-1} seja mantido por não satisfazer a condição imposta pelo Teorema 5.3, utilizada para coleta de lixo pelo RDT-LGC durante um período de execução normal. Por fim, o *checkpoint* s_i^n de cada processo p_i é mantido por ser o seu último *checkpoint* estável e fazer parte da linha de recuperação no caso de p_i falhar. Resumidamente, temos que para todo processo p_i, $CM[j]$ referencia o BCC do *checkpoint* s_i^j quando $i \neq j$ e o BCC de s_i^n quando $i = j$. Como todas as entradas são diferentes, cada processo mantém n *checkpoints* estáveis não coletados, totalizando n^2.

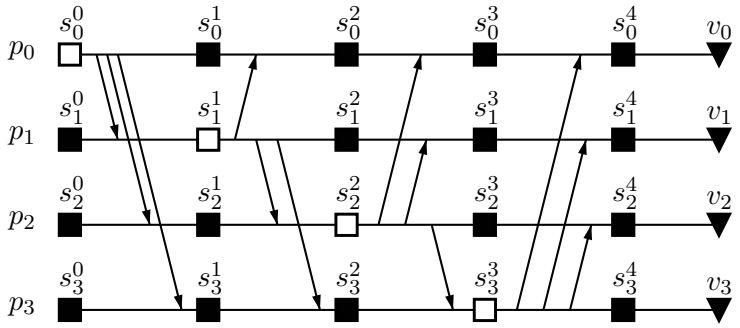

Figura 5.8: Cenário onde n^2 *checkpoints* estáveis são retidos pelo RDT-LGC.

Apesar da coleta de lixo ótima apresentar um limite global para o número de *checkpoints* mantidos menor que n^2 (como vimos na Seção 4.5.2, seu limite é $n(n+1)/2$), o limite máximo por processo é igual n. Desta maneira, como não é possível saber que processo poderá ficar com n *checkpoints* armazenados após a coleta, todos eles devem ter espaço suficiente em meio estável para armazená-los, o que nos leva na prática a um espaço mínimo global necessário de n^2 *checkpoints* em meio de armazenamento estável. O RDT-LGC respeita o limite máximo de n *checkpoints* mantidos por processo com vantagens práticas inexistentes na coleta de lixo ótima como a coleta em tempo de execução e a ausência de sincronização entre os processos ou troca de mensagens de controle.

Finalmente, um pequeno detalhe prático do RDT-LGC pode fazer com que um processo tenha que armazenar $n + 1$ *checkpoints* por um breve período de tempo. Isto ocorreria exatamente entre as linhas 1 e 2 do evento **gravar checkpoint** no Algoritmo 5.4. Neste caso, as n entradas do vetor CM poderiam estar referenciando *checkpoints* diferentes enquanto um novo *checkpoint* deve ser gravado. Sabemos que o *checkpoint* referenciado pela entrada de CM referente ao próprio processo será eliminado mas se a gravação do novo *checkpoint* fosse feita em cima do anterior, uma falha que ocorresse durante a gravação invalidaria ambos. Podemos resolver isto com uma pequena área extra de memória estável temporária a ser utilizada somente neste breve período ou então relaxando o modelo e assumindo que falhas não ocorrem durante a gravação de um *checkpoint*.

5.7.2 Análise de Complexidade

O RDT-LGC utiliza apenas dois vetores de n entradas, DV e CM, durante a sua execução. Além disso, por análise do algoritmo verificamos que todo BCC alocado é referenciado por pelo menos uma entrada do vetor CM. Portanto, concluímos que ele utiliza apenas $O(n)$ espaço de memória volátil em um período de execução normal. Já o espaço utilizado durante um retrocesso dependerá do número de *checkpoints* estáveis armazenados, uma vez que será alocado um BCC para cada um deles durante a inicialização do vetor CM. No entanto, como o RDT-LGC garante que o número máximo de *checkpoints* mantidos em um processo é n, obtemos também um limite de $O(n)$ memória utilizada durante este procedimento.

Com exceção de **inicializa**, todos os procedimentos apresentados no Algoritmo 5.3 executam em tempo $O(1)$. Isto possibilita uma fácil análise da complexidade de tempo do RDT-LGC durante uma execução normal. Verificamos que todos os eventos apresentados no Algoritmo 5.4 executam em tempo $O(n)$. Especificamente, o evento **gravar checkpoint** tem esta complexidade apenas por causa da gravação do vetor DV em meio estável uma vez que supomos uma taxa de gravação proporcional ao tamanho da estrutura. Estas complexidades ainda podem ser reduzidas por meio de uma implementação conjunta com o protocolo para *checkpointing* conforme veremos na seção seguinte.

Analisar a complexidade de tempo do RDT-LGC durante um retrocesso (Algoritmo 5.6) também não é complicado. É fácil verificar que o passo dominante do algoritmo corresponde à sua linha 9, onde é feita uma busca pelo *checkpoint* estável que deverá ser referenciado pela entrada $CM[f]$. Vamos supor que os *checkpoints* estão apresentados de forma ordenada crescente pelos seus índices. A maneira mais simples de realizar esta busca é comparando um a um os vetores de dependências de um *checkpoint* e seu sucessor o que tomaria tempo $O(n)$. No entanto, uma dependência causal sempre é transmitida de um *checkpoint* para o seu sucessor por definição. Desta forma, podemos encontrar o *checkpoint* desejado utilizando uma busca binária simples procurando o intervalo entre *checkpoints* onde a dependência aparece pela primeira vez. Esta otimização reduziria o tempo de execução deste passo

para $O(\log n)$. Como esta busca encontra-se dentro de um laço de repetição, temos uma complexidade total de tempo da ordem de $O(n \log n)$.

Uma outra otimização possível no Algoritmo 5.6 se refere aos processos que não falham mas que precisam retroceder por causa de algum outro processo. Neste caso, o vetor *CM* já se encontra parcialmente preenchido e este fato pode ser aproveitado para melhorar o desempenho do algoritmo. Em tais processos, é necessário apenas ajustar a entrada de *CM* referente ao próprio processo de forma a apontar ao BCC do *checkpoint* utilizado no retrocesso, liberar as demais entradas referentes aos *checkpoints* já retrocedidos e ao *checkpoint* utilizado no retrocesso, e finalmente, se o vetor *UI* estiver disponível, executar o procedimento **coleta_ótima**. Todas estas operações executam em tempo $O(n)$, o que é um pouco melhor que a complexidade $O(n \log n)$ do algoritmo para o caso geral. Note que para ajustar a entrada referente ao próprio processo basta procurar a entrada $CM[f]$ que aponta para o BCC do *checkpoint* utilizado, uma vez que este *checkpoint* é mantido por causa de um dos processos que falhou.

5.7.3 Implementação Conjunta com Protocolos RDT

A implementação do algoritmo RDT-LGC muito se assemelha com a implementação de protocolos quase-síncronos, mais especificamente com aqueles da classe ZPF que satisfaz a propriedade RDT. Os protocolos mais eficientes desta classe também se utilizam de vetores de dependências, o que cria uma interseção considerável entre eles e o RDT-LGC. Sendo assim, é possível implementar as duas coisas em conjunto, aproveitando as estruturas de dados, comandos e propriedades comuns. Como exemplo, considere o Algoritmo 5.7 que apresenta uma possível implementação conjunta do algoritmo RDT-LGC com o protocolo FDAS apresentado na Seção 3.2.2.

A inicialização apenas une as inicializações dos dois algoritmos, uma vez que DV é automaticamente inicializado dentro do procedimento **inicializa**. O envio de mensagens é praticamente o mesmo, sendo que a única diferença é que o FDAS requisita que a variável *enviou* receba o valor *verdadeiro* para indicar a ocorrência de um envio de mensagem. Ao receber uma mensagem, pode-se utilizar a mesma otimização do FDAS para constatar a existência de uma nova dependência causal (linha 1). O único cuidado que deve ser tomado é a gravação do *checkpoint* forçado antes da coleta de lixo, para que a entrada $CM[i]$ seja corretamente atualizada antes de modificar as entradas $CM[j]$ referentes às novas dependências causais identificadas. O código executado durante a gravação de um *checkpoint* é apenas uma união entre os códigos do FDAS e do RDT-LGC. Note que como a entrada $DV[i]$ indica o índice do *checkpoint* gravado, sua atualização deve ser feita por último (linha 3).

Algoritmo 5.7 FDAS com RDT-LGC em um processo p_i.

Estruturas de Dados {além daquelas definidas no Algoritmo 5.3}

```
1: Var
2:    enviou: booleano
```

Inicialização

```
1: enviou ← falso
2: inicializa()
```

Ao enviar mensagem m

```
1: enviou ← verdadeiro
2: for j ← 0 to n − 1 do
3:    m.DV[j] ← DV[j]
4: end for
5: envia_rede(m)
```

Ao receber mensagem m **de** p_k

```
1: if m.DV[k] > DV[k] then
2:    if enviou then
3:       grava checkpoint
4:    end if
5:    for j ← 0 to n − 1 do
6:       if m.DV[j] > DV[j] then
7:          DV[j] ← m.DV[j]
8:          libera(j)
9:          liga(j,i)
10:      end if
11:   end for
12: end if
13: entrega(m)
```

Ao gravar *checkpoint*

```
1: enviou ← falso
2: armazena DV junto com o checkpoint
3: libera(i)
4: newckpt(i,DV[i])
5: DV[i] ← DV[i] + 1
```

Conforme as análises feitas na seção anterior sobre a complexidade dos procedimentos utilizados pelo RDT-LGC e a complexidade do próprio algoritmo FDAS, podemos verificar que a implementação conjunta não aumentou a complexidade de tempo ou espaço já existente no FDAS. Isto significa que a utilização do algoritmo RDT-LGC para coleta de lixo implica em uma perda de desempenho desprezível, com a vantagem de limitar o espaço necessário

em meio de armazenamento estável. Este mesmo tipo de implementação conjunta pode ser realizada com outros protocolos RDT eficientes como o RDT-Partner [19] e o BHMR [4].

5.7.4 Conclusão

O RDT-LGC é, dentro de nosso conhecimento, o único algoritmo para coleta de lixo que realiza esta tarefa sem sincronizar os processos ou trocar mensagens de controle. Este algoritmo funciona em qualquer padrão de *checkpoints* que satisfaça a propriedade RDT e não está limitado a um protocolo para *checkpointing* específico. Além disso, ele garante que cada processo manterá no máximo n *checkpoints* armazenados durante sua execução normal, o que limita a quantidade de armazenamento estável necessária em um processo ao ótimo teórico. Isto significa que não é possível criar um protocolo com limite menor por processo. Além disso, os *timestamps* e estruturas utilizados pelo RDT-LGC são muito semelhantes àqueles utilizados por protocolos para *checkpointing* que satisfazem a propriedade RDT, o que permite uma implementação eficiente e sem degradação de desempenho com relação à utilização de um protocolo RDT apenas (sem coleta de lixo).

Todos os algoritmos para coleta de lixo anteriores ao RDT-LGC são baseados em alguma forma de sincronização explícita entre os processos para identificação dos *checkpoints* obsoletos. Mesmo a coleta ingênua, que não oferece um limite ao número de *checkpoints* mantidos, necessita que mensagens de controle sejam trocadas entre os processos. Desta maneira a utilização de protocolos quase-síncronos para *checkpointing* na prática esbarra na contradição existente entre a ausência de sincronização para a seleção dos *checkpoints* e a sua necessidade para efetuar a coleta de lixo. Isso é provavelmente uma das razões pelas quais protocolos síncronos são mais utilizados na prática [2, 30, 31]. Conforme foi apresentado antes o RDT-LGC não apresenta este problema, além de ser um algoritmo de implementação simples. Por este motivo, acreditamos que ele rompe uma grande barreira para a utilização prática deste tipo de protocolo em sistemas de recuperação de falhas por retrocesso de estado, o que aumenta ainda mais sua importância.

Capítulo 6

Conclusão

Esta dissertação abordou em detalhes o problema de coleta de lixo para protocolos de *checkpointing*. Este problema, apesar de ter uma definição simples, influencia diretamente na eficiência e desempenho de um sistema de tolerância a falhas baseado em recuperação por retrocesso de estado. Neste contexto, nos deparamos com a inexistência de um algoritmo de coleta de lixo capaz de acompanhar as vantagens existentes na utilização de protocolos para *checkpointing* quase-síncronos. Em geral, os algoritmos existentes exigem um alto grau de sincronismo e contradizem a autonomia oferecida por tais protocolos para a seleção de *checkpoints*. Este fato foi a principal motivação deste trabalho que culminou na elaboração de um algoritmo de coleta de lixo eficiente que funciona para uma classe inteira de protocolos quase-síncronos.

Inicialmente apresentamos todo um embasamento teórico necessário para a compreensão do problema e o seu entorno. O contexto dos mecanismos para *checkpointing* e recuperação são explicados em detalhes no Capítulo 2 evidenciando os resultados mais importantes encontrados na literatura sobre estes temas. Nossa proposta visa a especialização da coleta de lixo para os diferentes protocolos de *checkpointing* existentes na busca de soluções mais eficientes. Desta maneira, para ambientar o leitor, apresentamos no Capítulo 3 uma classificação desses protocolos com a identificação das principais características e propriedades dos diferentes grupos existentes.

As duas maiores contribuições desta dissertação são apresentadas nos Capítulos 4 e 5 e consistem de uma caracterização do problema de coleta de lixo baseada na relação de Z-precedência e vários algoritmos eficientes e inéditos para coleta de lixo em padrões que satisfaçam a propriedade RDT, respectivamente. O Capítulo 4 começa com uma revisão dos algoritmos mais clássicos conhecidos para coleta de lixo. Depois desta apresentação, revisitamos o problema da caracterização dos *checkpoints* obsoletos em um CCP, com um enfoque voltado às relações de Z-precedência existentes entre os *checkpoints*. Vale lembrar que os protocolos para *checkpointing* quase-síncronos diferem nos tipos de Z-precedência

permitidos dentro de um padrão, de forma que nossa caracterização pode ser mais facilmente especializada aos padrões gerados em tais protocolos. As caracterizações anteriores [37, 38] eram baseadas em estruturas gerais de representação de um CCP e não permitem este tipo de especialização facilmente. Este capítulo termina com a apresentação do algoritmo ótimo para coleta de lixo e sua análise. Apesar de tal algoritmo já ser conhecido, procuramos dar um enfoque diferente à sua apresentação, com uma abordagem prática e uma análise mais completa tendo em vista os algoritmos eficientes que apresentamos no Capítulo 5.

Ao especializarmos a nossa caracterização dos *checkpoints* obsoletos para os padrões gerados pelos protocolos RDT, obtivemos uma série de algoritmos mais eficientes para coleta de lixo em tais padrões, apresentados no Capítulo 5. Primeiramente mostramos como os algoritmos de coleta ingênua e ótima podem ser simplificados. Depois disso, apresentamos uma condição suficiente que permite uma coleta de lixo local por parte dos processos e garante o limite máximo ótimo para o número de *checkpoints* mantidos em um processo. Finalmente, construímos um algoritmo baseado nesta condição que é capaz de coletar os *checkpoints* obsoletos durante a execução da computação, eliminando-os tão logo possam ser identificados. Este algoritmo, que recebeu o nome de RDT-LGC, permite que um processo limite ao máximo a quantidade de memória estável necessária para o armazenamento de *checkpoints*. Além disso, mostramos que o RDT-LGC não apresenta uma queda de desempenho significativa pela sua utilização, uma vez que se utiliza das mesmas estruturas e *timestamps* dos protocolos RDT.

Quando a propriedade RDT foi apresentada pela primeira vez [36], suas vantagens incluíam apenas a facilidade de construção de *checkpoints* globais consistentes contendo um conjunto de *checkpoints*, o que é uma vantagem particularmente importante para o mecanismo de retrocesso e cálculo da linha de recuperação. Recentemente concluiu-se que tais padrões de *checkpoints* apresentam um limite de retrocesso $n - 1$ vezes menor que padrões onde apenas é garantida a ausência de ciclos-Z [1]. A existência de um protocolo para coleta de lixo descentralizada, sem sincronização e com limite para o número de *checkpoints* mantidos em meio estável é uma importante vantagem para a utilização de protocolos para *checkpointing* RDT na prática e, conforme dissemos anteriormente, somente o RDT-LGC apresenta tais características.

Acreditamos que existam várias possibilidades de extensão deste trabalho. A primeira delas seria uma análise mais prática das características do protocolo RDT-LGC como a taxa média de *checkpoints* mantidos ou a queda de desempenho real ocasionada em um processo. Do ponto de vista teórico, acreditamos que é possível obter algoritmos eficientes de coleta de lixo para as outras classes de protocolos quase-síncronos (ZCF e PZCF), por meio de um raciocínio semelhante ao utilizado para a construção do RDT-LGC. No entanto, como estas classes são menos restritas, julgamos que talvez seja necessário um estudo mais profundo na busca de novas características comuns à classe ou específicas de um ou outro algoritmo.

Referências

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

[2] L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain, and A. D. Mel. An Analysis of Communication-Induced Checkpointing. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing (FTCS-29)*, pages 242–249, June 1999.

[3] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal and Optimal. *IEEE Trans. on Software Engineering*, 24(2):149–159, February 1998.

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

[5] R. Baldoni, J. M. Hélary, and M. Raynal. Rollback-Dependency Trackability: A Minimal Characterization and its Protocol. Technical Report 1173, IRISA, March 1998.

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

[7] R. Baldoni, F. Quaglia, and P. Fornara. An Index-Based Checkpoint Algorithm for Autonomous Distributed Systems. *IEEE Trans. on Parallel and Distributed Systems*, 10(2):181–192, February 1999.

[8] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Databases Systems*. Addison-Wesley, 1987.

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

[10] D. Briatico, A. Ciuffoletti, and L. Simoncini. A Distributed Domino-Effect Free Recovery Algorithm. In *Proceedings of the 4th IEEE Symp. on Reliability in Distributed Software and Database Systems*, pages 207–215, October 1984.

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

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

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

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

[15] I. C. Garcia and L. E. Buzato. Monitorização e Recuperação por Retrocesso Utilizando Visões Progressivas de Aplicações Distribuídas. In *VIII Simpósio de Computação Tolerante a Falhas*, July 1999.

[16] I. C. Garcia and L. E. Buzato. Progressive Construction of Consistent Global Checkpoints. In *Proceedings of the 19th IEEE Int. Conf. on Distributed Computing Systems*, June 1999.

[17] I. C. Garcia and L. E. Buzato. Using Common Knowledge to Improve Fixed-Dependency-After-Send. In *Proceedings of the 2nd Workshop de Testes e Tolerância a Falhas*, July 2000.

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

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

[20] J. M. Hélary, A. Mostéfaoui, and M. Raynal. Communication-Induced Determination of Consistent Snapshots. *IEEE Trans. on Parallel and Distributed Systems*, 10(9):865–877, 1999.

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

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

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

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

[25] D. Manivannan, Robert H. B. Netzer, and M. Singhal. Finding Consistent Global Checkpoints in a Distributed Computation. *IEEE Trans. on Parallel and Distributed Systems*, 8(6):623–627, June 1997.

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

[27] 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.

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

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

[30] S. H. Russ, J. Robinson, B. K. Flachs, and B. Heckel. The Hector Distributed Run-Time Environment. *IEEE Trans. on Parallel and Distributed Systems*, 9(11):1102–1114, 1998.

[31] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, 1996.

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

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

[34] G. M. D. Vieira. Estudo Comparativo de Algoritmos para *Checkpointing*. Master's thesis, Instituto de Computação—Universidade Estadual de Campinas, December 2001.

[35] Y. M. Wang. Maximum and Minimum Consistent Global Checkpoints and Their Applications. In *Proceedings of the 14th Symposium on Reliable Distributed Systems*, pages 86–95, September 1995.

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

[37] Y. M. Wang, P. Y. Chung, and W. K. Fuchs. Tight Upper Bound on Useful Distributed System Checkpoints. Technical Report CRHC-95-16, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1995.

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