UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Rollback solidário : um novo protocolo otimista para simulação distribuída

Edmilson Marmo Moreira

São Carlos – SP

Rollback Solidário : Um novo protocolo otimista para Simulação Distribuída

Edmilson Marmo Moreira

Orientador: ***Profa. Dra. Regina Helena Carlucci Santana***

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências - Ciências de Computação e Matemática Computacional.

"VERSÃO REVISADA APÓS A DEFESA"

Data da Defesa:	11/08/2005
Visto do Orientador: Regina HC Santana	

USP – São Carlos
Outubro/2005

Dedico este trabalho
aos meus pais Antonio e Terezinha,
à minha esposa Karina e aos meus filhos Lucas e Mateus.

Agradecimentos

À minha orientadora profa. Dra. Regina Helena Carlucci Santana, pela orientação segura e pelas agradáveis discussões, sempre produtivas. Também pela amizade, incentivo nas horas difíceis e confiança que sempre depositou em mim.

Ao prof. Dr. Marcos José Santana, pela amizade e valiosas sugestões durante todas as fases deste trabalho.

Aos meus pais Antonio Marmo e Terezinha, pelo amor, dedicação e incentivo.

À minha esposa Karina e aos meus filhos Lucas e Mateus, por compreenderem a minha ausência.

Ao meu irmão Leonardo, pelo companheirismo e grande ajuda com a língua portuguesa.

Aos meus sogros Constantino e Regina, pela dedicação à minha família.

À Maria Stella, pelo amparo e apoio constante.

Ao casal Walter e Lourdes Moreira, pelo incentivo e laços de amizade.

Às amigas Célia e Sarita pelo constante interesse neste trabalho e sincero desejo de ajudar.

Aos amigos Tomás e Alexandre Donizete, pelo estímulo e troca de experiências acadêmicas.

Ao amigo Mauro César, por termos começado juntos esta jornada.

Ao meu irmão Eduardo, pela torcida e amizade.

Aos professores do curso de Ciência da Computação da UNIFENAS, em especial Alexandre e Marly, por acreditarem no meu potencial.

Aos professores do Instituto de Engenharia de Sistemas e Tecnologias da Informação da UNIFEI, pela acolhida fraterna e apoio neste novo ambiente de trabalho.

Aos meus alunos, pelos questionamentos e interação construtiva.

Aos meus colegas do LASDPC, pelas dúvidas levantadas nos seminários e pela agradável

convivência.

Aos funcionários do ICMC, pela disponibilidade em ajudar além da obrigação.

À Universidade de São Paulo, pela oportunidade.

A todos aqueles que direta ou indiretamente contribuíram para a conclusão deste trabalho.

E, sobretudo, a DEUS, por tudo...

Sumário

1 Introdução 1

1.1 Motivação 2

1.2 Objetivos 4

1.3 Estrutura da Tese 4

2 Simulação Distribuída e o Protocolo *Time Warp* 7

2.1 Simulação Distribuída de Eventos Discretos 7

2.1.1 O Protocolo Conservativo 9

2.1.2 O Protocolo Otimista 10

2.2 *Time Warp* 13

2.2.1 Anti-Mensagens 15

2.2.2 Mecanismo de Controle Global 16

2.3 Mecanismos para o cálculo do GVT 18

2.3.1 O Problema da Mensagem Transiente 18

2.3.2 O Problema do Relatório Simultâneo 20

2.3.3 Algoritmo de Samadi para o cálculo do GVT 21

2.3.4 Algoritmo de Mattern para o cálculo do GVT 22

2.3.5 Formas de Cancelamento das Anti-mensagens 26

2.4 Mecanismos de Salvamento de Estado 26

2.4.1 *Incremental State Saving* 28

2.4.2 *Sparse State Saving* 29

2.4.3 *Event Sensitive State Saving* 31

2.4.4 *Hybrid State Saving* 32

2.4.5 *Reverse Computation* 33

2.5 Considerações Finais 33

3 Sincronização em Sistemas Distribuídos **35**

3.1 Estados Locais e Estados Globais 36

3.2 Cortes Globais Consistentes 37

3.3 Relógios Lógicos 38

3.4 Relógios Vetoriais 41

3.5 *Checkpoints* Globais Consistentes 43

3.6 Condições para Consistência dos *Checkpoints* 45

3.7 Algoritmos para *Checkpoints* Síncronos 46

3.7.1 Algoritmo *Sync-and-Stop* 47

3.7.2 Algoritmo Chandy-Lamport 48

3.8 Algoritmos para *Checkpoints* Semi-Síncronos 49

3.8.1 *Strictly Z-path Free Checkpointing* 50

3.8.2 *Z-Path Free Checkpointing* 52

3.8.3 *Z-Cycle Free Checkpointing* 53

3.8.4 *Partially Z-cycle Free Checkpointing* 54

3.9 Visões Progressivas 55

3.10 Linhas de Recuperação 55

3.11 Considerações Finais 56

4 *Rollback* Solidário **57**

4.1 Arquitetura de um Ambiente para Simulação Distribuída 58

4.2 O Comportamento Geral do Protocolo *Rollback* Solidário 60

4.3 Conjunto de Linhas de Recuperação 62

4.4 Utilizando *Checkpoints* Síncronos 63

4.4.1 Algoritmo *Sync-and-Stop* Estendido 63

4.4.2 Algoritmo *Chandy-Lamport* Estendido 64

4.4.3 Tratamento dos *Rollbacks* na Abordagem Síncrona 66

4.5 *Rollback* Solidário com mecanismos de *Checkpoints* Semi-Síncronos 66

4.5.1 Tratamento dos *Rollbacks* na Abordagem Semi-Síncrona com Observador 71

4.6 Problemas com o Protocolo *Rollback* Solidário 73

4.6.1 O Problema da "Anomalia do Retorno Desnecessário" 73

4.6.2 O Problema dos *Rollbacks* Simultâneos 74

4.6.3 O Problema da Mensagem Transiente 76

4.7 Estrutura das Mensagens no Protocolo *Rollback* Solidário 77

4.8 *Rollback* Solidário sem o Processo Observador 78

4.9 *Reutilização da Memória* 85

4.10 Considerações Finais 86

5 Especificação para Implementação do Protocolo *Rollback* Solidário 89

5.1 O Diagrama de Classes 90

5.2 Diagramas de Seqüência 94

5.2.1 *Rollback* Solidário sem Observador 94

5.2.2 *Rollback* Solidário com Observador 99

5.3 Diagramas de Estados 102

5.4 Considerações Finais 104

6 Análise Comparativa 105

6.1 Comparação de Desempenho 105

6.1.1 Desperdício de Processamento no *Time Warp* 107

6.1.2 Desperdício de Processamento no *Rollback* Solidário sem observador 110

6.1.3 Desperdício de Processamento no *Rollback* Solidário com observador 112

6.1.4 Impacto dos *Checkpoints* 113

6.1.5 Análise dos Resultados 117

6.2 FDAS para Simulação Distribuída 121

6.3 Complexidade do Espaço de Armazenamento 123

6.4 Considerações Finais 126

7 Conclusões **127**

7.1 Contribuições deste trabalho 129

7.2 Sugestões para Trabalhos Futuros 131

Referências **134**

A Listagem dos Algoritmos Descritos na Tese **145**

A.1 Algoritmo *Sync-and-Stop* 145

A.2 Algoritmo *Chandy-Lamport* 147

A.3 Algoritmos do Padrão ZPF 149

A.4 Algoritmo *Sync-and-Stop* estendido 151

A.5 Algoritmo *Chandy-Lamport* estendido 153

A.6 Tratamento dos *Rollbacks* 155

A.7 Listagem do Código Observador 156

A.8 Método para Obtenção das Linhas de Recuperação 158

A.9 *Rollback* Solidário Semi-Síncrono 159

A.10 *Rollback* Solidário Sem Observador 160

Lista de Figuras

2.1 Eventos de um processo lógico durante a chegada de uma mensagem *straggler* (FUJIMOTO, 2000) . . . 14

2.2 Diagrama de espaço×tempo representando uma computação distribuída . . 15

2.3 Diagrama de espaço×tempo no *Time Warp* (FUJIMOTO, 2000) . . . 17

2.4 O problema da mensagem transiente (FUJIMOTO, 2000) . . . 19

2.5 O problema do relatório simultâneo . . . 21

2.6 Exemplo de corte dividindo a computação em passado e futuro . . . 23

2.7 Corte da figura 2.6 ilustrando o algoritmo síncrono para o cálculo do GVT 23

2.8 Cálculo do GVT usando dois cortes . . . 24

2.9 Exemplo de funcionamento do mecanismo *Copy State Saving* (FUJIMOTO, 2000) . . . 27

2.10 Exemplo de funcionamento do Mecanismo *Incremental State Saving* . . . 29

2.11 Ilustração do mecanismo *Sparse State Saving* . . . 30

2.12 Ilustração do Mecanismo *Hybrid State Saving* . . . 32

3.1 Diagrama de espaço×tempo com dois cortes em uma computação distribuída (BABAOGLU; MARZULLO, 1993) . . . 37

3.2 Evolução do tempo escalar . . . 40

3.3 Evolução dos mesmos processos da figura 3.2 utilizando um relógio vetorial 42

3.4 *Checkpoints* inúteis (efeito dominó) . . . 44

3.5 Exemplo de um sistema com caminho-Z . . . 46

3.6 Exemplo de um sistema com um ciclo-Z . . . 47

3.7 Funcionamento do algoritmo SNS (PLANK, 1993) . . . 48

3.8 Grafo representando os canais de comunicação entre os processos p_c, p_1 e p_2 48

3.9 Funcionamento do algoritmo CL 50

3.10 Relação entre os vários modelos de algoritmos de *checkpointing* semi-síncronos (MANIVANNAN; SINGHAL, 1999) 50

3.11 Padrão de *checkpoints* dos algoritmos do protocolo SZPF 51

3.12 Padrão de *checkpoints* do protocolo ZPF 52

3.13 Exemplo do padrão de *Checkpoints* do algoritmo BCS 54

3.14 Exemplo de Visões Progressivas. 55

4.1 Estrutura em camadas de um ambiente de Simulação Distribuída 59

4.2 DFD representando um ambiente para simulação distribuída 60

4.3 Análise de uma mensagem *straggler* 61

4.4 Exemplo de Rollback no modelo Solidário 62

4.5 Exemplo de linhas de recuperação 63

4.6 Comportamento do algoritmo SNS estendido 65

4.7 Comportamento do algoritmo Chandy-Lamport estendido 67

4.8 *Rollback* Solidário Síncrono ou Coordenado 68

4.9 Mensagens para o processo observador 69

4.10 Listas encadeadas que armazenam os vetores de dependências dos *checkpoints* 71

4.11 Comportamento do processo observador durante um *rollback* 72

4.12 Estrutura de uma mensagem no protocolo *Rollback* Solidário 78

4.13 Comportamento do protocolo *Rollback* Solidário sem observador durante um *rollback* 80

4.14 Ilustração do problema existente na solução anterior 80

4.15 *Rollback* Solidário sem observador 83

4.16 Hierarquia de Implementação para o protocolo proposto 87

5.1 Camada onde se encontra o protocolo *Rollback* Solidário 89

5.2 Diagrama de classes do protocolo *Rollback* Solidário 93

5.3 Diagrama de seqüência do protocolo *Rollback* Solidário 96

5.4 Diagrama de seqüência durante um *checkpoint* básico 97

5.5 Diagrama de seqüência durante um *rollback* no protocolo *Rollback* Solidário 98

5.6 Diagrama de seqüência durante um *checkpoint* básico na abordagem com observador . . . 100

5.7 Diagrama de seqüência durante um *rollback* da abordagem com observador 101

5.8 Diagrama de estados da classe Processo . . . 102

5.9 Diagrama de Estados da classe Receptor . . . 103

5.10 Diagrama de Estados da classe Emissor . . . 103

5.11 Diagrama de Estados da classe Observador . . . 104

6.1 DFD representando a ferramenta para análise de algoritmos para *checkpointing* . . . 114

6.2 Interface da ferramenta apresentando a tela de configuração do diagrama de Espaço × Tempo . . . 115

6.3 Interface da ferramenta apresentando a matriz que representa o diagrama de Espaço × Tempo . . . 116

6.4 Interface da ferramenta para análise de algoritmos para *checkpointing* . . . 116

6.5 Interface da ferramenta para análise de algoritmos para *checkpointing* . . . 117

6.6 Gráficos dos dados absolutos e relativos das 4 classes . . . 120

6.7 Gráfico considerando a diferença relativa de *checkpoints* com conforme varia o número de processos . . . 121

Lista de Tabelas

6.1 Classes dos testes para o Simulador de Algoritmos para *Checkpointing* . . . 117
6.2 Resultados da classe 1 118
6.3 Resultados da classe 2 118
6.4 Resultados da classe 3 118
6.5 Resultados da classe 4 118
6.6 Resultados relativos da classe 1 118
6.7 Resultados relativos da classe 2 118
6.8 Resultados relativos da classe 3 119
6.9 Resultados relativos da classe 4 119

Lista de Abreviaturas e Siglas

BCS	Briattico, Ciuffoletti e Simoncini
CAS	Checkpoint After Send
CASBR	Checkpoint After Send Before Receive
CBR	Checkpoint Before Receive
CL	Chandy-Lamport
CMB	Chandy, Misra e Bryant
FDAS	Fixed Dependency After Send
FDI	Fixed Dependency Interval
FIFO	First In First Out
GVT	Global Virtual Time
LVT	Local Virtual Time
NRAS	No Receive After Send
PZCF	Partially Z-Cycle Free
RDT	Rollback-dependency Trackability
SNS	Sync-and-Stop
SZPF	Strictly Z-Path Free
UML	Unified Modeling Language
Z-Path	Caminho-Z
Z-Cycle	Ciclo-Z
ZCF	Z-Cycle Free
ZPF	Z-Path Free

Resumo

Esta tese apresenta um novo protocolo otimista, denominado *Rollback* Solidário, que utiliza o conceito de *checkpoints* globais consistentes para melhorar a sincronização dos processos durante o procedimento de *rollback*, permitindo aumentar o desempenho da simulação e realizar um gerenciamento mais adequado da memória. Neste protocolo, não existe o conceito de anti-mensagens e, portanto, não acontecem *rollbacks* em cascata, pois a decisão do retorno é tomada com base nas informações de todos os processos. Para a obtenção dos *checkpoints* globais consistentes, o protocolo *Rollback* Solidário pode utilizar a abordagem síncrona ou a semi-síncrona, com destaque para o algoritmo *Fixed Dependency After Send.* O protocolo proposto é descrito e a especificação para implementação, através da UML (*Unified Modeling Language*), é fornecida. Uma comparação entre os protocolos *Rollback* Solidário e *Time Warp* é apresentada.

Abstract

This thesis presents a novel optimistic protocol named Solidary Rollback Protocol, that utilizes the concept of consistent global checkpoints aiming at improving process synchronization during the rollback procedure, thus allowing improving simulation performance and realizing a more adequate memory management. In this new synchronization protocol does not exist the concept of anti-messages and therefore cascade rollbacks are not possible, once the return decision is based on the information from all processes. The consistent global checkpoints are built from the Solidary Rollback protocol either based on a synchronous or semi-synchronous approach, with emphasis on the Fixed-Dependence After-Send algorithm. The proposed protocol is described and a specification looking at implementation is made through the adoption of UML (Unified Modeling Language). A comparison between the Solidary Rollback and the Time Warp protocols is also presented.

Capítulo 1

Introdução

O Processamento Paralelo, também conhecido como Computação de Alto Desempenho, tem sido usado para diminuir o tempo gasto na execução de diversas aplicações cujas versões seqüenciais apresentam desempenho insatisfatório.

A Simulação é uma das áreas da computação que têm aproveitado o poder computacional das máquinas paralelas e dos sistemas distribuídos para melhorar o tempo de resposta dos programas de simulação. Essa área é bastante abrangente, estando presente em diversos campos de atuação, destacando-se as áreas de meteorologia, saúde, engenharia, indústria, militar (FUJIMOTO, 2001; TAYLOR et al., 2002; FUJIMOTO, 2003; ROBINSON, 2005), redes de telecomunicações e de longa distância (ALLEYNE; TROPPER, 2000; KELLY et al., 2000; SIMMONDS; BRADFORD; UNGER, 2000; JI et al., 2004), etc.

A Simulação tem sido largamente empregada na avaliação de desempenho de sistemas computacionais, devido à sua flexibilidade e baixo custo (PEGDEN; SHANNON; SADOWSKI, 1995), pois é possível analisar o comportamento de um sistema real em diversas situações de uso, o que geralmente é realizado nas fases finais dos testes para a implantação do sistema. Entretanto, a Simulação auxilia a prevenir o comportamento de um sistema sem a sua existência real, antes que o investimento na compra de *software* e *hardware* seja realizado.

De forma geral, uma simulação baseia-se em um modelo que representa o comportamento de um sistema e que pode ser traduzido para um programa executado em um computador. Ela é idealizada para representar, o mais fielmente possível, um sistema real (SHAY, 1996), ou seja, a partir de um programa computacional gerador de um conjunto de resultados, que devem representar corretamente as características do sistema, pode-se exercitar o modelo e obter dados que indicam o desempenho e o comportamento do sistema simulado (SOARES, 1990).

Como a complexidade dos sistemas computacionais resulta em modelos também com-

plexos e a solução através da simulação seqüencial pode tornar o tempo de processamento inviável para a sua análise, o processamento distribuído tem sido usado para diminuir esse tempo. Nesse caso, a simulação é dividida em diversos processos e cada um deles é executado em paralelo, adotando assim a Simulação Distribuída (FUJIMOTO, 2000). Desta forma, a simulação distribuída utiliza conceitos de simulação e de programação paralela visando minimizar o tempo necessário para a execução dos programas de simulação.

Na simulação distribuída, como será discutido nos capítulos posteriores, a sincronização dos processos é uma tarefa complexa. Essa complexidade favoreceu a criação de vários protocolos para garantir a integridade dos dados sem perder a eficiência na utilização dos diversos processadores, o que nem sempre é possível. Os problemas inerentes aos protocolos para simulação distribuída surgem porque não existe um relógio global de referência entre as tarefas da aplicação e nem estados compartilhados.

Os protocolos desenvolvidos foram classificados como conservativos ou otimistas (FUJIMOTO, 2003). Para evitar erros de causa e efeito, os protocolos conservativos impõem uma sincronização explícita entre os processos lógicos. Desta forma, um evento somente será considerado pelo sistema quando for possível garantir que se trata de um evento seguro (PARK; FUJIMOTO; PERUMALLA, 2004). Nos protocolos otimistas há maior liberdade entre os processos na execução de suas tarefas. O sincronismo ocorre quando há identificação de um erro de causa e efeito. Nessa situação, os processos corrigem o problema através da restauração de um estado anteriormente armazenado, procedimento conhecido como *rollback*. Se determinado processo, que está realizando um *rollback*, enviou mensagens para outros processos com tempo lógico maior que o tempo para onde ele está retornando, estas mensagens precisam ser canceladas. Por conseguinte, nos protocolos otimistas tradicionais, o processo envia anti-mensagens avisando do cancelamento dos respectivos eventos. Ao receber uma anti-mensagem, primeiramente o processo verifica se a mensagem correspondente ainda não foi processada. Nesse caso, ele simplesmente apaga a mensagem da lista de eventos futuros, caso contrário também realiza um *rollback*. Esse procedimento reconstitui toda a simulação e pode ser visto como um mecanismo de sincronização implícito dos protocolos otimistas (FUJIMOTO, 1990). Entretanto, como pode-se perceber, essa sincronização não ocorre de uma só vez devido ao efeito cascata dos *rollbacks* quando há o envolvimento de diversos processos. Essa situação desperdiça ciclos dos processadores e prejudica o desempenho da simulação.

1.1 Motivação

Analisando-se o histórico apresentado nos parágrafos anteriores e a vasta bibliografia encontrada nessa área ao longo dos últimos 20 anos, observa-se que, apesar do grande

desenvolvimento dos protocolos para simulação distribuída, a maioria dos trabalhos têm como base os mesmos paradigmas. Para o caso dos protocolos conservativos, tem-se o CMB (BRYANT, 1977; CHANDY; MISRA, 1979) e suas variantes (MISRA, 1986; REED; MALONY; MCCDREDIE, 1988; LIU; TROPPER, 1990; CAI; TURNER, 1990; BOUKERCHE; TROPPER, 2001; TEO; NG; ONGGO, 2002) que consideram o tratamento de *deadlocks*, número de mensagens no sistema e a importância de se determinar um intervalo seguro para o processamento dos eventos (*lookahead*). No entanto, a rigidez na forma de sincronização desses protocolos dificulta a obtenção de bons resultados. Na abordagem otimista, o *Virtual Time* (JEFFERSON, 1985) tem sido a base para o sincronismo, apesar das tentativas de reduzir o custo computacional na execução do procedimento de *rollback*, através da diminuição do otimismo da aplicação (MCAFFER, 1990; PRAKASH; SUBRAMANIAN, 1991; STEINMAN, 1993a; BELLENOT, 1993; FERSCHA, 1995; SRINIVASAN; REYNOLDS, 1998; ISKRA; ALBADA; SLOOT, 2003) e melhorias nas técnicas de gerenciamento da memória, uma vez que este tipo de abordagem necessita que os processos armazenem seus estados para viabilizar o procedimento de *rollback* (STEINMAN, 1993b; PREISS; LOUCKS, 1995; SKOLD; RONNGREN, 1996; QUAGLIA; CORTELLESSA, 1997; CAROTHERS; PERUMALLA; FUJIMOTO, 1999; ZHANG; TROPPER, 2001; ZENG; CAI; TURNER, 2004).

A motivação para a proposta apresentada nesta tese consiste da possibilidade de se explorar uma nova abordagem de protocolo para sincronização dos processos de um programa de simulação distribuído. Esta abordagem, que pode ser classificada como otimista, baseia-se no fato de que os pontos de retorno dos processos lógicos envolvidos em um mesmo *rollback*, por construção, fazem parte de um corte consistente da computação. Desta forma, o novo protocolo procura identificar este corte, evitando os *rollbacks* em cascata.

Em sistemas de computação distribuídos, a diversidade é uma ocorrência comum. Arquiteturas diferentes compartilham recursos dispersos e, em conseqüência, não é fácil criar regras que possibilitem maximizar o uso dessa heterogeneidade em benefício do conjunto.

Pode-se fazer uma analogia do protocolo apresentado nesta tese com a realidade humana. A idéia de solidariedade, que está associada à interrupção dos próprios interesses em favor daqueles que mais precisam, surge à mente quando se entende o comportamento deste protocolo. O curioso é que, como nos relacionamentos humanos, a união na tentativa de socorro àquele que está com problemas produz vantagens que beneficiam a todos.

O estudo das formas de comunicação e sincronismo em ambientes distribuídos possibilita averiguar as possibilidades de aproveitamento dos recursos computacionais naquilo que eles possuem de melhor. E, nessa busca, é necessário identificar fatores que facilitem aos mais "hábeis" o auxílio aos mais "deficientes", permitindo melhorar o desempenho das aplicações distribuídas, em especial, dos programas de simulação.

1.2 Objetivos

O principal objetivo deste trabalho é melhorar o desempenho das simulações distribuídas, através de uma sincronização mais eficiente dos processos durante o mecanismo de *rollback*.

Para esse desiderato, esta tese propõe o protocolo *Rollback* Solidário, que está fundamentado na teoria dos *checkpoints* globais consistentes, utilizada em sistemas tolerantes a falhas, produzindo uma abordagem diferente para o sincronismo dos processos durante um *rollback* em relação aos demais protocolos otimistas que alcançam o sincronismo através das anti-mensagens.

O princípio de funcionamento do protocolo *Rollback* Solidário surge da constatação de que, utilizando *checkpoints* globais consistentes, é possível retornar todos os processos que estarão envolvidos no *rollback* para um *checkpoint* global consistente, durante a ocorrência de algum erro de causa e efeito. Assim sendo, a decisão do retorno é tomada com base nas informações de todos os processos. O nome *Rollback* Solidário se originou dessa idéia, ou seja, quando um processo precisa retornar, os demais "ajudam" na solução do problema e, se necessário, retornam "juntos". O protocolo proposto pode ser classificado como otimista e a principal diferença entre ele e os outros protocolos otimistas é a ausência de anti-mensagens e *rollbacks* em cascata.

Com intuito de mostrar a viabilidade, tanto em termos de implementação como em termos de desempenho, este trabalho apresenta a especificação formal do protocolo proposto e a sua comparação com o protocolo *Time Warp*.

1.3 Estrutura da Tese

Este trabalho está organizado da seguinte forma: o próximo capítulo apresenta um estudo introdutório sobre Simulação Distribuída destacando o protocolo *Time Warp*, o protocolo otimista mais conhecido, dando ênfase na forma de sincronização dos processos durante um *rollback* e nos seus mecanismos de salvamento de estados. Com este estudo, será possível discutir as principais diferenças entre os protocolos *Time Warp* e *Rollback* Solidário.

Uma introdução sobre sincronização em sistemas distribuídos e *checkpoints* globais consistentes é apresentada no capítulo 3, abordando os conceitos necessários para a compreensão do protocolo proposto.

O capítulo 4 apresenta o funcionamento do protocolo *Rollback* Solidário e o capítulo seguinte contém o projeto, orientado a objetos, para a implementação do protocolo.

O capítulo 6 apresenta um estudo comparativo entre o protocolo *Time Warp* e o protocolo *Rollback* Solidário.

Finalmente, o último capítulo encerra a tese apresentando as contribuições do trabalho, conclusões obtidas e plano para continuidade do mesmo.

Vários termos foram mantidos conforme o original em inglês, na tentativa de não prejudicar a legibilidade do texto.

Capítulo 2

Simulação Distribuída e o Protocolo *Time Warp*

A simulação envolve a geração de um histórico artificial do sistema e a observação deste histórico permite criar inferências concernentes às características operacionais do sistema real que ele representa (BANKS, 1999).

A simulação atua sobre uma descrição do sistema, sendo essa descrição denominada de modelo. É possível criar um modelo para cada tipo de problema que pode surgir no sistema e que se deseja analisar. Portanto, pode-se ter vários modelos para o mesmo sistema, cada qual com o objetivo de resolver um determinado problema.

Segundo Soares (1990), os modelos de um sistema podem ser classificados em discretos e contínuos. Os modelos discretos representam sistemas onde as alterações no seu estado ocorrem em pontos específicos e descontínuos do tempo simulado. Os modelos contínuos, ao contrário, retratam sistemas em que as mudanças de estado podem ocorrer continuamente ao longo do tempo da simulação.

Para a representação de sistemas computacionais, tem-se que os modelos discretos são mais adequados devido, principalmente, ao comportamento discreto destes sistemas, tornando a simulação desse tipo de sistema mais flexível.

2.1 Simulação Distribuída de Eventos Discretos

Utilizar a simulação para avaliar o comportamento de um sistema computacional apresenta diversas vantagens como, por exemplo, a possibilidade de prever conseqüências causadas por mudanças no número e capacidade dos processadores, quantidade de recursos de entrada e saída, entre outras, sem a necessidade de implementar essas mudanças para verificar os seus efeitos, o que implicaria em gastos desnecessários. Todavia, um

programa de simulação executando sobre um único processador pode levar muito tempo para apresentar um resultado confiável do comportamento do sistema.

O tempo para simular um sistema depende do ambiente onde a simulação está sendo executada, da complexidade do sistema modelado, da quantidade de eventos e do número de replicações necessárias para garantir resultados estáveis. Como a carga de trabalho imposta ao sistema, que está sendo simulado, é baseada em variáveis aleatórias, é preciso garantir que haja quantidade de eventos e número suficiente de replicações para que as funções de distribuição sejam alcançadas. Este fator se torna, em muitos casos, uma desvantagem para a utilização desta técnica na avaliação de desempenho.

Como foi apresentado, para diminuir o tempo gasto pelos programas de simulação seqüencial é possível distribuir a simulação entre diversos processadores de uma máquina paralela ou sobre um sistema distribuído. Em princípio, essa idéia parece resolver o problema, mas pode-se perceber as novas implicações que passam a existir devido às características próprias deste tipo de aplicação. As novas dificuldades estão relacionadas com a necessidade de sincronização dos processos, além de outros problemas que estão intimamente relacionados com os sistemas distribuídos como, por exemplo, o balanceamento de carga e a sobrecarga na rede de comunicação.

No paradigma da simulação discreta, três estruturas de dados são utilizadas para a implementação das primitivas básicas no desenvolvimento da simulação orientada a eventos:

- As variáveis que descrevem o estado do sistema;
- Uma lista de eventos, denominada lista de eventos futuros, que contém todos os eventos pendentes para execução;
- Um relógio global que controla o progresso da simulação.

Os eventos que devem ser executados possuem uma marca de tempo (*timestamp*). Esta marca determina quando a mudança no estado do sistema deve ocorrer, ou seja, quando o evento deve ser processado. O programa de simulação, repetidamente, remove o evento com a menor marca de tempo do início da lista para executá-lo. A ativação desses eventos é controlada por um relógio global que determina o tempo da simulação. Quando o evento é retirado da lista, o relógio global é então avançado para o tempo de ocorrência do evento e o programa de simulação inicia a sua execução (MACDOUGALL, 1987; FUJIMOTO, 1990; NICOL; FUJIMOTO, 1994). É possível verificar que esse mecanismo garante que os eventos do sistema real sejam simulados em ordem cronológica no tempo de simulação.

Para a implementação da simulação distribuída, a estrutura da simulação tradicional, inerentemente seqüencial, devido à existência da lista de eventos futuros, teve que sofrer adaptações para incorporar os conceitos de computação distribuída (REED; MALONY; MCCDREDIE, 1988). Novos mecanismos foram desenvolvidos para garantir a execução correta dos eventos da simulação. O sistema passou a ser dividido em um conjunto de n processos lógicos $p_1, p_2, \ldots, p_n$, cada um representando um processo do sistema real.

Na simulação distribuída, cada processo lógico possui um relógio que indica o progresso da simulação. Todas as interações entre eles são modeladas por troca de mensagens e cada processo pode escalonar eventos para qualquer processo lógico, inclusive ele próprio. As mensagens são transmitidas por intermédio dos canais de comunicação, não existindo objetos compartilhados uma vez que cada processo possui as suas próprias estruturas de dados. Como todo o processamento dos eventos deve ser análogo ao comportamento do sistema real, os relacionamentos de causa e efeito nunca devem ser violados. Esse relacionamento determina que, se em um sistema real o evento e_1 ocorreu antes do evento e_2, então, na simulação, esta situação deve ser preservada. Segundo Chandy e Misra (1979) para garantir a correctude dos resultados, em uma simulação distribuída de eventos discretos, é suficiente executar os eventos em ordem cronológica.

Os conceitos de sincronização de processos levaram ao desenvolvimento de protocolos, classificados como conservativos ou otimistas, para garantir a sincronização entre os processos da simulação distribuída, evitando ou corrigindo erros de causa e efeito (FUJIMOTO, 2000).

2.1.1 O Protocolo Conservativo

Em um protocolo de sincronização conservativo, os tempos de ocorrência dos eventos são verificados para que nenhum deles seja executado fora da sua ordem cronológica. Os protocolos conservativos foram os primeiros a serem implementados. O maior problema desse tipo de abordagem é determinar quando é seguro executar um evento, ou seja, se a sua execução não irá provocar erros de causa e efeito.

Os primeiros algoritmos para este tipo de abordagem foram desenvolvidos, independentemente, por Chandy e Misra (1979) e Bryant (1977). Por esse motivo, o protocolo conservativo mais conhecido é denominado CMB, em homenagem aos seus criadores.

No protocolo CMB, a especificação dos canais de comunicação é realizada de forma estática, por conseguinte, um processo p_i comunica-se diretamente com um processo p_j somente se existir um canal de p_i para p_j definido antes do início da simulação.

Como os canais de comunicação são previamente conhecidos, é possível, para cada processo, determinar o tempo lógico dos seus processos vizinhos, já que cada mensagem

recebida está rotulada (*timestamped*) com o relógio local do processo emissor (LVT - *Local Virtual Time*). Com essa informação, os processos podem tratar os eventos que possuam tempo de ocorrência menor que o tempo lógico dos canais de comunicação que chegam ao processo.

O processo lógico seleciona repetidamente o canal com o menor LVT e, se houver mensagem na fila, executa-a. A ordem de processamento dos eventos estará correta porque todas as mensagens que serão recebidas no futuro terão marcas de tempo com valores maiores que o LVT, uma vez que as mensagens trafegam pelos canais em fila FIFO (*First In First Out*) e, em função disso, irão chegar em ordem cronológica de ocorrência.

Quando algum canal de comunicação não possui mensagens para serem processadas ocorre um problema indesejável: o processo é incapaz de saber qual o LVT daquele canal e fica esperando por uma mensagem. Esse tipo de situação pode levar à ocorrência de *deadlock*.

A partir da constatação de que os protocolos conservativos poderiam conduzir os processos da simulação para situações de *deadlock*, melhorias foram propostas. Misra (1986) apresentou um mecanismo utilizando mensagens nulas, que servem somente para sincronização. Elas não interferem na simulação, mas atualizam os LVTs dos canais que estão vazios e, portanto, com tempo lógico desatualizado. Uma desvantagem desse mecanismo é a sobrecarga provocada pelas mensagens nulas. Uma variação consiste em enviar mensagens nulas sob demanda e não após o processamento de cada evento. A freqüência dessa demanda pode ser dada por um temporizador ou quando o menor relógio de todos os canais for o de uma fila vazia.

Além das variações apresentadas, SRADS (REYNOLDS, 1982), *Appointments* (NICOL; REYNOLDS, 1984), *Turner Carrier-null Scheme* (CAI; TURNER, 1990) e *SPaDES/Java* (TEO; NG, 2002) são outros exemplos de protocolos conservativos.

2.1.2 O Protocolo Otimista

Ao contrário dos protocolos conservativos, que previnem a ocorrência de erros de causa e efeito, os protocolos otimistas permitem que essa regra seja violada provendo uma forma de corrigir o problema quando ele surge. Jefferson (1985) desenvolveu um mecanismo denominado *Time Warp* que, como já foi comentado, se tornou o protocolo otimista mais conhecido. Os conceitos fundamentais e mecanismos usados por outros protocolos otimistas tais como *rollback*, anti-mensagens e controle global do tempo de simulação, apareceram, primeiramente, no *Time Warp* (FUJIMOTO, 2000).

Nos protocolos otimistas, da mesma forma que nos protocolos conservativos, não existem estados compartilhados. A comunicação entre os processos é realizada através

de canais confiáveis, o que significa dizer que toda a mensagem enviada chega ao seu destino. Ao contrário dos protocolos conservativos, não é exigido que os processos lógicos enviem mensagens na ordem de seus rótulos e os canais de comunicação não garantem a entrega das mensagens na mesma ordem em que elas foram enviadas. Os processos podem ser criados e destruídos durante a execução da simulação e não existe a necessidade de se definir, explicitamente, com quais processos lógicos um determinado processo pode se comunicar.

A liberdade que os processos possuem facilita a ocorrência de erros de causa e efeito. Logo, é necessário um mecanismo de recuperação da simulação. Este mecanismo é conhecido como *rollback* e, após um processo receber uma mensagem com marca de tempo menor que o seu LVT, ele restaura o estado imediatamente anterior ao tempo lógico da mensagem, processa a mensagem recebida e continua a simulação a partir deste ponto.

Definição 2.1.1 (Mensagem *Straggler*) *Uma mensagem endereçada a um processo p_i, que possui tempo lógico menor do que o LVT do processo p_i, é chamada de mensagem straggler.*

Nos protocolos otimistas dois elementos podem comprometer o desempenho do programa de simulação:

1. A ocorrência de *rollback*, pois, além do tempo de processamento perdido na execução dos eventos desfeitos, quando um processo desfaz parte do seu processamento, ele necessita avisar os processos que, por ventura, tenham recebido mensagens suas, podendo produzir novos *rollbacks*; e

2. A necessidade de armazenar, continuamente, os estados de cada processo para posterior recuperação, o que degrada o sistema, além de ser necessário implementar um bom mecanismo para o gerenciamento da memória.

Devido à forma como é realizada a reconstituição da computação, a partir do surgimento de uma mensagem *straggler*, podem ser identificados dois tipos de *rollbacks* nos protocolos otimistas tradicionais: primários e secundários.

Definição 2.1.2 (*Rollback* Primário) *Quando um processo necessita realizar um rollback devido a uma mensagem straggler, este rollback é denominado rollback primário.*

Definição 2.1.3 (*Rollback* Secundário) *Quando um processo necessita realizar um rollback devido a uma anti-mensagem, este rollback é denominado rollback secundário.*

Nos protocolos otimistas, a ocorrência de um *rollback* primário pode provocar vários *rollbacks* secundários ou *rollbacks* em cascata. Devido a essa ocorrência, pode-se perceber que a sincronização dos processos acontece lentamente, permitindo, num primeiro momento, que aqueles que sofrerão *rollbacks* secundários continuem a sua computação, mesmo que a mensagem *straggler* já tenha sido recebida pelo processo lógico destinatário e esse já esteja recuperando o seu estado.

Diversas mudanças foram introduzidas nos protocolos otimistas para melhorar o desempenho da simulação. Srinivasan (1995) classificou os protocolos otimistas de acordo com as modificações propostas para diminuir o otimismo e minimizar a ocorrência de *rollbacks*. Segundo Srinivasan (1995) os protocolos otimistas se dividem nas seguintes classes:

- **Baseado em Janelas**: cada processo lógico executa, de forma otimista, somente os eventos cujos tempos lógicos estejam inseridos no intervalo de uma janela de tempo (SOKOL; BRISCOE; WIELAND, 1988; MCAFFER, 1990; STEINMAN, 1993a).
- **Baseado em Espaço**: os processos lógicos são particionados em *clusters*. As interações dentro de um mesmo *cluster* são tratadas de maneira otimista, enquanto que as interações entre diferentes *clusters* apresentam comportamento conservativo (GIMAC, 1989; DICKENS; REYNOLDS, 1990; BELLENOT, 1993).
- **Baseado em Penalidades**: os processos lógicos são penalizados (bloqueados) ou favorecidos (os eventos são executados de forma otimista) dependendo do comportamento dos últimos *rollbacks* (REIHER; JEFFERSON; WIELAND, 1989; BALL; HOYT, 1990).
- **Baseado em Conhecimento**: Assim que é identificado um erro de causa e efeito, o processo lógico afetado limita o seu otimismo (PRAKASH; SUBRAMANIAN, 1991).
- **Probabilístico**: um processo especial envia, periodicamente, uma mensagem probabilística para todos os processos lógicos, forçando-os a re-sincronizar, ou seja, retornar, se necessário, para o tempo lógico da mensagem (MADISETTI; HARDAKER; FUJIMOTO, 1992).
- **Baseado em Estados**: os processos tomam decisões de ajustes sobre os seus próprios otimismos, ou seja, se devem, temporariamente bloquear ou otimisticamente continuar o seu processamento, através da obtenção, direta ou indiretamente, de informações sobre estados de outros processos (LIN, 1992; DAS; FUJIMOTO, 1994; FERSCHA, 1995; SRINIVASAN; REYNOLDS, 1998).

Além da classificação de Srinivasan (1995), outras taxonomias foram propostas abordando características diferentes, além da limitação do otimismo, como, por exemplo, Das

(1996) e Spolon, Santana e Santana (1999). Em particular, o trabalho de Spolon, Santana e Santana (1999) apresenta uma convergência das outras classificações, buscando agrupar de forma natural a grande variedade dos protocolos propostos ao longo dos anos (SPOLON, 2001).

Por sua importância na comparação com o protocolo proposto nesta tese, o protocolo *Time Warp* será melhor detalhado nas seções seguintes.

2.2 *Time Warp*

O mecanismo *Time Warp* foi originalmente desenvolvido por Jefferson (1985). O otimismo desse protocolo se baseia no fato de que nenhum erro de causa e efeito irá ocorrer no futuro. Com essa premissa, os processos lógicos executam os seus eventos sem se preocupar com o andamento dos demais. Todavia, quando um evento fora de ordem é detectado, o mecanismo de sincronização é responsável por desfazer toda a computação executada de forma errônea, através de *rollbacks*, e por restaurar o estado da simulação para um estado consistente. A simulação continua a execução dos eventos a partir do estado em que foi retrocedida (FUJIMOTO, 2003).

O *Time Warp* é constituído por duas partes distintas: controle local, que é interno a cada processo e se responsabiliza em garantir que os eventos sejam executados em ordem cronológica, utilizando a política de escalonamento Menor *Timestamp* Primeiro (*Smallest-Timestamp-First policy*); e o controle global destinado ao gerenciamento de memória e cálculo do *Global Virtual Time* (GVT) (JEFFERSON, 1985).

Existem diversas implementações conhecidas do protocolo *Time Warp*, podendo ser citados o ambiente *Jade Time Warp* (BAEZNER; LOMOW; UNGER, 1994), o sistema SPEEDES (*Synchronous Parallel Environment for Emulation and Discrete Event Simulation*) (STEINMAN, 1992), o WARPED (MARTIN; MCBRAYER; WILSEY, 1996) e o *Georgia Tech Time Warp* (GTW) (DAS et al., 1994).

O comportamento de um processo no *Time Warp* é semelhante a de um programa de simulação seqüencial, ou seja, ele possui uma estrutura de dados denominada lista de eventos, que inclui todos os eventos que foram escalonados mas que ainda não foram processados. Entretanto, existem duas diferenças básicas:

- Os eventos em cada processo *Time Warp* podem resultar de mensagens que foram enviadas por outros processos da simulação;
- Um processo no *Time Warp* não descarta eventos após tê-los processados. Isto é essencial devido a uma eventual necessidade de refazer o processamento.

As mensagens carregam duas marcas de tempo: o tempo virtual de envio, que é o valor do LVT do processo emissor da mensagem, e o tempo virtual de recebimento, que representa o tempo de simulação para o evento. O tempo virtual de recebimento é chamado apenas de *timestamp* (QUAGLIA; CORTELLESSA; CICIANI, 1999).

A lista de eventos é organizada em ordem cronológica do seu tempo lógico e as mensagens que geraram estes eventos são denominadas mensagens positivas. Os eventos que já foram processados são mantidos em uma lista denominada lista de saída de mensagens. Essas mensagens contém uma marca de tempo menor ou igual ao LVT do processo, visto que já foram processadas, e são denominadas mensagens negativas ou anti-mensagens. Essa lista só é utilizada se for necessária a realização de *rollbacks*.

Em adição, os processos no *Time Warp* armazenam cópias de suas estruturas internas (estados dos processos), para viabilizar a execução do *rollback*. Cada processo possui, conseqüentemente, uma lista de estados. Quando um processo recebe uma mensagem com marca de tempo (*timestamp*) menor que o seu LVT, ele interrompe o seu processamento, recupera o estado imediatamente anterior ao tempo da mensagem na lista de estados e envia novamente as mensagens da lista de saída, sinalizando aos respectivos processos que aquelas mensagens estão incorretas e não devem ser consideradas.

A figura 2.1 ilustra a chegada de uma mensagem *straggler*. Os eventos 12, 22 e 35 já foram processados e, portanto, o tempo lógico do processo é igual a 35. O próximo evento a ser tratado é o de *timestamp* igual a 40. Neste momento, uma mensagem com marca de tempo 17 chega ao processo provocando um erro de causalidade, uma vez que ela deveria ter sido tratada antes dos eventos 22 e 35. Então, o processo deve retroceder o seu processamento para um tempo lógico menor que 17. Neste exemplo, ele deve recuperar o estado salvo imediatamente após a execução da mensagem 12, processar a mensagem 17 e continuar a partir deste ponto. Se durante o processamento das mensagens 22 e 35, houve envio de mensagens para serem tratadas por outros processos do sistema, estes últimos deverão ser avisados através das respectivas anti-mensagens.

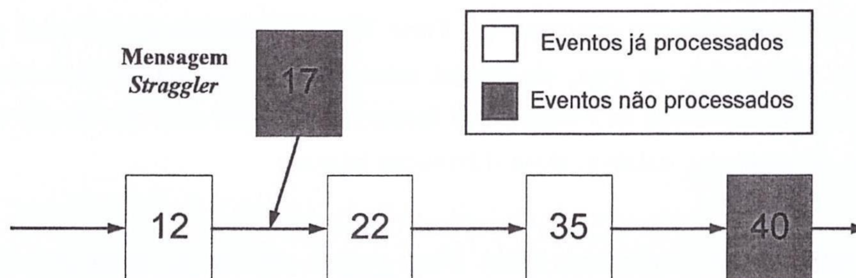

Figura 2.1: Eventos de um processo lógico durante a chegada de uma mensagem *straggler* (FUJIMOTO, 2000)

2.2.1 Anti-Mensagens

Se um processo realizar *rollback*, para cada mensagem que foi enviada com marca de tempo maior do que o ponto de retorno deverá ser enviada uma anti-mensagem correspondente. Se o processo recebe uma anti-mensagem cujo evento ainda não foi processado, ou seja, ainda está presente na lista de eventos futuros, ele remove a mensagem da lista de eventos.

Quando um processo recebe uma anti-mensagem referente a um evento já processado, ele deve fazer *rollback* (*rollback* secundário) para o estado imediatamente anterior ao tempo lógico da anti-mensagem. A repetição desse procedimento de forma recursiva permite que todos os efeitos das computações incorretas sejam cancelados (FUJIMOTO, 1990).

Outra situação que pode ocorrer, durante o recebimento de uma anti-mensagem, é a constatação de que a mensagem correspondente ainda não chegou, o que equivale dizer que a anti-mensagem chegou primeiro que a mensagem original. Isso pode acontecer se a rede de comunicação não garante a ordem de chegada das mensagens. Nessa situação, o processo armazena a anti-mensagem até que a sua versão original chegue, para então eliminar as duas.

Uma maneira conveniente de visualizar uma computação distribuída é através de um diagrama de espaço×tempo (SCHWARZ; MATTERN, 1994). Na figura 2.2, as linhas horizontais representam a execução de um processo, com o tempo progredindo da esquerda para a direita, sendo que os círculos representam os eventos. As setas representam as mensagens que são trocadas entre os processos, sendo a base da seta um evento de envio e a ponta da seta o respectivo evento de recebimento da mensagem. As setas representam, ainda, o relacionamento entre os eventos do sistema. Por conseguinte, um caminho no diagrama implica em dependência entre os eventos que o compõe. Esta dependência foi formalmente definida por Lamport (1978) como "Precedência Causal" e será discutida no próximo capítulo.

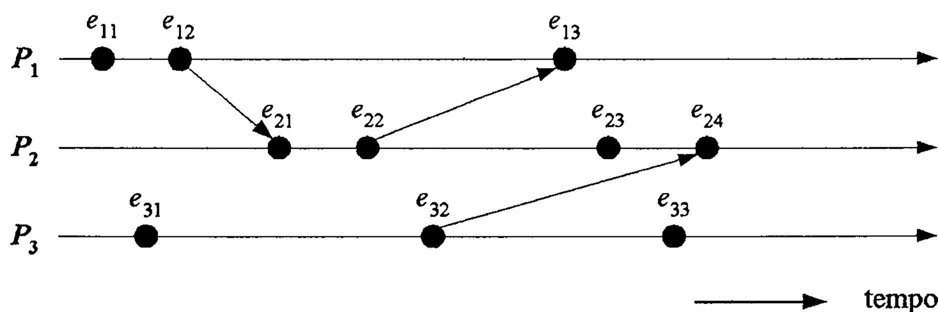

Figura 2.2: Diagrama de espaço×tempo representando uma computação distribuída

O diagrama de espaço×tempo, quando utilizado em simulação distribuída, acrescenta o conceito de relógio lógico, ou seja, as coordenadas (x, y) de cada círculo indicam o tempo lógico de cada evento, assim como o respectivo processo lógico.

Quando o processamento de um evento e é desfeito, em conseqüência de um *rollback*, todos os eventos que são causalmente dependentes de e são, da mesma forma, desfeitos, caso já tenham sido executados, ou cancelados através de anti-mensagens.

Através da ilustração da figura 2.3, Fujimoto (2000) identificou duas importantes propriedades do protocolo *Time Warp*:

1. Os *rollbacks* sempre se propagam para o tempo futuro simulado. Devido ao algoritmo de controle do relógio lógico da simulação, o grafo, representado no diagrama de espaço×tempo, é sempre acíclico. Os arcos, que representam as mensagens, sempre têm sentido da esquerda para a direita. Em outras palavras, se um processo deve realizar um *rollback* para o tempo lógico T, todas as anti-mensagens enviadas como resultado deste *rollback* devem ter um tempo maior do que T. Logo, quaisquer *rollbacks* secundários, conseqüentes destas anti-mensagens, deverão ter tempo lógico maior do que T. Essa propriedade mostra que não é possível haver efeito dominó [1], onde um *rollback* eventualmente force o retorno da simulação para o seu início.

2. Em qualquer instante, durante a execução, a computação associada ao menor rótulo de mensagem ou anti-mensagem no sistema, que ainda não foi completada, não será retrocedida. A computação associada com uma anti-mensagem é uma eliminação dos pares mensagem/anti-mensagem. Se existe mais de uma computação contendo o menor *timestamp*, esta regra se aplica a pelo menos uma destas computações. Intuitivamente, o menor *timestamp* não pode ser retornado porque os *rollbacks* se propagam da esquerda para a direita no diagrama de espaço×tempo, e não existe computação à esquerda, o que impede que algum evento force um *rollback* no menor *timestamp*. Esta propriedade é importante porque ela mostra que o *Time Warp*, executando a computação de menor *timestamp*, sempre garante o progresso da computação, o que implica que situações de *livelock* não podem ocorrer.

2.2.2 Mecanismo de Controle Global

A forma com que os processos lógicos manipulam as estruturas garante que a simulação aconteça como se fosse seqüencial, isto é, todos os eventos são processados em ordem cronológica do tempo lógico. Entretanto, segundo Fujimoto (2000), dois problemas ocorrem na aplicação:

[1] O conceito de efeito dominó será apresentado e discutido no próximo capítulo.

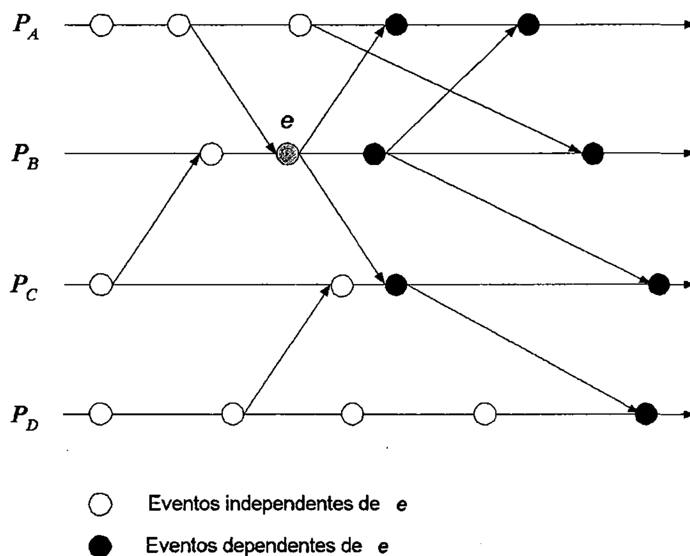

Figura 2.3: Diagrama de espaço×tempo no *Time Warp* (FUJIMOTO, 2000)

1. Com o progresso da simulação, o consumo de memória aumenta devido à criação de novos eventos. É necessário um mecanismo que gerencie a memória e possa recuperar o espaço gasto por eventos que não serão mais utilizados. Essa situação é conhecida como coleta de fóssil.

2. A execução de certas operações não podem ser refeitas, como é o caso de operações de I/O (Entrada/saída).

Ambos os problemas podem ser resolvidos se a aplicação garantir que certos eventos não estarão propensos a participar de um *rollback*. Por exemplo, se a aplicação tem certeza de que nenhum *rollback* irá ocorrer para um tempo menor do que T, as informações históricas geradas por quaisquer eventos com tempo lógico menor do que T podem ser descartadas. Igualmente, operações de I/O geradas por eventos com tempo lógico menor do que T podem ser executadas sem o perigo da operação ser refeita posteriormente.

A solução para os dois problemas é determinar um limite inferior sobre o tempo da simulação, que não está sujeito a ser refeito devido a uma situação de *rollback*. Este mecanismo é conhecido como *Global Virtual Time* (GVT). Portanto, um evento (*committed*) não poderá ser desfeito durante a ocorrência de um *rollback* se o seu tempo lógico (*timestamp*) for menor do que o GVT (SANTORO, 2003).

Definição 2.2.1 (*Global Virtual Time*) *O GVT é definido como sendo o menor valor entre os timestamps das mensagens que ainda não foram processadas ou que estão sendo processadas ou, ainda, que estão em trânsito.*

O mecanismo de controle global é responsável pelo cálculo do GVT. A partir do GVT, é possível eliminar as informações registradas com marca de tempo menor do que o seu valor. Isso garante que não irá ocorrer nenhum evento com LVT menor que o GVT. Por conseguinte, as informações mais antigas que o GVT podem ser removidas das listas onde estão armazenadas (HAGENAUER, 1999).

2.3 Mecanismos para o cálculo do GVT

Os processos não utilizam constantemente o valor do GVT, pois a freqüência com que o cálculo é realizado pode variar conforme a necessidade de liberação de espaço de memória.

Se for possível capturar, instantaneamente, uma fotografia de todos os processos, considerando o estado global do sistema, o cálculo do GVT se torna uma tarefa, relativamente, simples. No entanto, existem dois problemas referentes à obtenção desse estado global. Eles são conhecidos como "o problema da mensagem transiente" e "o problema do relatório simultâneo" (FUJIMOTO, 2000). Estes problemas serão descritos nas seções seguintes.

2.3.1 O Problema da Mensagem Transiente

Supondo que, para calcular o GVT, fosse possível congelar, simultaneamente, todos os processos no sistema e obter um relatório do estado local de cada processo e das mensagens e anti-mensagens ainda não processadas, ainda assim, o algoritmo responsável por esta tarefa poderia computar o valor do GVT de forma incorreta. A razão é simples, já que os processos estão congelados, é possível que exista uma ou mais mensagens trafegando pela rede, ou seja, mensagens que foram enviadas antes do congelamento e que ainda não chegaram aos seus destinos. A figura 2.4 ilustra uma situação que demonstra o fato de que tais mensagens podem provocar *rollback*. Essas mensagens são conhecidas como "mensagens transientes" e podem induzir um erro no cálculo do GVT. Pela figura, é possível perceber que os processos serão forçados a eliminar os estados anteriores ao tempo lógico 15. Assim, seria impossível ao processo P_B recuperar o estado anterior ao tempo lógico 10.

Fujimoto (2000) apresenta um algoritmo simples para resolver o problema das mensagens transientes. O algoritmo segue os seguintes passos:

1. Um processo "controlador" envia uma mensagem `Inicia-GVT` em *broadcast*, instruindo cada processo do sistema a iniciar a computação do GVT.

2. Ao receber a mensagem `Inicia-GVT`, cada processo interrompe a sua execução e

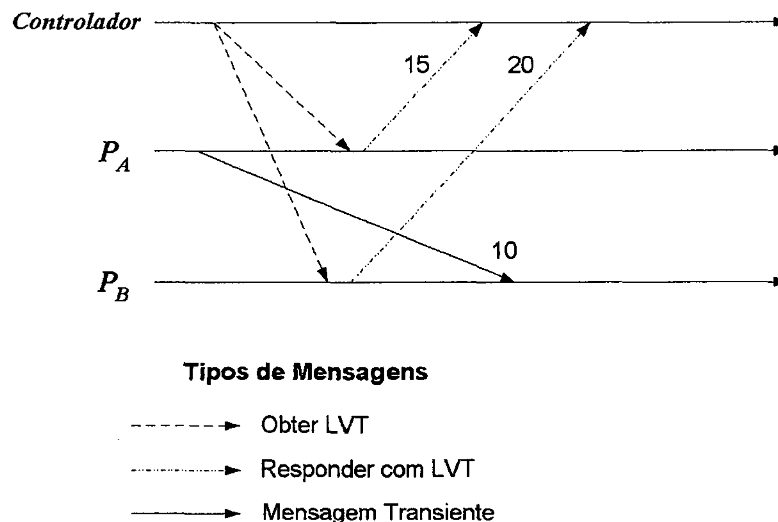

Figura 2.4: O problema da mensagem transiente (FUJIMOTO, 2000)

devolve uma mensagem de confirmação para o coordenador (`Confirma-Início-GVT`).

3. Quando o controlador receber a mensagem `Confirma-Início-GVT` de todos os processos, ele deve enviar outra mensagem, `Calcular-Mínimo-Local`, para que cada processo calcule o valor do seu limite mínimo.

4. Ao receber a mensagem `Calcular-Mínimo-Local`, cada processo calcula o tempo lógico mínimo entre:

   (a) os eventos não processados e as anti-mensagens enviadas pelo processo; e

   (b) o mínimo entre os tempos lógicos rotulados nas mensagens que o processo tenha enviado, mas ainda não tenha recebido confirmação.

5. Quando o "controlador" receber o mínimo local de cada processo, ele calcula o GVT e envia o seu valor através de uma mensagem para todos os processos.

É importante observar que, para resolver o problema das mensagens transientes, o algoritmo descrito utiliza mensagens de confirmação. A idéia principal é garantir que todas as mensagens transientes sejam reconhecidas por, pelo menos, um dos processos lógicos quando o GVT estiver sendo calculado. Se, para cada mensagem, houver uma confirmação, o emissor da mensagem fica responsável pela contabilização das mensagens transientes. Portanto, quando o controlador pedir o valor do mínimo local, o processo lógico informa o valor do menor *timestamp* das mensagens que foram enviadas mas que ainda não foram confirmados os seus recebimentos. Se o processo recebeu confirmação de todas as mensagens, ele informa ao coordenador o valor do seu LVT.

2.3.2 O Problema do Relatório Simultâneo

A solução apresentada na seção anterior introduz um novo fator que limita o desempenho do protocolo *Time Warp*: a diminuição do otimismo da simulação devido à necessidade de bloquear os processos lógicos durante o período que compreende o intervalo entre o recebimento das mensagens `Inicia-GVT` e `Calcular-Mínimo-Local`. Ademais, algum processo pode atrasar todo o processamento quando estiver executando algum evento durante o recebimento da mensagem `Inicia-GVT`.

Uma abordagem mais adequada seria permitir que os processos continuassem executando os seus eventos durante o cálculo do GVT, isto é, utilizar computação assíncrona que não possui pontos de sincronização global.

Uma forma direta para o cálculo assíncrono do GVT é fazer com que o processo controlador envie uma mensagem `Calcular-Mínimo-Local` para os demais processos e, em seguida, simplesmente colete as respostas e compute o valor do GVT. Desta forma, os processos podem continuar seu processamento enquanto o controlador calcula o GVT. A dificuldade com esta abordagem é que ela introduz um novo problema conhecido como "O Problema do Relatório Simultâneo".

Durante o cálculo assíncrono, nem todos os processos irão relatar os seus "mínimos locais" no mesmo intervalo de tempo. Logo, poderão existir mensagens na rede de comunicação cuja marca de tempo não foi considerada no cálculo do mínimo local do emissor, nem do receptor da mensagem.

Um cenário ilustrando o problema do relatório simultâneo pode ser visto na figura 2.5. O controlador envia a mensagem `Calcular-Mínimo-Local` para todos os processos. O processo P_B, por sua vez, recebe essa mensagem e envia uma resposta para o coordenador com o valor 35. A mensagem `Calcular-Mínimo-Local`, que é enviada para o processo P_A, atrasa-se na rede de comunicação. Neste intervalo, P_A envia a mensagem com *timestamp* 30 para P_B e retira da lista de eventos futuros um evento que atualiza o LVT para 40. Neste ponto, P_A recebe a mensagem `Calcular-Mínimo-Local` e responde ao controlador com o valor de 40. Pode-se verificar que o controlador irá calcular um valor incorreto para o GVT, ou seja, 35. Essa situação pode ocorrer mesmo que mensagens de confirmação sejam utilizadas.

O problema básico é a dificuldade de identificar todas as mensagens em trânsito no sistema, se os processos continuam sua execução produzindo novos eventos e, conseqüentemente, novas mensagens durante o cálculo do GVT.

Diante dos problemas das mensagens transientes e dos relatórios simultâneos, algumas soluções foram propostas como, por exemplo os algoritmos de Samadi (1985) e de Mattern (1993) que serão detalhados nas seções seguintes.

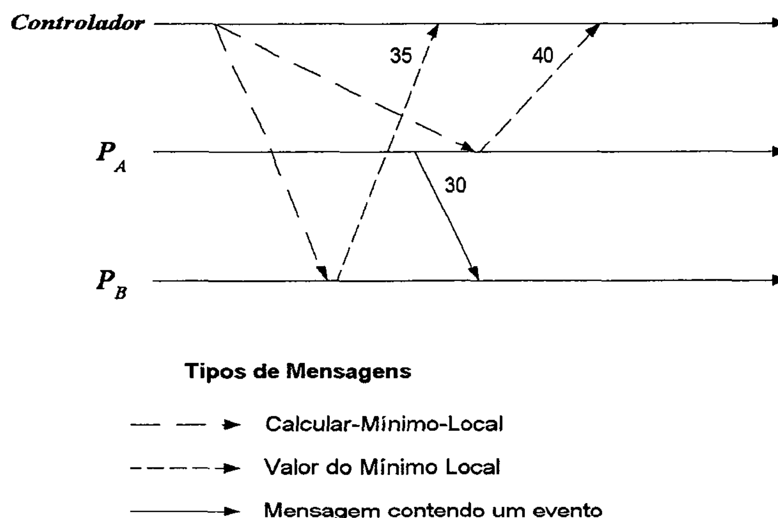

Figura 2.5: O problema do relatório simultâneo

2.3.3 Algoritmo de Samadi para o cálculo do GVT

Este algoritmo foi proposto por Samadi (apud FUJIMOTO, 2000) e assume que haverá confirmação para todas as mensagens e anti-mensagens que trafegam no sistema. O problema do relatório simultâneo é resolvido rotulando-se as mensagens de confirmação que foram enviadas no intervalo entre o último envio do mínimo local até o valor do recebimento do novo GVT. Assim, é possível identificar mensagens que estavam trafegando na rede desde o início do procedimento para o cálculo do GVT e notificar o emissor de que ele é responsável em considerar a mensagem no cálculo do seu mínimo local. Por conseqüência, na figura 2.5, o processo P_B irá rotular a mensagem de confirmação com o *timestamp* 30 e enviá-la para o processo P_A, que passará a considerar o valor 30 no cálculo do seu mínimo-local.

Em resumo, o algoritmo de Samadi funciona da seguinte forma:

1. O processo "controlador" envia uma mensagem `Calcular-Mínimo-Local` em *broadcast*, iniciando o procedimento de cálculo do GVT.

2. Ao receber a mensagem `Calcular-Mínimo-Local`, cada processo calcula o *timestamp* mínimo entre:

   (a) os eventos não processados e anti-mensagens enviadas pelo processo;

   (b) os *timestamps* de qualquer mensagem que o processo tenha enviado, mas ainda não tenha recebido confirmação; e

   (c) todas as mensagens de confirmação rotuladas recebidas desde o último cálculo do GVT.

O processo sinaliza, através de um *flag*, indicando que ele está no modo *find*.

3. Para cada mensagem ou anti-mensagem recebida pelo processo, enquanto ele está no modo *find*, é enviada uma mensagem rotulada de confirmação. Quando o processo não se encontra no modo *find*, as mensagens não são marcadas.

4. Quando o "controlador" recebe o mínimo local de cada processo, ele calcula o GVT e envia o seu valor através de uma mensagem em *broadcast* para todos os processos.

5. Ao receber o novo valor do GVT, cada processo altera a variável *flag* sinalizando que o seu modo de operação é, agora, *not find*.

2.3.4 Algoritmo de Mattern para o cálculo do GVT

Um problema encontrado no algoritmo de Samadi é a necessidade de utilizar mensagens de confirmação para todas as mensagens e anti-mensagens do sistema. Mattern (1993) apresentou um algoritmo que também evita sincronização global, mas não exige que mensagens de confirmação sejam trocadas entre os processos da aplicação.

O mecanismo proposto por Mattern faz uso do conceito de corte consistente de uma computação distribuída, que será apresentado no capítulo seguinte (seção 3.2).

A idéia básica neste algoritmo é dividir a computação distribuída, através de um "corte", em "passado" e "futuro". Cada processo define um "ponto de corte" na sua execução, com todas as ações (eventos internos e externos) antes do corte sendo referenciadas como o passado do processo e todas as ações após o corte como sendo o futuro do processo.

Uma fotografia obtida na fronteira de um corte consistente inclui os estados locais de cada processo no seu ponto de corte e todas as mensagens transientes que atravessam o corte, ou seja, todas as mensagens que foram enviadas na parte "passada" e recebidas na parte "futura" da computação. A idéia principal neste método é que uma fotografia da simulação, obtida a partir de um corte consistente, pode ser utilizada para calcular o GVT.

Para ilustrar, será considerada a execução do algoritmo síncrono para o cálculo do GVT apresentado anteriormente. Seja os pontos de corte da figura 2.6 a representação do instante em que a mensagem `Calcular-Mínimo-Local` é recebida pelos processos lógicos, o cálculo do GVT é baseado em um corte definido em um instante após todos os processos estarem congelados (figura 2.7), uma vez que, no algoritmo síncrono, há o bloqueio dos processos. O algoritmo síncrono calcula corretamente o GVT_T, onde T é o instante em que o corte é finalizado (figura 2.7).

Supondo que os processos possam executar o mesmo algoritmo sem bloqueio, ou seja, de maneira assíncrona, e que o corte realizado pelo algoritmo assíncrono é consistente.

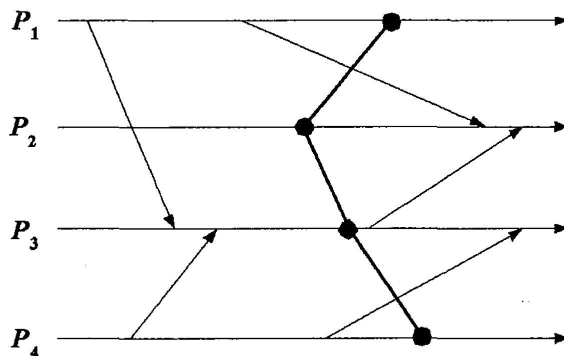

Figura 2.6: Exemplo de corte dividindo a computação em passado e futuro

Neste contexto, o conjunto de mensagens transientes no mecanismo síncrono é exatamente o mesmo do conjunto de mensagens transientes do algoritmo assíncrono. Isso ocorre, porque qualquer mensagem no mecanismo síncrono irá também aparecer na forma assíncrona e o mecanismo assíncrono não pode incluir novas mensagens, pois qualquer mensagem atravessando o corte deve ter sido enviada por um processo após o seu ponto de corte, o que poderia causar um corte inconsistente. Igualmente, qualquer estado local obtido por um processo no algoritmo assíncrono irá ser idêntico ao algoritmo síncrono, uma vez que a consistência natural do corte previne que qualquer novo evento apareça no mecanismo assíncrono, sem que ele tenha aparecido também no algoritmo síncrono.

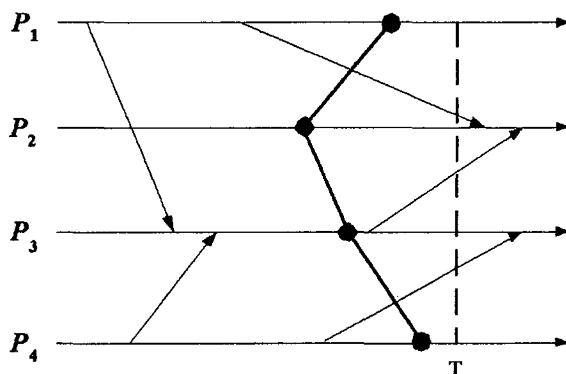

Figura 2.7: Corte da figura 2.6 ilustrando o algoritmo síncrono para o cálculo do GVT

A partir dessa discussão, observa-se que, se um corte assíncrono não for consistente, o estado global baseado neste corte não corresponderá a execução do algoritmo síncrono, pois, pelo menos um dos processos irá incluir o recebimento de uma mensagem em seu estado local, cujo o envio não apareceu no corte.

Felizmente, um algoritmo para o cálculo do GVT assíncrono não precisa construir um corte consistente. Ele pode, simplesmente, ignorar todas as mensagens que irão criar um corte inconsistente. Estas mensagens podem ser ignoradas, pois elas devem ter um

timestamp maior que o GVT calculado usando o corte consistente. Em conseqüência, para calcular o GVT deve-se somente identificar o menor rótulo de um evento não processado por cada processo no seu ponto de corte e o menor *timestamp* de qualquer mensagem transiente atravessando o corte do "passado" para o "futuro".

A parte mais complexa desse método é determinar o conjunto de mensagens transientes sem utilizar mensagens de confirmação. Isso pode ser alcançado através da utilização de dois cortes, como ilustra a figura 2.8, e calculando o GVT na fronteira do segundo corte ($C2$). A função do primeiro corte é notificar cada processo para iniciar o registro do menor *timestamp* de qualquer mensagem que ele enviou. Estas mensagens podem ser transientes que irão atravessar o segundo corte e devem ser incluídas no cálculo do GVT. O segundo corte é definido de forma a garantir que não haverá mensagens geradas antes do primeiro corte, que ainda não foram entregues ao destinatário. Dessa forma, todas as mensagens transientes no sistema, que estão atravessando o segundo corte, foram enviadas após o primeiro corte e serão incluídas no cálculo do GVT.

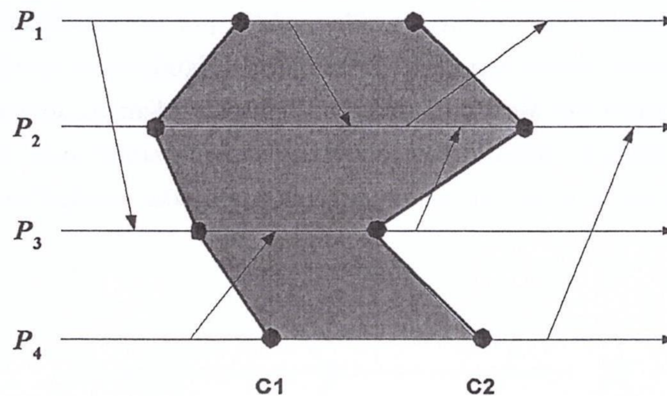

Figura 2.8: Cálculo do GVT usando dois cortes

Os processos são coloridos para denotar onde eles estão com respeito aos dois cortes. Inicialmente, os processos são coloridos de branco. Após o primeiro ponto de corte ser alcançado, a cor do processo muda para vermelho. Após o segundo corte, o processo retorna a cor branca. Mensagens que são enviadas enquanto o processo ainda é branco são denominadas mensagens brancas e mensagens enviadas enquanto o estado é vermelho são chamadas de mensagens vermelhas. Por construção, todas as mensagens brancas devem ser recebidas antes do segundo corte. Desta maneira, todas as mensagens transientes que atravessam o segundo corte devem ser mensagens vermelhas. O conjunto de mensagens atravessando o segundo corte é um subconjunto de todas as mensagens vermelhas. Para funcionar, o *timestamp* mínimo entre todas as mensagens vermelhas é um limite inferior do *timestamp* de todas as mensagens transientes que atravessam o segundo corte.

O GVT é calculado como o mínimo entre todas as mensagens vermelhas e o menor

timestamp entre as mensagens não processadas no estado global definido pelo segundo corte. Isso pode ser realizado por cada processo com base nas informações locais. Assim, a única questão que ainda persiste é se nenhuma mensagem branca atravessou o segundo corte.

O primeiro corte pode ser construído organizando os processos em um anel lógico e enviando uma mensagem de controle ao redor do anel. Ao receber a mensagem de controle, cada processo altera sua cor de branco para vermelho e envia a mensagem de controle para o próximo processo no anel. Quando o primeiro corte estiver completo, nenhuma nova mensagem branca irá ser criada no sistema. O segundo corte pode ser construído com cada processo enviando a mensagem de controle novamente através do anel. Todavia, um processo p_i não irá passar a mensagem de controle para o próximo processo enquanto ele não puder garantir que não há mensagens brancas destinadas a ele. Isso pode ser alcançado através dos seguintes passos:

1. Comparar o número de mensagens brancas enviadas para p_i;

2. Calcular o número de mensagens brancas recebidas por p_i;

3. Aguardar enquanto esses dois números não forem iguais, ou seja, aguardar enquanto todas as mensagens enviadas não tiverem sido recebidas.

Para realizar a primeira tarefa, cada processo mantém um contador indicando o número de mensagens brancas enviadas para p_i. Esses contadores são atualizados durante a criação do primeiro corte, anexando essa informação às mensagens de construção do corte. Após a ficha de controle do anel passar por todos os processos, a soma na ficha indica o número total de mensagens brancas que foram enviadas para p_i.

No segundo passo, p_i mantém um contador indicando o número de mensagens brancas que foram recebidas independentemente do processo emissor. Quando esse segundo contador é igual ao número de mensagens brancas enviadas para p_i, o processo passa a mensagem de controle para o próximo processo no anel.

Contadores, indicando o número de mensagens brancas enviadas para um processo e o número de mensagens brancas recebidas por cada processo, devem ser mantidos para cada processo no sistema. Assim, faz-se necessário a utilização de um vetor. Em suma, o processo p_i mantém um vetor local V_i tal que $V_i[j]$ $(i \neq j)$ denota o número de mensagens brancas enviadas pelo processo p_i para o processo p_j. $V_j[j]$ é um número negativo que indica o número de mensagens brancas recebidas por p_j. Quando $V_j[j] + \sum V_i[j]$, $(j \neq i) \leq 0$, então p_j já recebeu todas as mensagens brancas que foram enviadas para ele.

2.3.5 Formas de Cancelamento das Anti-mensagens

Com relação ao procedimento de envio de anti-mensagens, durante a chegada de uma mensagem *straggler*, ainda é importante considerar que existem, basicamente, três abordagens. O cancelamento de eventos através de anti-mensagens pode ser realizado por meio dos seguintes tipos (GAFNI, 1985; REIHER et al., 1990; ISKRA; ALBADA; SLOOT, 2003):

Cancelamento Agressivo: quando um processo efetua *rollback* para o tempo T, anti-mensagens são, imediatamente, enviadas para desfazer todas as mensagens anteriormente submetidas com marca de tempo maior do que T.

Cancelamento Preguiçoso: neste tipo de cancelamento, os processos não enviam as anti-mensagens imediatamente após o *rollback.* Ao contrário, esperam para verificar se a nova execução da computação produz os mesmos eventos. Neste caso, não existe a necessidade de efetuar o cancelamento.

Cancelamento Dinâmico: utiliza uma técnica na qual cada processo decide qual das duas estratégias de cancelamento anteriores ele deseja utilizar.

2.4 Mecanismos de Salvamento de Estado

O mecanismo de salvamento de estados constitui uma parte importante em qualquer sistema que necessita restaurar a computação em caso de falhas. Uma das maiores vantagens dos protocolos otimistas para Simulação Distribuída é o fato de que os processos não são bloqueados durante o processamento, aumentando o paralelismo da aplicação. Entretanto, o custo computacional para se realizar *rollbacks* e a quantidade de memória necessária para armazenar as estruturas de dados são as maiores desvantagens desta abordagem. Mesmo assim, alguns estudos indicam que as vantagens dos protocolos otimistas são maiores que as desvantagens quando comparadas aos protocolos conservativos (FUJIMOTO, 1990; LIN; LAZOWSKA, 1990; RöNNGREN et al., 1996).

Para suportar o mecanismo de salvamento de estados é necessário que o sistema armazene informações suficientes a respeito dos processos para garantir que qualquer estado possa ser recuperado em caso de *rollback.* O mecanismo de salvamento de estados deve, por conseguinte, ser bem projetado uma vez que ele interfere no desempenho do sistema, além, é claro, do gasto de memória (RöNNGREN et al., 1996).

O método mais simples de gerenciamento de memória, proposto para o *Time Warp*, denomina-se *Copy State Saving.* A sua simplicidade reside no fato do mecanismo armazenar todos os estados dos processos a cada evento. A figura 2.9 ilustra o salvamento e

a recuperação do estado de um processo para o cenário do *rollback* apresentado na figura 2.1. O estado do processo inclui três variáveis, x, y e z que detém, respectivamente, os valores 1, 2 e 3 após o processamento do evento de tempo lógico 12. O evento de *timestamp* 22 altera o valor da variável x para 4 e o evento 35 atribui 5 para x e 9 para z. Como apresentado pela figura 2.9, o estado completo do processo é armazenado na lista de estados. Quando a mensagem *straggler* com marca de tempo igual a 17 chega, o processo lógico recupera o seu estado antes da execução do evento de *timestamp* 22.

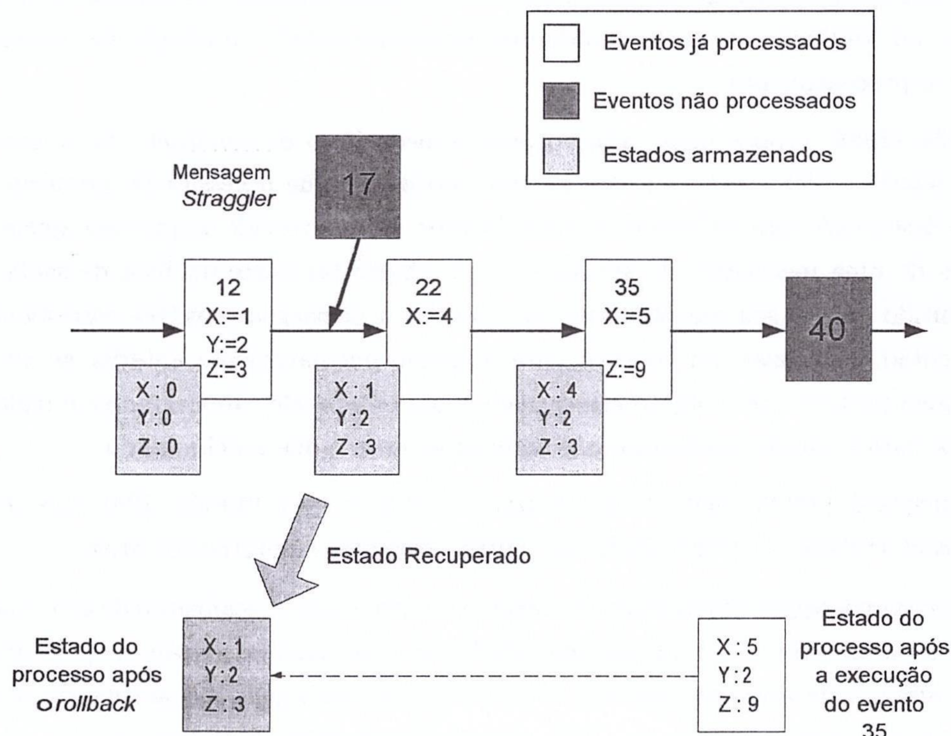

Figura 2.9: Exemplo de funcionamento do mecanismo *Copy State Saving* (FUJIMOTO, 2000)

Se o protocolo otimista necessitar de poucos *rollbacks* a maioria dos eventos salvos não será utilizada, resultando em desperdício de memória e de tempo gasto durante o armazenamento dos estados.

Diversos esquemas de gerenciamento têm sido propostos para reduzir o espaço de memória usado pelo *Time Warp*. Das e Fujimoto (1997) classificaram estas abordagens em duas categorias: passivos e ativos.

Esquemas de gerenciamento de memória passivos reduzem o gasto de memória utilizado, mas abortam o programa sempre que o espaço exigido pela aplicação ultrapassar a quantidade de memória disponível. *Copy State Saving*, *Incremental State Saving*, *Sparse State Saving* e *Hybrid State Saving* são exemplos deste tipo de estratégia.

Ao contrário dos esquemas passivos, os mecanismos de gerenciamento de memória ativos estão aptos a executar a simulação sem a quantidade de memória necessária para o funcionamento da simulação. Esses mecanismos exigem um volume mínimo para iniciar a simulação e recuperam áreas de memória sob demanda quando necessárias.

Mensagens de *sendback*, conforme proposto por Jefferson (1985) em seu trabalho original, é um exemplo de protocolo ativo. Quando uma mensagem é recebida por um processo e não existe memória suficiente para armazená-la, o processo envia a mensagem para o seu local de origem, ou seja, para o seu processo emissor. O emissor deve, então, realizar um *rollback* para o seu estado imediatamente anterior a criação da mensagem e repetir o processamento.

Gafni (1988) propôs uma variação para o mecanismo de *sendback*. O protocolo de Gafni remove objetos armazenados por um processo p que necessita de memória. Se o objeto descartado está na lista de eventos futuros, ele é devolvido ao processo que o gerou através de uma mensagem de *sendback*. Se o objeto faz parte da lista de saída, ele é transmitido para o seu receptor, que irá cancelar a mensagem positiva correspondente, para, então, p realizar um *rollback* para o estado imediatamente anterior ao envio da mensagem positiva. Se o objeto armazenado é um estado, ele é descartado e p realiza um *rollback* para o estado imediatamente anterior ao estado que foi eliminado.

Cancelback (JEFFERSON, 1990), *Artificial Rollback* (LIN; PREISS, 1991; LIN, 1992) e *Pruneback* (PREISS; LOUCKS, 1995) são outros exemplos de protocolos ativos.

As próximas seções apresentam um resumo dos principais mecanismos de gerenciamento de memória passivos, visto que são mais eficientes, uma vez que partem do princípio que, com a limpeza de memória (*garbage collection*) promovida pelo mecanismo de gerenciamento global, após o cálculo do GVT, sempre haverá memória suficiente para terminar a simulação e, portanto, são mecanismos adequados para a comparação com o protocolo *Rollback* Solidário proposto nesta tese.

2.4.1 *Incremental State Saving*

Em muitas simulações, o tamanho dos estados que devem ser salvos é relativamente grande, o que degrada ainda mais o desempenho do sistema quando se utiliza o mecanismo *Copy State Saving*. Para resolver este problema foi proposto por Steinman (1993b) o método *Incremental State Saving*, que armazena apenas as informações que serão modificadas a cada evento, agilizando o procedimento de salvamento dos estados.

A figura 2.10 apresenta o mesmo exemplo ilustrado pela figura 2.1 com a utilização do método *Incremental State Saving*. Durante a simulação, apenas as variáveis que serão modificadas devem ser salvas. Isso pode ser evidenciado pela figura 2.10. Antes de se

executar o evento de *timestamp* 22, apenas o valor da variável x é salvo, uma vez que este evento somente altera o seu valor. Da mesma forma, as variáveis x e y são salvas antes da execução do evento de tempo lógico 35. Entretanto, durante um *rollback*, não é possível reconstituir, diretamente, o estado do processo após a execução do evento 12, já que somente o valor da variável x foi armazenado nesse intervalo. Portanto, a restauração do estado é realizada de forma decremental. Restaura-se o estado imediatamente anterior ao estado atual do processo e, assim, sucessivamente, até reconstituir o estado necessário à continuação da simulação.

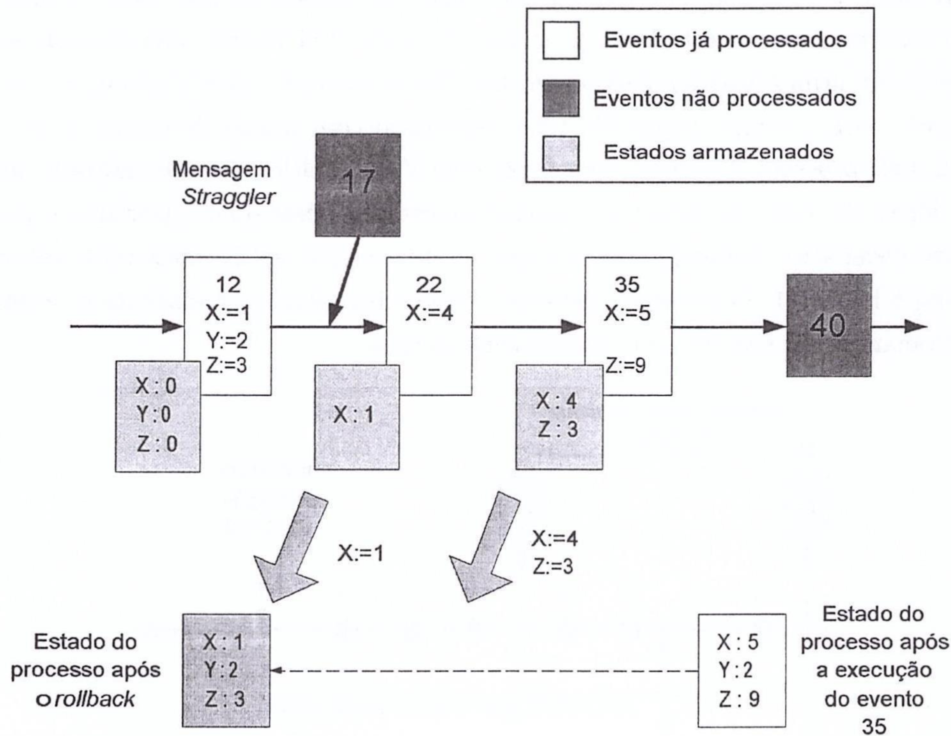

Figura 2.10: Exemplo de funcionamento do Mecanismo *Incremental State Saving*

A recuperação decremental é uma desvantagem deste mecanismo, pois a necessidade de reconstituir o estado do processo, através do procedimento inverso de salvamento, prejudica o mecanismo de *rollback* degradando o desempenho do sistema.

2.4.2 *Sparse State Saving*

Um potencial ponto de gargalo do *Time Warp* é a necessidade de, periodicamente, salvar os estados de cada processo lógico para, no futuro, ser possível recuperar estados anteriores (SKOLD; RONNGREN, 1996). No entanto, para garantir que a simulação possa ser restaurada nas situações de *rollback*, os estados da computação não necessitam ser, continuamente, armazenados após o processamento de cada evento (RöNNGREN; AYANI,

1994). No método *Sparse State Saving*, também chamado de *Infrequent State Saving* (LIN et al., 1993) ou ainda *Periodic State Saving* (PREISS; MACINTYRE; LOUCKS, 1992; FLEISCHMANN; WILSEY, 1995), o salvamento dos estados é realizado em intervalos maiores de tempo, englobando o resultado de vários eventos (BELLENOT, 1992). Em caso de *rollback*, a computação é restaurada pela recuperação do último estado armazenado antes do tempo lógico para onde deve retornar a computação do processo. Em seguida, é realizado um processamento (*coast forward*) até o ponto de retorno, para recuperar o estado desejado. Durante esta fase nenhum evento de envio de mensagens é realizado pelo processo. Isso acontece porque a re-execução dos eventos na fase *coast forward* serve apenas para recompor o estado do processo. A figura 2.11 ilustra uma situação em que os estados são armazenados a cada 4 eventos. Neste exemplo, após a execução do evento escalonado para o tempo lógico 35, uma mensagem com marca de tempo igual a 20 é recebida pelo processo. O evento com *timestamp* 17 foi o último evento realizado antes do tempo lógico 20, mas não houve salvamento de estados nesse ponto, portanto o processo deve retroceder até o último estado salvo que, neste exemplo, está representado pelo evento com tempo lógico 11. O processo restaura o respectivo estado e re-executa os eventos 14 e 17 retomando, em seguida, o processamento normal.

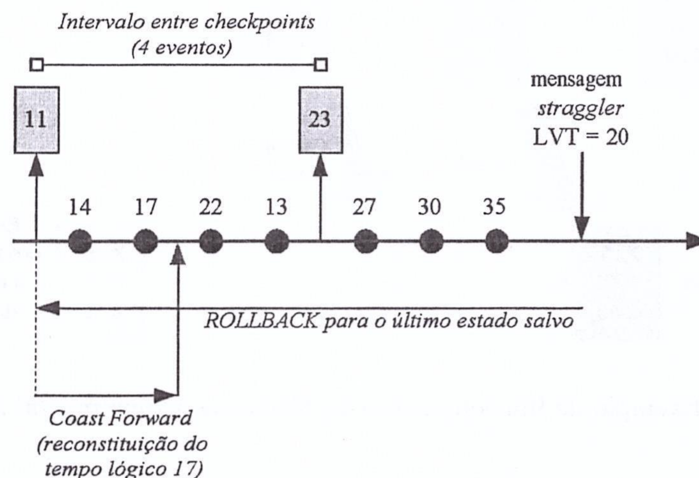

Figura 2.11: Ilustração do mecanismo *Sparse State Saving*

O mecanismo *Sparse State Saving* apresenta duas vantagens em relação aos mecanismos apresentados anteriormente. A primeira delas é a economia de memória utilizada durante o salvamento de estados. A segunda vantagem é o desempenho superior quando o número de *rollbacks* não é grande, pois perde-se pouco tempo no salvamento de estados, apesar da necessidade de processar eventos na fase *coast forward.*

Um problema encontrado com este mecanismo é a dificuldade em se definir adequadamente o intervalo de eventos entre os salvamentos de estados, pois se a freqüência do salvamento de estados é reduzida, o tempo para salvar os estados também é reduzido

e, conseqüentemente, o desempenho da simulação tende a ser maior. Entretanto, o desempenho pode ser prejudicado por longos *rollbacks* e pela necessidade de executar vários eventos na região *coast forward.* Neste contexto, alguns estudos foram realizados destacando-se o trabalho de Lin et al. (1993).

O objetivo principal das pesquisas com este mecanismo é a identificação de um ponto de equilíbrio entre o custo para o salvamento dos estados, após a execução de cada evento, e o custo computacional da recuperação de um estado através da re-execução dos eventos da região *coast forward* (PREISS; LOUCKS; MACINTYRE, 1994).

Em adição, vários estudos foram desenvolvidos com intuito de comparar os métodos *Incremental State Saving* e *Sparse State Saving*, com destaque para os trabalhos de Palaniswamy e Wilsey (1993), Rönngren et al. (1996) e Soliman (1999).

2.4.3 *Event Sensitive State Saving*

Segundo Skold e Rönngren (1996), a abordagem *Sparse State Saving* pode ser dividida em duas:

1. *Fixed Sparse State Saving*, onde a escolha do intervalo entre os *checkpoints* é mantida constante durante a simulação, e
2. *Adaptive Sparse State Saving*, em que o intervalo entre os salvamentos de estados é adaptado dinamicamente de acordo com determinados parâmetros monitorados durante a simulação.

Esta separação é importante uma vez que o tamanho do intervalo afeta diretamente o comportamento do *rollback* e, portanto, o desempenho da simulação. Skold e Rönngren (1996) observaram que os modelos adaptativos, para cálculo do intervalo entre *checkpoints*, dependem de uma estimativa do tempo de execução para a região *coast forward* e da estimativa do tempo de salvamento dos estados. Entretanto, para aumentar a precisão dessas estimativas, a variança entre esses tempos não deve ser grande, ou seja, todos os tipos de eventos devem ter desempenho semelhante quando em comparação com tempo de execução e o tempo necessário para a re-execução dos eventos na fase *coast forward.* Desta forma, os autores propuseram um mecanismo para o salvamento de estados denominado *Event Sensitive State Saving* que procura melhorar o desempenho da simulação considerando classes ou tipos de eventos que, tipicamente, possuem grande variação no tempo utilizado para processar os eventos da região *coast forward.*

Os autores acreditam que o tempo despendido durante a fase *coast forward* pode ser reduzido se for possível diminuir a probabilidade de ocorrer eventos de granularidades

diferentes nesta fase. Além disso, eles observaram que a variação das características dos eventos é um importante fator para a definição dos intervalos entre *checkpoints*. Portanto, os autores criaram um modelo de salvamento de estados sensível aos diferentes tipos de classes de eventos, cujos membros exibem comportamentos similares durante os *rollbacks*.

2.4.4 *Hybrid State Saving*

Este método foi proposto por Quaglia e Cortellessa (1997) com o objetivo de aproveitar as vantagens dos métodos *Incremental State Saving* e *Sparse State Saving*. De acordo com esta técnica, um processo lógico salva periodicamente seus estados, mas também habilita o salvamento por meio do método incremental, quando alterações dos seus estados ocorrem dentro do intervalo entre dois *checkpoints*. Desta forma, os processos lógicos estão aptos a recuperar estados através dos dois métodos descritos anteriormente.

Durante um *rollback*, o mecanismo de gerenciamento de memória pode escolher um retorno através do método *Incremental State Saving*, se a distância entre o estado atual e o ponto de retorno não for grande, ou utilizar o mecanismo *Sparse State Saving*, caso contrário. Na figura 2.12 os salvamentos dos estados do processo são realizados após 6 eventos ($X = 6$), mas, a partir do quarto evento ($Y = 2$), o método incremental é aplicado, salvando-se apenas as variáveis que serão modificadas pelo tratamento do evento escalonado. Neste exemplo, se for preciso restaurar um estado após o quarto evento dentro de um intervalo entre *checkpoints*, o processo pode retornar através do procedimento decremental, utilizado no mecanismo *Incremental State Saving*, pois o custo computacional é menor do que processar quatro eventos da fase *coast forward*. Por sua vez, se o processo deve realizar um retorno para um tempo lógico próximo ao início do intervalo, o procedimento utilizado será o descrito pelo mecanismo *Sparse State Saving*.

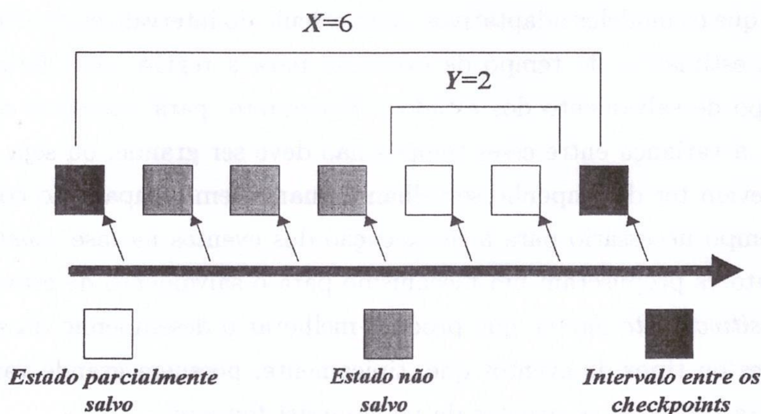

Figura 2.12: Ilustração do Mecanismo *Hybrid State Saving*

A grande vantagem deste método é a sua flexibilidade, podendo se comportar tanto como o mecanismo *Sparse State Saving* quanto como o mecanismo *Incremental State Saving*, permitindo ainda, uma escolha dinâmica entre os mecanismos de salvamento de estados durante a simulação.

2.4.5 *Reverse Computation*

Em contraste com os mecanismos de salvamento de estados anteriores, *Reverse Computation* é uma técnica diferente, na qual o *rollback* é realizado pela execução de operações individuais inversas das que foram feitas pelos eventos dos processos. Esta abordagem garante que as operações inversas reconstituem o estado desejado pelo processo (CAROTHERS; PERUMALLA; FUJIMOTO, 1999; AKGUL; MOONEY, 2004).

Uma vantagem desta abordagem consiste no menor gasto de memória em relação aos mecanismos de salvamento de estados tradicionais. Aparentemente, a computação reversa evita a necessidade de armazenar informações de estados. Entretanto, dependendo da estrutura do evento, para executar a respectiva computação reversa, algumas informações sobre o fluxo de execução podem ser necessárias como, por exemplo, aquelas que serão usadas para desfazer instruções condicionais. Dependendo da complexidade da estrutura do evento, o tamanho destas informações pode ser maior que o próprio estado do processo. Neste caso, o uso de alguma técnica de armazenamento, como *Sparse State Saving* ou *Copy State Saving*, será mais conveniente. Além disso, algumas operações podem não ser reversíveis e, por conseguinte, será obrigatório o uso de algum mecanismo de salvamento de estados (SANTORO, 2003).

2.5 Considerações Finais

Há 20 anos foi publicado o artigo intitulado "Virtual Time" por Jefferson (1985), em que o autor apresentou o protocolo otimista *Time Warp*. Desde então, diversos trabalhos foram desenvolvidos com o objetivo de melhorar o desempenho desse protocolo. Com o passar dos anos vários problemas foram identificados e soluções foram apresentadas. Ainda hoje, vários artigos têm sido publicados tentando encontrar novas soluções para alguns dos seus tradicionais problemas como, por exemplo, novos mecanismos para o gerenciamento da memória (VEE; HSU, 2002; CHUNG; XU, 2002; LI; TROPPER, 2004) e modificações no procedimento de *rollback* (ZHANG; TROPPER, 2001; ZENG; CAI; TURNER, 2004). Mudanças no protocolo também foram propostas, como no trabalho de Kalantery (2004), que introduziu a idéia de um mecanismo de comunicação orientada a conexão entre os processos lógicos, semelhante àqueles utilizados nos protocolos conservativos. Esses esforços demonstram que o protocolo *Time Warp* ainda está em evolução.

A revisão apresentada neste capítulo é importante para a compreensão das pesquisas que são realizadas na área de Simulação Distribuída e, principalmente, para compreender o comportamento dos protocolos otimistas nesse tipo de aplicação. Este entendimento permitirá avaliar as vantagens e desvantagens do protocolo proposto nesta tese em relação aos demais protocolos otimistas tradicionais.

O próximo capítulo apresenta um estudo sobre *checkpoints* globais consistentes, base para a compreensão do protocolo *Rollback* Solidário.

Capítulo 3

Sincronização em Sistemas Distribuídos

Os sistemas distribuídos têm se tornado parte presente na vida das pessoas, já que gerenciam recursos que são utilizados por grande número delas, como os terminais eletrônicos dos bancos. Os esforços dos pesquisadores e das grandes companhias de *software* em aperfeiçoar e resolver os problemas inerentes a esse tipo de sistema têm apresentado resultados rápidos, visto que as soluções encontradas fazem parte de sistemas mais simples como os que são utilizados no comércio em geral e, principalmente, na Internet.

Apesar da crescente presença na vida das pessoas, os sistemas distribuídos ainda não são tão comuns como os sistemas centralizados. Isso se deve a vários fatores, mas principalmente, pela maior dificuldade de se desenvolver aplicações confiáveis para este tipo de sistema.

Segundo Oliveira, Carissimi e Toscani (2000), os sistemas operacionais devem ser eficientes e convenientes. Obtém-se eficiência através de uma boa distribuição dos recursos do sistema entre os diversos usuários e alcança-se conveniência escondendo do usuário os detalhes de acesso aos recursos do sistema. É fácil perceber que esses dois objetivos são mais difíceis de serem alcançados em um ambiente onde os elementos que o compõem estão fisicamente separados. Como, freqüentemente, os sistemas distribuídos precisam realizar tarefas que necessitam de um comportamento coordenado, as técnicas de sincronização neste tipo de sistema têm sido largamente estudadas.

Em sistemas com uma única CPU, regiões críticas, exclusões mútuas, e outros problemas de sincronização constituem dificuldades que são geralmente resolvidas usando métodos tradicionais como semáforos e monitores (AXFORD, 1989; BEN-ARI, 1990; SHAY, 1996; TANENBAUM, 1992). Estes métodos não são adequados para os sistemas distribuídos, uma vez que os processadores não compartilham o mesmo espaço de endereçamento.

Para tratar de problemas que envolvem o sincronismo entre processos de um sistema distribuído, em especial nos protocolos para a simulação distribuída, é preciso apresentar alguns conceitos referentes às técnicas que garantam que a observação remota do sistema não seja obsoleta, incompleta ou inconsistente, uma vez que os estados se alteram rapidamente e não existe um relógio global de referência.

3.1 Estados Locais e Estados Globais

Para se ter uma fotografia completa de uma aplicação distribuída é preciso obter o estado de cada processo que a compõe. Os processos de uma aplicação distribuída têm a sua execução modelada como uma seqüência de eventos, sendo o k-ésimo evento executado pelo processo p_i representado por e_i^k.

Os eventos são classificados como eventos internos e eventos externos ou de comunicação. Os eventos externos representam o envio ou a recepção de uma mensagem, sendo todos os outros eventos considerados internos.

Definição 3.1.1 (Estado Local) *O estado local σ_i^k de um processo p_i é definido como sendo os valores das variáveis do processo p_i após a execução do seu evento e_i^k.*

A partir da definição de estado local é possível estender esta definição para estado global.

Definição 3.1.2 (Estado Global) *Estado global é uma n-tupla de estados locais $\Sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$, sendo um estado para cada processo.*

Em 1978, Lamport (1978) apresentou um modelo com base na relação de causalidade entre as tarefas de um sistema distribuído e simplificou o problema da falta de um relógio global. O autor observou que em um sistema centralizado o conhecimento do tempo real, na maioria das vezes, é irrelevante para a ordenação dos eventos de uma aplicação, pois esses eventos já estão implicitamente ordenados, visto que um processador realiza uma instrução de cada vez. Assim, Lamport concluiu que também em um sistema distribuído o que importa, na maioria das aplicações, é saber qual a ordem de execução dos eventos e não o tempo real de cada um deles. O que Lamport propôs então, foi a discretização do tempo, ou seja, um sistema distribuído deve ser visto como uma sucessão de eventos discretos, em seqüência, tornando possível a construção de uma relação que captura causalidade entre os eventos. A esta relação Lamport denominou como "Precedência Causal".

Definição 3.1.3 (Precedência Causal) *A expressão $a \rightarrow b$ é lida como "**a** precede **b**" e significa que todos os processos concordam com o fato de primeiro acontecer o evento **a** e depois ocorrer o evento **b**. A relação de causa e efeito ocorre se:*

1. *a e b são eventos no mesmo processo e se a ocorre antes de b;*

2. *a é o evento do envio de uma mensagem por um processo e b é o evento da mesma mensagem sendo recebida pelo processo de destino.*

A relação de precedência causal é uma relação transitiva, isto é, se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$. Além disso, se dois eventos x e y acontecerem em processos diferentes e não trocarem mensagens entre si, nem mesmo indiretamente, através de um terceiro processo, então, nem $x \rightarrow y$, nem $y \rightarrow x$. Tais eventos são, desta forma, considerados concorrentes ($x \| y$ ou $y \| x$).

Pode-se verificar no diagrama da figura 3.1 que o evento e_2^1 antecede o evento e_3^6, uma vez que existe o caminho composto pelos eventos e_2^1, e_1^2, e_1^3, e_1^4, e_1^5, e_3^6, criando uma relação de dependência entre eles. Em adição, os conceitos de estado local e estado global também podem ser observados. O estado do sistema após a execução dos eventos representados pela tupla $\{e_1^2, e_2^1, e_3^2\}$ é um dos estados globais da computação que o diagrama representa.

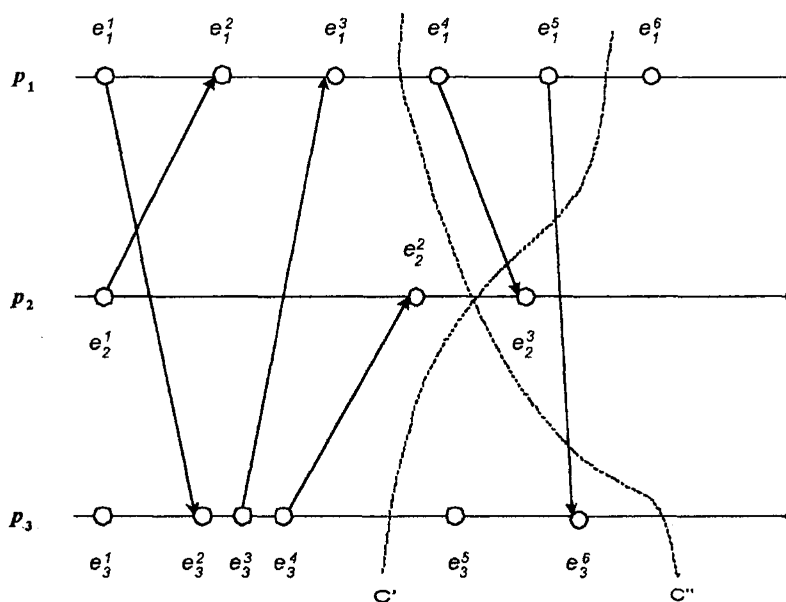

Figura 3.1: Diagrama de espaço×tempo com dois cortes em uma computação distribuída (BABAOGLU; MARZULLO, 1993)

3.2 Cortes Globais Consistentes

A definição de corte global consistente é importante para se identificar quando um estado global obtido do sistema é também consistente, uma vez que a observação desse sistema pode ser obsoleta ou mesmo inválida para um observador externo.

Definição 3.2.1 (Corte) *Um corte é um subconjunto C dos eventos executados pela aplicação distribuída.*

O conjunto contendo o último evento de cada processo contido no corte é denominado "fronteira do corte". A figura 3.1 contém dois cortes C' e C'' correspondendo as tuplas $\{e_1^5, e_2^2, e_3^4\}$ e $\{e_1^3, e_2^2, e_3^6\}$, respectivamente.

Definição 3.2.2 (Corte Consistente) *Um corte* $\boldsymbol{C}$ *é consistente se para todo evento* $\boldsymbol{e}$ *e* $\boldsymbol{e'}$*:*

$$(e \in C) \land (e' \to e) \Rightarrow e' \in C$$

Um corte consistente é fechado à esquerda sobre a relação de precedência causal. Toda a seta que intercepta o corte tem sua origem à esquerda. Dessa forma, na figura 3.1, o corte C' é consistente e o corte C'' não é consistente, uma vez que o evento e_3^6 corresponde ao recebimento de uma mensagem cujo respectivo comando de envio (evento e_1^5) não faz parte do corte.

A partir da definição de corte consistente, obtém-se o conceito de "Estado Global Consistente", que corresponde a um corte global consistente. Na figura 3.1, o corte C' é um corte global consistente e, por conseqüência, o estado global da aplicação é consistente para um observador externo.

3.3 Relógios Lógicos

Os relógios dos computadores são dispositivos físicos que geram interrupções com freqüência contínua. A saída dessas interrupções pode ser lida por um *software* que traduz o número de interrupções para um valor do tempo real. Este número pode ser usado para marcar qualquer evento dos processos que estão executando no computador. As aplicações que estão interessadas somente na ordem dos eventos e não no tempo real em que eles ocorrem, utilizam apenas o valor do contador, como é o caso da simulação distribuída.

Um relógio lógico mede o tempo discreto, ou seja, um contador acumula o número de eventos ocorridos entre um evento de referência e um outro evento e a sincronização dos relógios lógicos ocorre seguindo a relação de precedência causal entre os eventos (definição 3.1.3) .

Definição 3.3.1 (Relógio Lógico) *É uma função* $\boldsymbol{C}$ *que mapeia um evento* $\boldsymbol{e}$ *em um sistema distribuído para um domínio de tempo* $\boldsymbol{T}$*, denotado como* $\boldsymbol{C(e)}$ *e chamado de timestamp de* $\boldsymbol{e}$*, sendo definida como*

$$C : H \to T$$

tal que a seguinte propriedade seja satisfeita:

$$e_1 \to e_2 \Rightarrow C(e_1) < C(e_2)$$

Essa propriedade é chamada de condição para consistência do relógio. O sistema é considerado fortemente consistente quando T e C satisfazem a condição:

$$e_1 \to e_2 \Leftrightarrow C(e_1) < C(e_2)$$

Para implementar um sistema utilizando relógios lógicos é necessário manter uma estrutura de dados local para todos os processos. Esta estrutura tem o objetivo de representar o tempo lógico do processo. Além disso, é necessário um protocolo, ou seja, um conjunto de regras para atualizar a estrutura de dados sem permitir que o relógio lógico perca a sua condição de consistência.

Existem, basicamente, duas regras que o protocolo deve implementar:

R1 Define como o relógio lógico local deve ser atualizado por um processo quando ele executa um evento (envio ou recebimento de uma mensagem e evento interno).

R2 Define como um processo atualiza o seu relógio global para manter sua visão de tempo global e progresso global, especificando qual informação sobre o tempo lógico deve ser anexada em uma mensagem e como esta informação é usada pelo processo que recebe a mensagem para atualizar sua visão do tempo global.

Sistemas que utilizam relógios lógicos podem diferir na forma de representar o tempo lógico e também no protocolo para atualizar os seus relógios. Entretanto, todos os sistemas de relógios lógicos implementam as duas regras e, conseqüentemente, garantem a propriedade fundamental de consistência de relógio associada à causalidade (RAYNAL; SINGHAL, 1995).

O tempo discreto proposto por Lamport (1978) apresenta um algoritmo simples, com base nas duas regras citadas, para permitir que processadores mantenham um relógio lógico. Cada processo p_i mantém uma variável inteira C_i para armazenar o seu tempo lógico local. As regras para atualizar os relógios são:

R1 Antes de tratar um evento, o processo p_i executa a instrução a seguir, onde d é a

evolução escalar do tempo:

$$C_i = C_i + d \qquad (d > 0)$$

R2 Cada mensagem enviada pelo processo p_i para o processo p_j é rotulada com o valor do relógio local do processo emissor. Quando p_j recebe a mensagem com *timestamp* C_{msg}, ele executa as seguintes ações:

1. $C_j = max(C_j, C_{msg})$.
2. Aplicar regra 1.
3. Liberar a mensagem.

O diagrama de espaço×tempo da figura 3.2 apresenta um exemplo da execução de três processos ordenados pelas das regras de um relógio lógico com evolução escalar igual a 1 ($d = 1$). Se o valor de d é sempre igual a 1, o relógio lógico apresenta uma interessante propriedade: se um evento e possui um *timestamp* h, então $h - 1$ representa a duração lógica mínima, contada em unidades de eventos, necessária para que o evento e ocorra, ou seja, pelo menos $h - 1$ eventos foram produzidos, seqüencialmente, antes do evento e, independente dos processos que os produziram.

Sistemas que utilizam relógios lógicos através de variáveis inteiras não são fortemente consistentes, isto é, para quaisquer dois eventos e_1 e e_2,

$$C(e_1) < C(e_2) \not\Rightarrow e_1 \rightarrow e_2.$$

Como exemplo, pode-se observar na figura 3.2 que os eventos a e b não possuem relação de dependência entre si, mas a relação $C(a) < C(b)$ é verdadeira.

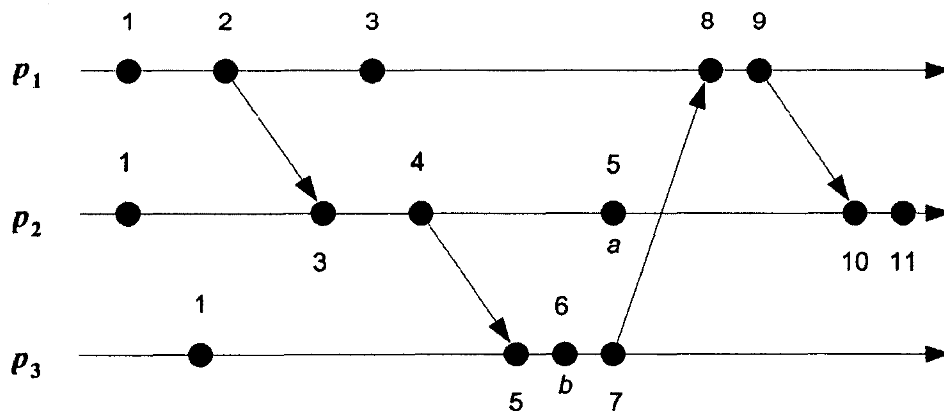

Figura 3.2: Evolução do tempo escalar

3.4 Relógios Vetoriais

Os relógios vetoriais foram desenvolvidos, independentemente, por Fidge (1991), Mattern (1989) e Schmuck (1988). Neste tipo de relógio, o tempo é representado por um vetor de inteiros não-negativos de n-dimensões. Cada processo p_i mantém um vetor $vl_i[1..n]$, onde $vl_i[i]$ é o relógio lógico local de p_i e descreve a evolução do tempo lógico no processo p_i. No estado inicial de cada processo, os respectivos vetores possuem todas as posições valendo zero. $vl_i[j]$ representa a última informação conhecida pelo processo p_i do tempo local do processo p_j, ou seja, é o número de eventos de p_j que precede causalmente os eventos de p_i. Se $vl_i[j] = x$, então o processo p_i sabe que o tempo local no processo p_j progrediu até x. $vl_i[i]$ conta o número de eventos que p_i executou até e_i. Como a quantidade de informações armazenadas pelo relógio vetorial é maior do que aquelas armazenadas pelo relógio lógico é possível detectar uma forte condição de relógio em sua estrutura.

Há, igualmente, duas regras de atualização para os processos que utilizam relógios vetoriais:

R1 Antes de executar um evento, o processo p_i atualiza o seu relógio local através da instrução:

$$vl_i[i] = vl_i[i] + d \qquad (d > 0)$$

R2 Em cada mensagem m enviada pelo processo p_i para o processo p_j, o relógio vetorial é anexado. Quando p_j recebe a mensagem, as seguintes ações são empreendidas:

1. Atualizar o relógio local com a seguinte instrução:

$$1 \leq k \leq n : vl_j[k] = max(vl_i[k], vl_j[k])$$

2. Aplicar a regra 1.
3. Liberar a mensagem.

Em sistemas que utilizam relógios vetoriais, o *timestamp* associado a um evento é o valor do relógio vetorial do seu processo quando o evento é executado.

Segundo Babaoglu e Marzullo (1993), os relógios vetoriais apresentam propriedades importantes para a ordenação de eventos em uma aplicação distribuída. A primeira delas é a forte condição de relógio. Dado dois vetores de números naturais n-dimensionais V e V', define-se a relação "menor que" ($<$) entre eles da seguinte forma:

$$V < V' \Leftrightarrow (V \neq V') \wedge (\forall k : 1 \leq k \leq n : V[k] \leq V'[k]).$$

Na figura 3.2 foi observado, através dos eventos a e b, que não era possível identificar uma forte condição de relógio. No entanto, se a mesma aplicação fosse implementada utilizando relógios vetoriais, os valores correspondentes aos eventos a e b seriam, respectivamente, $[2,4,0]$ e $[2,3,3]$ (figura 3.3). Como $a \not\rightarrow b$, então, $vl(a)$ não pode ser menor que $vl(b)$ e vice-versa.

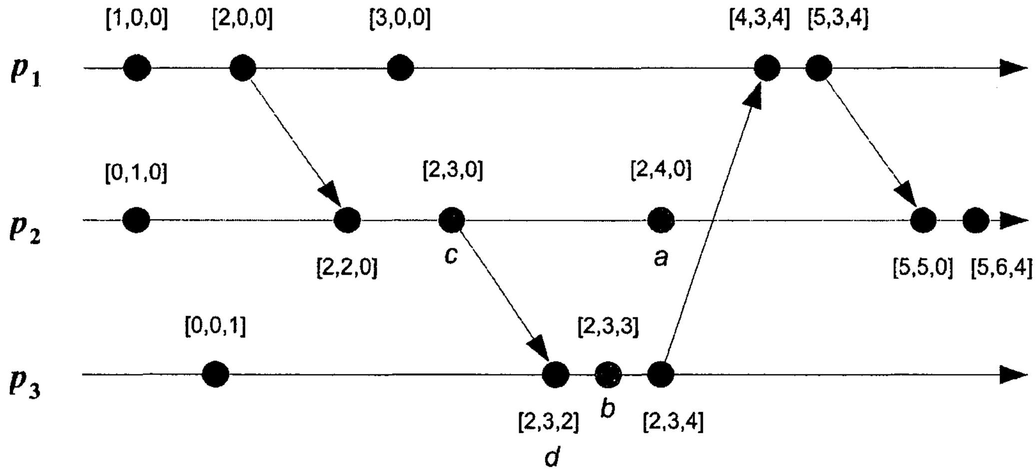

Figura 3.3: Evolução dos mesmos processos da figura 3.2 utilizando um relógio vetorial

Por construção, se um evento e_i do processo p_i antecede o evento e_j do processo p_j, sendo $i \neq j$, então $vl_i[i] \leq vl_j[i]$. A condição $vl_i[i] = vl_j[i]$ é possível e representa a situação onde e_i é o último evento de p_i, que precede causalmente e_j de p_j (e_i deve ser um evento de envio de mensagem). É possível identificar esta situação na figura 3.3, através dos eventos c e d, pois $vl_c[2] = vl_d[2] = 3$. Neste exemplo, destaca-se o teste na posição dois do vetor, correspondendo ao índice do processo que enviou a mensagem.

Uma vez que os vetores proporcionam um mecanismo com forte condição de relógio, de maneira direta, obtém-se um meio de verificar se dois eventos são concorrentes. Dado o evento e_i do processo p_i e o evento e_j do processo p_j, então,

$$e_i \| e_j \Leftrightarrow (vl_i[i] > vl_j[i]) \wedge (vl_j[j] > vl_i[j]).$$

Outra característica interessante dos relógios vetoriais é a possibilidade de verificar se dois eventos podem pertencer a mesma fronteira de um corte consistente. Dois eventos, e_i do processo p_i e e_j do processo p_j com $i \neq j$, não poderão fazer parte do mesmo corte consistente, se e somente se,

$$(vl_i[i] < vl_j[i]) \vee (vl_j[j] < vl_i[j]).$$

Os termos da disjunção caracterizam as duas possibilidades para o corte incluir um

evento de recebimento de mensagem (*receive*) sem o respectivo evento de envio (*send*). Dessa forma, um corte é consistente se, em sua fronteira, a condição acima for falsa.

Finalmente, é possível ainda identificar o número de eventos que antecede causalmente um determinado evento. Dados o evento e_i do processo p_i e seu relógio vetorial $vl_i(e_i)$, o número de eventos e, tal que $e \rightarrow e_i$, é dado por:

$$\sharp(e_i) = (\sum_{j=1}^{n} vl_i(e_i)[j]) - 1.$$

Na figura 3.3, pode-se verificar que o número de eventos que antecede o evento b é 7, pois:

$$\sharp(b) = (\sum_{j=1}^{n} vl_b[j]) - 1 = (vl_b[1] + vl_b[2] + vl_b[3]) - 1 = (2 + 3 + 3) - 1 = 7.$$

Usualmente, dependendo da aplicação distribuída, somente um subconjunto de eventos produzidos é necessário para a análise de suas dependências. Assim, não é preciso detectar a relação causal entre todos os eventos, mas apenas em um subconjunto (ANCEAUME; HéLARY; RAYNAL, 2002). Um exemplo é a localização das dependências entre *checkpoints* com objetivo de identificar se o conjunto é consistente. Esta análise será realizada nas próximas seções.

3.5 *Checkpoints* Globais Consistentes

Checkpoints são estados de um processo armazenados em um mecanismo "persistente" de memória para a recuperação de um sistema em caso de falhas. Um *checkpoint* local representa uma fotografia de um processo e um *checkpoint* global é o conjunto de *checkpoints* locais, sendo um para cada processo envolvido na computação distribuída.

Em um sistema distribuído, onde não existe um relógio global de referência, é importante garantir que um *checkpoint* global seja consistente para, realmente, poder ser utilizado como ponto de partida em caso de falha do sistema. O *checkpoint* global será considerado consistente se nenhum *checkpoint* do conjunto possuir relação de dependência com qualquer outro *checkpoint* do mesmo conjunto.

Manivannan e Singhal (1999) definem a relação de precedência causal utilizando os estados dos processos e não seus eventos, ou seja, um estado σ_q de um processo p_q é causalmente dependente do estado σ_p do processo p_p se uma mensagem (ou seqüência de

mensagens) enviada pelo processo p_p, após o estado σ_p, foi recebida por p_q antes de alcançar o estado σ_q. Esta visão facilita a compreensão das condições que tornam consistente um *checkpoint* global.

Definição 3.5.1 (***Checkpoint* Global Consistente**) *Considerando que o conjunto de inteiros* $\{c'_0, c'_1, \ldots, c'_{n-1}\}$ *representa os índices de cada checkpoint de um conjunto de checkpoints* $S = \{C_0^{c'_0}, C_1^{c'_1}, \ldots, C_{n-1}^{c'_{n-1}}\}$*, sendo um por cada processo, o conjunto* S *será consistente se, e somente se,*

$$\forall i, j : 0 \leq i, j < n : C_i^{c'_i} \| C_j^{c'_j}$$

Para um *checkpoint* global ser consistente é necessário que ele esteja associado a um corte consistente da computação e represente apenas eventos internos, ou seja, o *checkpoint* não deve ser realizado imediatamente após um evento de envio ou recebimento de uma mensagem.

A dependência entre estados dificulta a restauração de um estado seguro do sistema em caso de falha no mesmo. Isso ocorre porque durante a falha de um processo essas dependências podem forçar outros processos que não falharam a retroceder, criando o que é comumente denominado de propagação de *rollbacks* (ELNOZAHY et al., 2002).

Por exemplo, considerando a situação onde o emissor de uma mensagem m retorna para um estado que precede o envio de m. O receptor de m deve também retornar para um estado que precede a recepção de m; caso contrário, os estados dos dois processos serão inconsistentes porque eles estarão sinalizando o recebimento de uma mensagem que não foi enviada, o que é impossível em qualquer operação livre de falhas. Sob certas condições, a propagação dos *rollbacks* pode estender o retorno dos processos para o estado inicial da computação, perdendo todo o trabalho executado antes da falha (ELNOZAHY et al., 2002). Esta situação é conhecida como efeito dominó (RANDELL, 1975), conforme ilustra a figura 3.4.

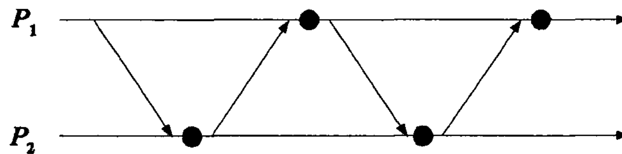

Figura 3.4: *Checkpoints* inúteis (efeito dominó)

O efeito dominó pode ocorrer se cada processo obtiver seus *checkpoints* de maneira independente, constituindo uma abordagem conhecida como *checkpointing* assíncrono ou não-coordenado. Como se deseja evitar a ocorrência do efeito dominó, outra técnica pode ser utilizada: *checkpointing* síncrono ou coordenado (CHANDY; LAMPORT, 1985). No

esquema síncrono, os processos coordenam suas atividades de *checkpointing*, mantendo os *checkpoints* do sistema sempre consistentes. O espaço de armazenamento é geralmente menor, uma vez que os processos necessitam de apenas dois *checkpoints* armazenados. A desvantagem deste esquema é que todos os processos devem suspender o seu processamento sincronamente para o armazenamento dos *checkpoints*, degradando o desempenho e aumentando o custo computacional para coordenar esta atividade.

Uma outra abordagem denominada quase-síncrona, semi-síncrona ou induzida por comunicação, está baseada em um protocolo obedecido pelos processos da aplicação para a seleção de *checkpoints*, a partir das informações registradas nas mensagens que trafegam no sistema (RUSSELL, 1980; GUPTA; LIU; LIANG, 2004). Esta abordagem será melhor detalhada na seção 3.8.

3.6 Condições para Consistência dos *Checkpoints*

Em aplicações que utilizam *checkpoints*, para a recuperação do sistema em caso de falhas, é comum a situação em que o sistema necessita encontrar um *checkpoint* global consistente a partir de um subconjunto de *checkpoints* ou de um único *checkpoint* local. Entretanto, para se atingir este objetivo não basta localizar *checkpoints* que não possuam relação de dependência entre si. Netzer e Xu (1995) determinaram as condições necessárias e suficientes para que um conjunto de *checkpoints* possa fazer parte de um mesmo *checkpoint* global consistente. Eles introduziram a noção de caminho zigzag (*zigzag path*) ou caminho-Z (*Z-path*), que é uma generalização da relação de precedência causal de Lamport (1978) (definição 3.1.3). Um caminho-Z entre dois *checkpoints* é semelhante a um caminho causal, mas um caminho-Z permite que uma mensagem seja enviada antes que uma outra seja recebida no mesmo caminho.

Definição 3.6.1 (Caminho-Z (*Z-path*)) *Caminho-Z é uma seqüência de mensagens* $m_1, m_2, \ldots, m_p$ *($p \geq 1$) entre dois checkpoints* C_i^k *e* C_j^l *tal que:*

1. m_1 *é enviada por* p_i *após* C_i^k*;*
2. *se* m_a *($1 \leq a < p$) é recebida por* p_r*, então* m_{a+1} *é enviada por* p_r *neste mesmo intervalo de checkpoints ou em algum intervalo posterior a este; e*
3. m_p *é recebida por* p_j *antes de* C_j^l*.*

A existência de um caminho-Z entre *checkpoints* dificulta a localização dos *checkpoints* globais consistentes, uma vez que *checkpoints* que não possuam relação de causalidade entre si nem sempre poderão fazer parte de um mesmo *checkpoint* global consistente, como

pode ser verificado na figura 3.5. Nesta figura, os *checkpoints* C_1^1 e C_3^1 são concorrentes entre si, mas não existe *checkpoint* no processo P_2 que possa ser unido a estes *checkpoints* para formar um conjunto consistente.

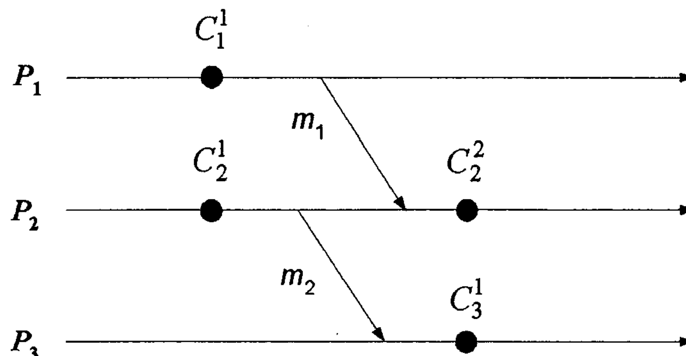

Figura 3.5: Exemplo de um sistema com caminho-Z

Além da seqüência de mensagens necessárias para formar um caminho-Z, Manivannan, Netzer e Singhal (1997) estenderam essa definição para incluir *checkpoints* de um mesmo processo. Assim, entre dois *checkpoints* de um mesmo processo sempre haverá um caminho-Z.

Outro conceito importante é o de ciclo-Z, conforme a definição 3.6.2.

Definição 3.6.2 (Ciclo-Z (*Z-cycles*)) *Um ciclo-Z é a situação em que um caminho-Z liga um checkpoint a ele mesmo.*

Segundo Netzer e Xu (1995), um ciclo-Z corresponde a um *checkpoint* inútil, pois não pode fazer parte de nenhum *checkpoint* global consistente. Por conseguinte, vários trabalhos tem focalizado o problema de garantir que cada *checkpoint* possa pertencer a pelo menos um *checkpoint* global consistente e, desta forma, ser útil (HéLARY et al., 1997; BALDONI; QUAGLIA; CICIANI, 1998). A figura 3.6 apresenta um ciclo-Z envolvendo o *checkpoint* C_A^α.

3.7 Algoritmos para *Checkpoints* Síncronos

Checkpoints síncronos exigem que os processos orquestrem seus *checkpoints* em conjunto para formar um *checkpoint* global consistente. A coordenação dos *checkpoints* simplifica o mecanismo de *rollback* e não está sujeita ao efeito dominó, uma vez que o sincronismo impede a formação de caminhos-Z que relacionem *checkpoints* do mesmo conjunto.

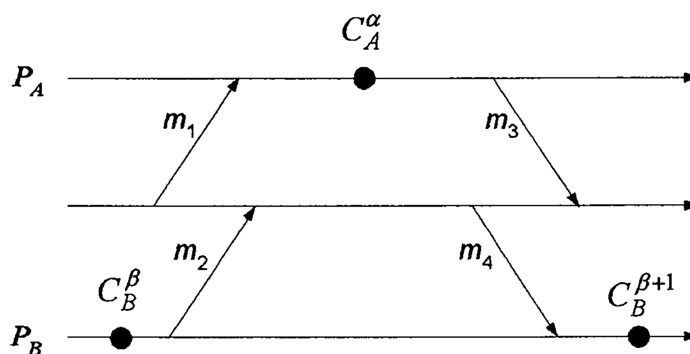

Figura 3.6: Exemplo de um sistema com um ciclo-Z

3.7.1 Algoritmo *Sync-and-Stop*

Uma abordagem direta para coordenar os *checkpoints* consiste em bloquear as comunicações enquanto o protocolo de *checkpointing* está sendo executado, sendo que esse procedimento é semelhante ao mecanismo síncrono para o cálculo do GVT na simulação distribuída, conforme descrito no capítulo anterior. Um processo é designado para ser o coordenador do procedimento de *checkpointing*. É este processo que inicia a obtenção dos *checkpoints* globais consistentes pelo sistema. Primeiramente, ele obtém um *checkpoint* e envia uma mensagem para todos os outros processos pedindo para que eles realizem seus *checkpoints*. Quando um processo recebe essa mensagem, ele interrompe sua execução, espera pela liberação de todos os canais de comunicação, realizando, em seguida, um *checkpoint*. Em adição, ele envia uma mensagem de volta ao coordenador confirmando a tarefa. Após o coordenador receber as respostas de todos os processos, ele retorna com uma mensagem de confirmação que completa esse protocolo de duas-fases (ELNOZAHY et al., 2002). Esse algoritmo também é conhecido como *Sync-and-Stop* (SNS) (PLANK, 1993) e está apresentado na listagem A.1 do apêndice A.

A forma como os processos liberam os canais de comunicação pode variar dependendo, inclusive, da estrutura física da rede de comunicação entre os processadores. A forma mais simples de realizar esta tarefa é através de mensagens de confirmação.

A figura 3.7 mostra o diagrama de espaço×tempo do funcionamento do algoritmo SNS. Nesta representação, o processo P_2 envia uma mensagem m para P_1, e aguarda de p_1 a confirmação do recebimento de m antes de enviar sua mensagem `confirmaSNS` para o processo coordenador (P_c). É possível verificar como o algoritmo trabalha para definir C, o corte correspondente aos *checkpoints* locais. Como nenhuma mensagem atravessa o corte, ele é consistente, e, desta forma, válido. De fato, todos os cortes produzidos pelo algoritmo SNS são consistentes, pois nunca haverá uma mensagem atravessando o corte (PLANK, 1993).

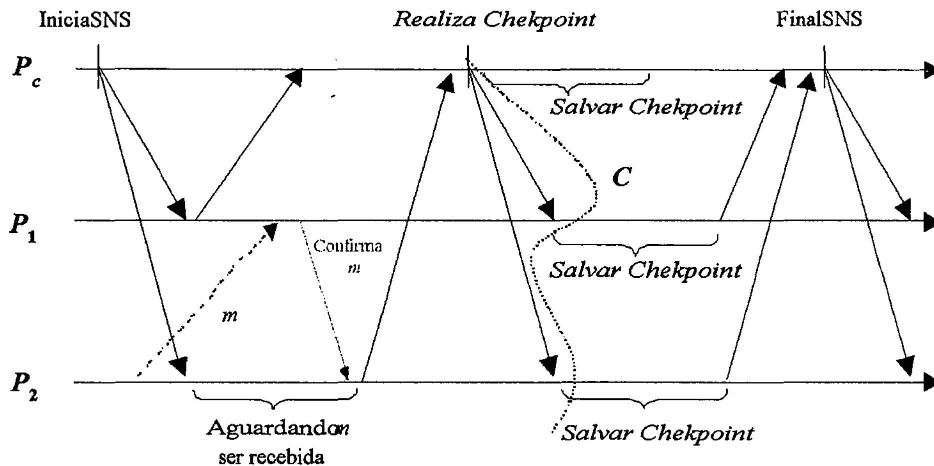

Figura 3.7: Funcionamento do algoritmo SNS (PLANK, 1993)

O algoritmo SNS é relativamente simples, entretanto, impõe um alto custo ao sistema. Uma alternativa é a utilização do algoritmo Chandy-Lamport (CHANDY; LAMPORT, 1985), que será descrito na próxima seção.

3.7.2 Algoritmo Chandy-Lamport

O algoritmo Chandy-Lamport (CL) resolve o maior problema do algoritmo SNS, ou seja, o congelamento de todos os processos para realizar um *checkpoint* global consistente. Durante a geração de um corte no algoritmo CL são permitidas mensagens atravessando o corte, que são registradas como parte do *checkpoint* global.

No algoritmo CL os processos se comunicam através de canais estáticos, ou seja, um processo P_i somente poderá enviar uma mensagem diretamente para um processo P_j se existir um canal que conecta P_i até P_j. Caso contrário, as mensagens devem percorrer caminhos alternativos passando por outros processos. Além disso, os canais entregam as mensagens em ordem FIFO (*First In First Out*), semelhantemente ao que ocorre com os protocolos conservativos na simulação distribuída.

A figura 3.8 ilustra a idéia dos canais utilizada neste algoritmo. O processo P_c só pode enviar mensagens diretamente para P_1. Todas as mensagens dirigidas para P_2 devem passar através de P_1.

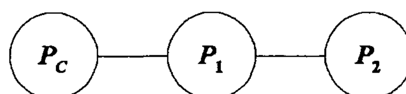

Figura 3.8: Grafo representando os canais de comunicação entre os processos p_c, p_1 e p_2

Para realizar um *checkpoint*, o processo coordenador do algoritmo CL envia uma mensagem `IniciaCL` para todos os processos que possuem canais de comunicação com ele. Imediatamente após o envio da mensagem, o processo coordenador realiza um *checkpoint*. Todos os processos da computação executam as seguintes atividades:

- Se o processo p recebe `IniciaCL` e ainda não realizou o seu *checkpoint* local, então ele envia a mensagem `IniciaCL` para todos os processos que possuem canais de comunicação com ele, e realiza o seu *checkpoint*.

- Após essa etapa, se p recebe uma mensagem m por um canal c, e ainda não recebeu `IniciaCL` pelo mesmo canal (c), então m é uma mensagem que está cruzando o corte e deve ser armazenada.

- Quando todos os processos receberem `IniciaCL` através dos canais de entrada, então não existirá mais mensagens atravessando o corte e o *checkpoint* local estará finalizado. Cada processo notifica o processo coordenador pelo envio de uma mensagem `FinalProcessoCL`.

- Quando o processo coordenador recebe todas as mensagens `FinalProcessoCL`, ele retorna uma confirmação para todos os processos com a mensagem `FinalCL` (novamente, se p não está diretamente conectado com o processo coordenador, então a mensagem é enviada por nós intermediários).

- Quando p recebe a última mensagem do coordenador, o procedimento de *checkpointing* está encerrado.

O detalhamento deste algoritmo está inserido no apêndice A (listagem A.2) e a figura 3.9 apresenta um exemplo do seu funcionamento. É importante observar que, na figura, m é uma mensagem que atravessa o corte e, portanto, deve ser armazenada. O Algoritmo Chandy-Lamport é correto, pois, define um corte consistente, e obtém um *checkpoint* global consistente referente a esse corte (CHANDY; LAMPORT, 1985).

Os esquemas de *checkpoiting* coordenados resultam em grande custo computacional e, desta forma, esquemas não bloqueantes são preferíveis (ELNOZAHY; JOHNSON; ZWAENEPOEL, 1992).

3.8 Algoritmos para *Checkpoints* Semi-Síncronos

Os algoritmos que exigem que os processos obtenham *checkpoints* induzidos pela comunicação são chamados algoritmos de *checkpointing* quase-síncronos ou semi-síncronos, pois, algumas atividades de *checkpointing* são disparadas pelo padrão de mensagens e através

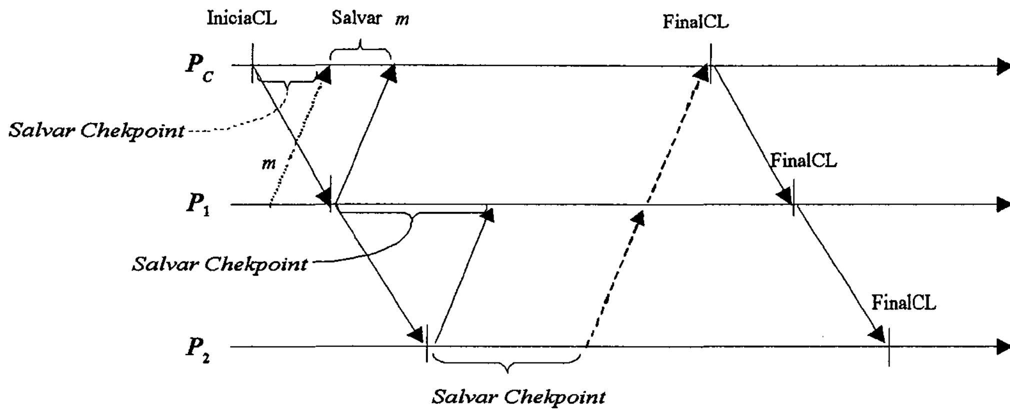

Figura 3.9: Funcionamento do algoritmo CL

das dependências que existem entre os *checkpoints* dos processos (MANIVANNAN; SINGHAL, 1999). Prioritariamente, os processos selecionam *checkpoints* livremente, chamados *checkpoints* básicos e, eventualmente, podem ser induzidos a selecionar *checkpoints* adicionais, denominados *checkpoints* forçados, segundo predicados avaliados sobre informações de controle, que são propagadas através das mensagens da aplicação (GARCIA, 2001).

Os algoritmos semi-síncronos para *checkpoints* são classificados em quatro classes denominadas: *Strictly Z-Path Free* (SZPF), *Z-Path Free* (ZPF), *Z-Cycle Free* (ZCF) e *Partially Z-Cycle Free* (PZCF). Esta classificação está baseada no grau de prevenção dos caminhos-Z e respeitam a relação de continência ilustrada na figura 3.10.

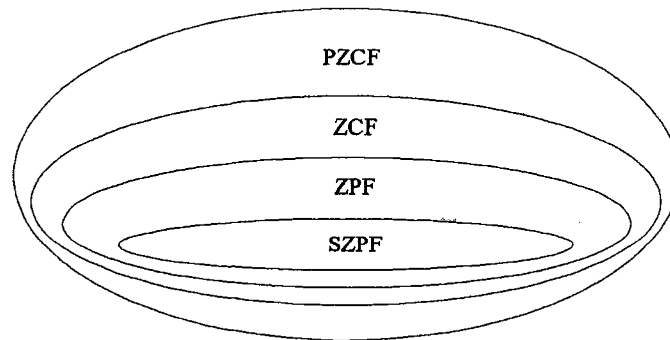

Figura 3.10: Relação entre os vários modelos de algoritmos de *checkpointing* semi-síncronos (MANIVANNAN; SINGHAL, 1999)

3.8.1 *Strictly Z-path Free Checkpointing*

O padrão SZPF elimina totalmente os caminhos-Z que não possuam relação de causalidade (caminhos-Z não causais) entre os *checkpoints*, sendo o mais completo entre todas

as classes. Neste padrão, se existir caminho-Z, os *checkpoints* que o compõem devem possuir relação de causalidade entre si.

Através dos padrões de mensagens, é garantido que todos os eventos de recepção de mensagens precedem aos eventos de envio de mensagens em um mesmo intervalo de *checkpoints*. Com esse padrão, todos os *checkpoints* realizados pelos processos são úteis e podem ser rastreados durante a execução dos mesmos.

Definição 3.8.1 (Padrão SZPF) *Um padrão de checkpoints é dito ser Strictly Z-path Free (SZPF) se não existe nenhum caminho-Z não-causal entre quaisquer dois checkpoints, não necessariamente distintos.*

São exemplos de algoritmos do padrão SZPF: *Checkpoint-After-Send-Before-Receive* (CASBR), *Checkpoint-After-Send* (CAS), *Checkpoint-Before-Receive* (CBR), *No-Receive-After-Send* (NRAS) (WANG, 1997). O padrão de *checkpoints* destes algoritmos é ilustrado na figura 3.11.

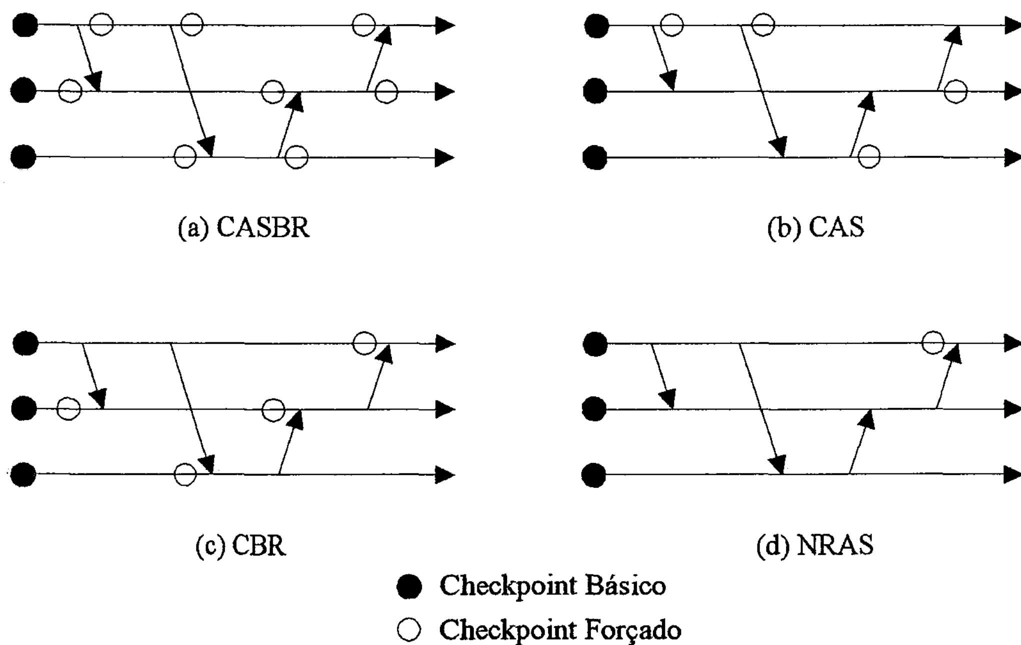

Figura 3.11: Padrão de *checkpoints* dos algoritmos do protocolo SZPF

Dentre os exemplos citados, o algoritmo NRAS é o que produz a menor sobrecarga no sistema, ou seja, induz o menor número de *checkpoints* forçados, entretanto, o método CAS possui uma interessante e útil propriedade: o conjunto consistindo de todos os últimos *checkpoints* de cada processo forma um *checkpoint* global consistente (MANIVANNAN; SINGHAL, 1999).

Instituto de Ciências Matemáticas e de Computação
Biblioteca

3.8.2 *Z-Path Free Checkpointing*

No padrão SZPF, a ausência de caminhos-Z não causais entre *checkpoints* faz com que todos eles sejam utilizáveis e também facilita a construção de *checkpoints* globais consistentes de maneira incremental. No entanto, pode-se ter essas características sem, necessariamente, eliminar todos os caminhos-Z não causais. Os benefícios podem ser alcançados pela eliminação somente dos caminhos-Z não causais correspondentes àqueles que não possuam caminhos duplicados causalmente (*sibling causal path*). Esse relaxamento define o padrão ZPF (*Z-path Free*) (MANIVANNAN; SINGHAL, 1999).

Definição 3.8.2 (Sibling Causal Path) *Se existe um caminho-Z de A para B, e também um caminho causal de A para B, o caminho causal é um caminho causalmente duplicado ("sibling" do caminho-Z).*

Definição 3.8.3 (Padrão ZPF) *Um padrão de checkpoints é dito ser Z-path Free (ZPF) se, e somente se, para quaisquer dois checkpoints A e B, existindo um caminho-Z de A para B, também existir um caminho causal de A para B.*

Em um sistema ZPF, apesar de caminhos-Z não causais poderem existir entre *checkpoints*, eles sempre possuem um caminho duplicado causalmente e, desta forma, tais caminhos-Z não causais podem ser rastreados em tempo de execução através dos *sibling causal path* correspondentes. A figura 3.12 apresenta um diagrama de espaço×tempo representando um padrão de *checkpoints* que é ZPF mas não é SZPF. Nesta figura, os *checkpoints* forçados são obtidos para prevenir caminhos-Z não causais que não possuem *sibiling causal path* correspondentes.

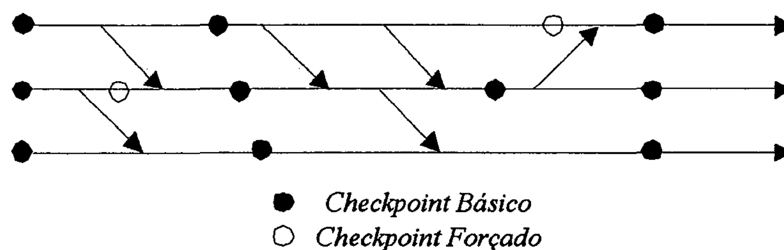

Figura 3.12: Padrão de *checkpoints* do protocolo ZPF

Uma propriedade importante dos algoritmos que obedecem aos padrões SZPF e ZPF é que todas as dependências entre *checkpoints* podem ser rastreadas em tempo de execução através da utilização de vetores de dependências ou relógios vetoriais (WANG, 1997). Essa propriedade é conhecida como *Rollback-Dependency Trackability* (RDT), sendo importante para o protocolo apresentado nesta tese, pois em padrões RDT, um

conjunto de *checkpoints* concorrentes S sempre pode fazer parte de um mesmo *checkpoint* global consistente.

Definição 3.8.4 (Propriedade RDT) *Um padrão de checkpoints é dito satisfazer a propriedade rollback-dependency trackability quando, para quaisquer dois* checkpoints C_p^i *e* C_q^j*, se existir um caminho-Z de* C_p^i *para* C_q^j *então o vetor de dependências de* C_q^j *na posição* p *deve ser maior ou igual a* $i+1$.

Dentre os algoritmos que implementam esse padrão, podem ser citados os métodos *Fixed-Dependency-Interval* (FDI) e *Fixed-Dependency-After-Send* (FDAS). O algoritmo FDI utiliza um relógio vetorial (RV) (RAYNAL; SINGHAL, 1995) para acompanhar a precedência causal entre os *checkpoints*. Este relógio registra as dependências entre os *checkpoints*. Ao enviar uma mensagem, o processo emissor anexa o seu vetor de dependências. Quando um processo p_q recebe uma mensagem m, p_q processa a mensagem se $m.RV[r] \leq m.RV_q[r] \forall r$; caso contrário, ele primeiro realiza um *checkpoint* forçado, atualiza o seu relógio vetorial RV_q e, então, processa a mensagem. Este procedimento permite que um processo envie e receba mensagens em um intervalo de *checkpoints* desde que ele não tenha alterado o seu vetor de dependências no mesmo intervalo. É interessante observar que o algoritmo FDI apenas avalia a dependência causal entre os *checkpoints*, ignorando as relações entre as mensagens do sistema.

O algoritmo FDAS é uma variação do algoritmo FDI. Ele acrescenta a observação utilizada no algoritmo NRAS (*No Receive After Send*) de que um caminho-Z não causal só se forma após algum envio de mensagem. Desta forma, este algoritmo quebra quaisquer caminhos-Z, exceto aqueles duplicados causalmente no intervalo corrente (VIEIRA, 2001). O padrão ZPF apresentado na figura 3.12 pode ser obtido através do algoritmo FDAS.

Os algoritmos FDI e FDAS são apresentados, respectivamente, nas listagens A.3 e A.4 do apêndice A.

3.8.3 *Z-Cycle Free Checkpointing*

Se o objetivo de um algoritmo semi-síncrono for criar *checkpoints* úteis, então é possível realizar um relaxamento nos protocolos SZPF e ZPF. Esse relaxamento permite que o padrão de *checkpoints* contenha caminhos-Z não causais mas não permite que o padrão contenha ciclos-Z. Conseqüentemente, garante-se que todos os *checkpoints* serão úteis, ou seja, podem fazer parte de, pelo menos, um *checkpoint* global consistente Manivannan e Singhal (1999).

Definição 3.8.5 (Padrão ZCF) *Um padrão de checkpoints é dito ser Z-cycle Free (ZCF) se, e somente se, nenhum dos checkpoints estiver envolvido em um ciclo-Z.*

Uma peculiaridade relevante do padrão ZCF é que ele produz uma sobrecarga no sistema menor que os padrões SZPF e ZPF, uma vez que os algoritmos que implementam este padrão são mais simples e propagam menos informações de controle. O seu maior problema é a dificuldade que existe em construir *checkpoints* globais consistentes de forma incremental devido à presença de caminhos-Z não causais.

O primeiro algoritmo criado nesse padrão é conhecido como BCS (BRIATICO; CIUFFOLETTI; SIMONCINI, 1984). Esse algoritmo usa a relação de precedência causal para determinar quando deve induzir um *checkpoint* forçado. O algoritmo propaga um relógio lógico que sempre é incrementado durante um *checkpoint* básico. Toda vez que um processo recebe uma mensagem contendo uma informação de relógio maior do que o seu relógio atual, um *checkpoint* forçado é realizado. A figura 3.13 mostra o padrão deste algoritmo.

Outro exemplo de algoritmo que implementa o padrão ZCF pode ser encontrado em Manivannan e Singhal (1996). Neste algoritmo, cada processo mantém um contador que é periodicamente incrementado. Quando um processo realiza um *checkpoint*, ele associa o valor corrente do seu contador com o *checkpoint*. Cada mensagem é rotulada com o número de seqüência do *checkpoint* atual. Se o número de seqüência que acompanha uma mensagem é maior que o número de seqüência do *checkpoint* do processo que recebe a mensagem, então este processo obtém um *checkpoint* e associa o número recebido da mensagem com o novo *checkpoint*, para depois processar a mensagem. Uma característica interessante deste algoritmo é a diminuição do número total de *checkpoints* da aplicação, pois quando ocorre um *checkpoint* forçado ele não realiza o próximo *checkpoint* básico.

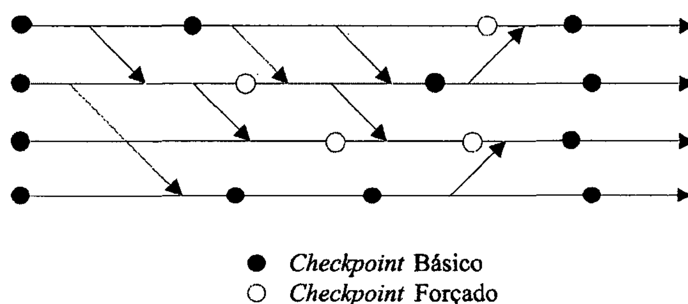

Figura 3.13: Exemplo do padrão de *Checkpoints* do algoritmo BCS

3.8.4 *Partially Z-cycle Free Checkpointing*

Um padrão PZCF pode conter caminhos-Z e, por conseguinte, *checkpoints* inúteis. No entanto, espera-se que seja feito um esforço para que, pelo menos uma parte dos *checkpoints*, sejam úteis. O protocolo proposto por Wang e Fuchs (1993) é uma variação

do protocolo BCS restringindo a indução de *checkpoints* forçados para índices múltiplos de um determinado valor g. Um *checkpoint* cujo índice é múltiplo de g é garantidamente útil, enquanto os outros *checkpoints* podem ser úteis ou não (GARCIA, 2001).

3.9 Visões Progressivas

Uma visão progressiva de uma computação distribuída é uma seqüência de *checkpoints* globais consistentes de maneira que cada *checkpoint* global na seqüência aparece acontecendo um após outro. Por exemplo, os *checkpoints* globais (Σ^0, Σ^1, Σ^2) formam um visão progressiva da computação apresentada na figura 3.14. A restrição imposta nesta seqüência é que, se um *checkpoint* x precede causalmente um *checkpoint* y, um *checkpoint* global consistente que inclua y não pode preceder um *checkpoint* global consistente que inclua x na seqüência em questão (GARCIA; BUZATO, 1999).

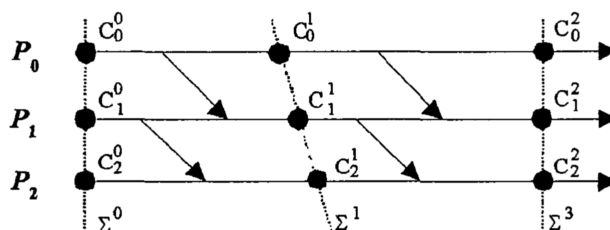

Figura 3.14: Exemplo de Visões Progressivas.

Definição 3.9.1 (Visões Progressivas) *Uma visão progressiva de uma computação distribuída é uma seqüência de checkpoints globais consistentes* $(\Sigma^0, \Sigma^1, \Sigma^2, \ldots, \Sigma^m)$ *tal que*

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

A restrição imposta na seqüência de *checkpoints* globais consistentes induz uma bem definida ordenação para um observador dos *checkpoints*. Um *checkpoint* x deve ser observado antes de um *checkpoint* y, em uma visão progressiva, se y não pode fazer parte de um *checkpoint* global consistente que contenha x ou com um *checkpoint* precedente de x no mesmo processo (GARCIA; BUZATO, 1999).

3.10 Linhas de Recuperação

Qualquer *checkpoint* global consistente pode ser usado para restaurar o sistema após uma falha. Todavia, é desejável minimizar o montante de trabalho perdido pela restau-

ração do sistema ao mais recente *checkpoint* global consistente, que é chamado de linha de recuperação (*recovery line*) (RANDELL, 1975; CHIU; YOUNG, 1996).

Este trabalho estende o conceito de Linhas de Recuperação para o conceito de Conjunto de Linhas de Recuperação. O conceito de linha de recuperação é utilizado em sistemas tolerantes a falhas. Nesse tipo de sistema, a principal característica é o retorno do sistema para o último *checkpoint* global consistente armazenado antes da falha. Dessa forma, não faz sentido um retorno para um *checkpoint* global consistente mais distante. Agbaria et al. (2001) demonstraram que algoritmos que garantem a propriedade RDT minimizam o tamanho do retorno durante o procedimento de *rollback*.

Na abordagem otimista da simulação distribuída, entretanto, o retorno do processamento não se dá por falhas no sistema, mas sim, por erros de causalidade que podem exigir que um processo retorne até um ponto anterior ao último *checkpoint* armazenado, isto é, retornar para *checkpoints* globais consistentes anteriores a linha de recuperação. Em vista disso, é preciso manter as informações dos *checkpoints* globais consistentes, que já foram linhas de recuperação durante toda a computação, ou mais especificamente, *checkpoints* globais consistentes com relógios lógicos iguais ou maiores do que o GVT.

Definição 3.10.1 (Conjunto de Linhas de Recuperação) *Um conjunto de linhas de recuperação é um conjunto de checkpoints globais consistentes que foram observados em uma visão progressiva de uma computação distribuída, particularmente em uma simulação distribuída.*

Este conceito será discutido no próximo capítulo durante a apresentação do protocolo *Rollback* Solidário.

3.11 Considerações Finais

A abordagem síncrona para obtenção de *checkpoints* globais consistentes naturalmente produz uma visão progressiva da computação distribuída, entretanto, esta abordagem se torna inviável nas aplicações que necessitam construir linhas de recuperação com relativa freqüência, devido à necessidade de interromper as atividades dos processos durante o procedimento de *checkpointing*. Neste contexto, a abordagem semi-síncrona surge como uma alternativa viável para a obtenção das linhas de recuperação, sem o risco da aplicação obter somente *checkpoints* que possuam dependências entre si, o que poderia ocasionar o indesejável efeito dominó quando fosse necessário reconstituir o sistema.

O próximo capítulo apresenta o protocolo *Rollback* Solidário e discute a utilização de *checkpoints* globais consistentes no sincronismo dos processos de um programa de simulação distribuída.

Capítulo 4

Rollback Solidário

Este capítulo apresenta um novo protocolo otimista, denominado ***Rollback*** **Solidário**. Inicialmente, o protocolo poderia ser visto como uma variante do *Time Warp*, entretanto, ele se fundamenta na teoria dos *checkpoints* globais consistentes, apresentando diferenças significativas na forma como os processos são sincronizados durante um *rollback*.

As duas principais diferenças entre os protocolos *Rollback* Solidário e *Time Warp* são:

1. O protocolo *Rollback* Solidário não utiliza anti-mensagens e, portanto, os processos lógicos não precisam manter cópia das mensagens trocadas entre si; e
2. Quando uma mensagem *straggler* surge no sistema, o mecanismo proposto identifica, simultaneamente, os estados que devem ser recuperados pelos processos que estão envolvidos no *rollback*.

Essas diferenças:

1. Facilitam o cálculo do GVT,
2. Evitam a ocorrência de *rollbacks* em cascata, permitindo uma utilização mais adequada da memória,
3. Minimizam o trafego de mensagens, uma vez que eliminam o envio de anti-mensagens,
4. Evitam a reincidência de *rollbacks*.

No protocolo *Time Warp*, inicialmente apenas o processo que identificou o erro de causa e efeito realiza *rollback*. Se esse processo enviou mensagens, ele deve desfazer essa atividade com anti-mensagens, podendo provocar novos *rollbacks* em outros processos. Desta forma, uma anti-mensagem pode iniciar *rollbacks* em cascata aumentando a sobrecarga na rede de comunicação e diminuindo o desempenho do programa de simulação.

A proposta principal do protocolo *Rollback* Solidário é retornar todos os processos que estarão envolvidos em um *rollback* para um *checkpoint* global consistente. Deste modo, a decisão do retorno é tomada com base nas informações de todos os processos.

Inicialmente, são apresentadas duas conjecturas sobre o desempenho da simulação e a quantidade de memória necessária para o bom funcionamento do protocolo proposto:

Melhor utilização da memória: A utilização de *checkpoints* globais consistentes deve diminuir o montante de espaço de memória empregado pelos protocolos otimistas tradicionais. Isso pode ser evidenciado pelo fato dos processos não precisarem armazenar as mensagens já enviadas, também conhecidas como mensagens negativas. Em caso de *rollback*, os processos retornam juntos para um *checkpoint* global consistente que contenha o *checkpoint* do processo que provocou o *rollback*, com *timestamp* menor ou igual ao tempo lógico de criação da mensagem *straggler*. Além disso, não haverá salvamento de estados a cada evento e o intervalo entre *checkpoints* pode ser dimensionado com relação ao tamanho de memória disponível para o armazenamento dos estados, apesar desta última possibilidade estar também disponível, de forma similar, no protocolo *Time Warp* através do mecanismo *Sparse State Saving*.

Melhor desempenho durante a simulação: A ausência de *rollbacks* em cascata agiliza a restauração do estado geral do sistema, simplificando o procedimento de *rollback*. O salvamento de estados em intervalos de eventos, permite um desempenho superior, caso não ocorram *rollbacks*. Em adição, esse mecanismo possibilita um maior controle sobre o desenvolvimento da simulação facilitando a implementação de mecanismos que diminuam o otimismo do protocolo, quando necessário.

Estas conjecturas serão discutidas no próximo capítulo. O restante deste capítulo descreve o funcionamento do protocolo *Rollback* Solidário, demonstrando o seu funcionamento em um ambiente que utilize algoritmos síncronos e semi-síncronos para a geração dos *checkpoints* globais consistentes.

4.1 Arquitetura de um Ambiente para Simulação Distribuída

Um ambiente para Simulação Distribuída pode ser dividido em uma arquitetura de camadas que separam os diversos níveis de abstração que envolvem a aplicação. Para este tipo de programa três níveis são suficientes, como pode ser visto na figura 4.1.

O primeiro nível é responsável pela transmissão das mensagens pelos canais de comunicação. Esta camada trata das funções referentes à comunicação entre as estações da rede,

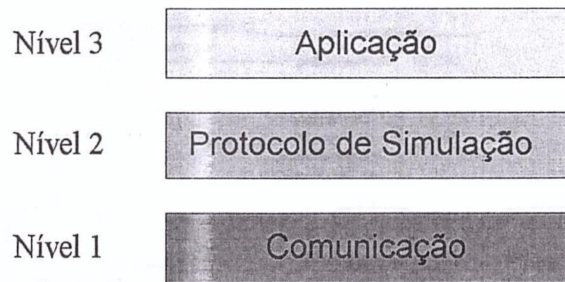

Figura 4.1: Estrutura em camadas de um ambiente de Simulação Distribuída

sendo responsável pelo empacotamento dos dados e por sua conversão para um formato em que todos os elementos compreendam, quando estes forem heterogêneos. Esta camada também proteje a aplicação das falhas recuperáveis, pois implementa as semânticas de tratamento dessas falhas.

No segundo nível se encontra o protocolo de sincronização do programa de simulação, proporcionando recursos de *software* para a implementação de aplicações para simulação distribuída e servindo de interface entre as camadas de Comunicação e Aplicação.

Ferramentas que facilitam a utilização e o desenvolvimento da simulação por parte dos usuários estão inseridas na terceira camada. É neste nível que acontece as interações entre os usuários da simulação distribuída para a criação dos modelos que devem ser simulados. Um exemplo deste tipo de ferramenta pode ser encontrado em Bruschi (2003).

O diagrama de fluxo de dados (DFD) da figura 4.2 apresenta uma visão de alto nível da interação entre estes três níveis. Inicialmente, o usuário interage com o sistema no nível de aplicação definindo os parâmetros do modelo a ser simulado. A forma como isso é realizado depende do ambiente utilizado para a modelagem. Estes parâmetros configuram o programa para representar, adequadamente, o modelo. Em outras palavras, são definidos a quantidade de processos e a relação de dependência entre eles, além do tamanho do intervalo entre os *checkpoints*, no caso do protocolo *Rollback* Solidário. Após os procedimentos de configuração o modelo pode ser simulado. No diagrama, os depósitos de dados D3 (Lista de Eventos Futuros) e D4 (Estados dos Processos), simbolizam a existência das principais estruturas do sistema que estarão subdivididas nos diversos processos da simulação.

O detalhamento do processo P3 (Simular Modelo) identifica o comportamento do protocolo utilizado para expressar o paralelismo da aplicação e será detalhado na próxima seção.

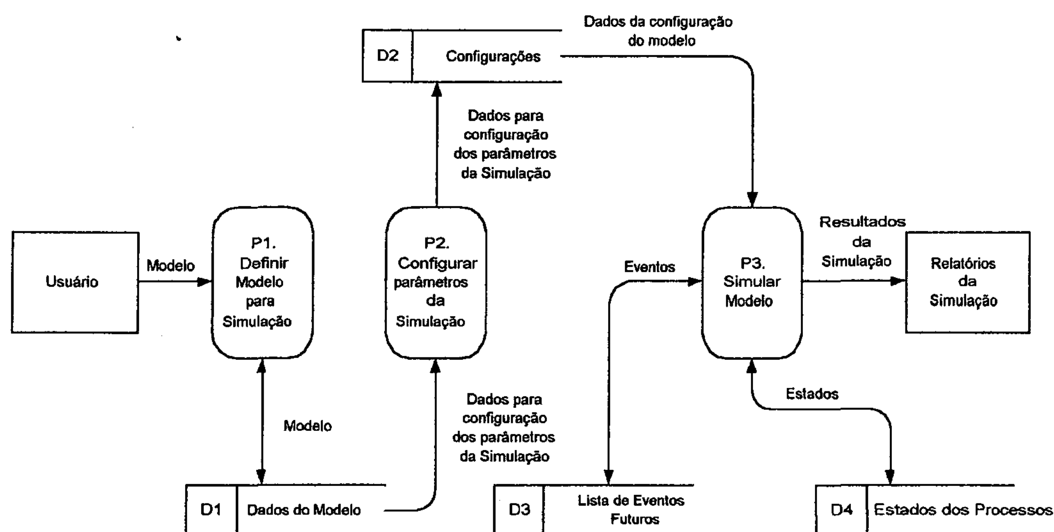

Figura 4.2: DFD representando um ambiente para simulação distribuída

4.2 O Comportamento Geral do Protocolo *Rollback* Solidário

Durante a simulação, os processos lógicos do protocolo *Rollback* Solidário interagem entre si, trocando informações através dos canais de comunicação e se comportando de forma similar aos processos do protocolo *Time Warp*. Todavia, se o processo receber uma mensagem *straggler*, ele identifica o estado salvo (*checkpoint*) em um tempo lógico imediatamente anterior ao *timestamp* da mensagem e o sistema se encarrega de identificar o *checkpoint* global consistente que contenha o estado identificado pelo processo e que produza a menor sobrecarga durante o *rollback*. Isso significa que os processos deverão retornar apenas o suficiente para recompor o sistema.

O diagrama da figura 4.3 representa o comportamento dos processos lógicos da simulação durante o recebimento de uma mensagem. Primeiramente é verificado se a mensagem recebida viola a relação de causa e efeito. Quando esta situação ocorre, o processo identifica para qual estado ele deve retornar e o sistema localiza um *checkpoint* global consistente que contenha o estado do processo. Todos os processos da simulação são sincronizados para continuar a partir do estado recuperado. Dessa forma, em uma simulação distribuída, utilizando o protocolo *Rollback* Solidário, os processos lógicos podem executar os eventos da mesma forma como é realizado no protocolo *Time Warp*, ou seja, sem se preocupar com os problemas de causa e efeito. As diferenças entre esses dois protocolos ocorre nos seguintes pontos:

1. O procedimento utilizado quando uma mensagem *straggler* surge no sistema, conforme apresentado nos parágrafos anteriores;

2. A forma de organizar os *checkpoints* através do conjunto de linhas de recuperação. A seção 4.3 discute esse ponto.

3. A maneira como é determinado o estado para onde os processos deverão retornar, na ocorrência de um *rollback*. Esse procedimento depende do mecanismo de *checkpoint* utilizado, síncrono ou semi-síncrono, e sua discussão se encontra nas seções 4.4 e 4.5, respectivamente.

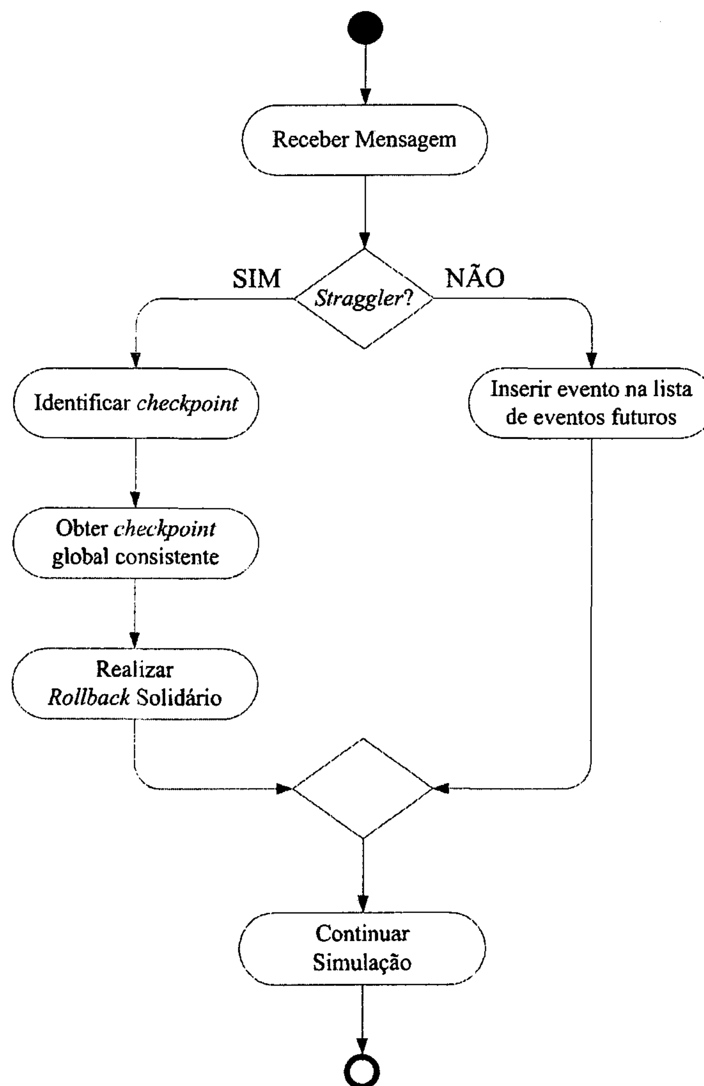

Figura 4.3: Análise de uma mensagem *straggler*

4.3 Conjunto de Linhas de Recuperação

A decisão dos pontos de retorno de cada processo deve ser realizada com eficiência, para que os processos voltem a menor distância possível necessária para recompor o processamento, conquanto um dos objetivos do protocolo *Rollback* Solidário é retornar a computação rapidamente a um *checkpoint* global consistente.

Com a utilização de *checkpoints* globais consistentes, o intervalo em que os estados são armazenados é maior do que nos mecanismos *Copy State Saving* e *Incremental State Saving*. Um dos objetivos é diminuir a área *coast forward*, que é a diferença entre o ponto em que o processo deveria retornar e o *checkpoint* utilizado para recompor a computação, conforme ilustra a figura 4.4. Supondo que um processo p receba, no tempo lógico $t + \alpha$, uma mensagem com evento escalonado para ser executado no tempo t e, portanto, deva retornar para algum tempo lógico menor ou igual a t, este processo retorna para o *checkpoint* local b, que foi realizado antes de t. Resta ao sistema encontrar um *checkpoint* global consistente que contenha b, fazendo com que os demais processos sincronizem nessa linha de recuperação.

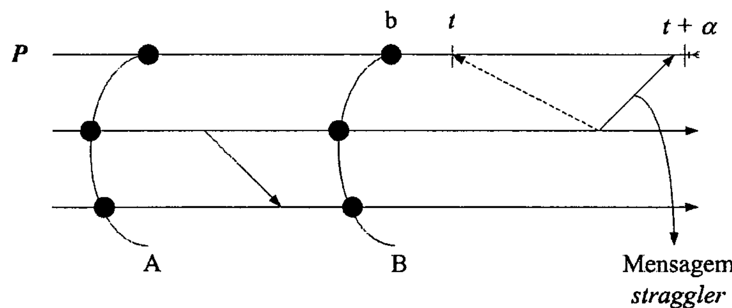

Figura 4.4: Exemplo de Rollback no modelo Solidário

A dificuldade surge quando, ao montar um *checkpoint* global consistente, o sistema volta para um ponto anterior a b, ou ainda, ao combinar o *checkpoint* b com *checkpoints* de outros processos, o sistema acaba forçando os demais a retornar mais do que o necessário. Isto pode ocorrer, pois um mesmo *checkpoint* local pode fazer parte de um ou mais *checkpoints* globais consistentes. No exemplo ilustrado pela figura 4.4 seria o mesmo que voltar a simulação para a linha de recuperação A, ao invés de retrocedê-la para o *checkpoint* global consistente B.

Definição 4.3.1 (Linhas de Recuperação Máxima e Mínima) *Dado um checkpoint ou conjunto de checkpoints concorrentes entre si, o checkpoint global consistente que contém este conjunto, produzindo o menor prejuízo para a reconstituição do sistema, é a linha de recuperação máxima. Por sua vez, o checkpoint global consistente que provoca o maior retorno dos processos é definida como linha de recuperação mínima.*

A figura 4.5 ilustra uma situação onde nem todos os processos precisam retroceder para a mesma linha de recuperação. Nesta figura, o processo P_4 pode retornar para o *checkpoint* b', que faz parte da linha de recuperação B, enquanto os demais processos deverão retornar para os *checkpoints* pertencentes à linha de recuperação A. Conseqüentemente, é necessário armazenar diversas linhas de recuperação.

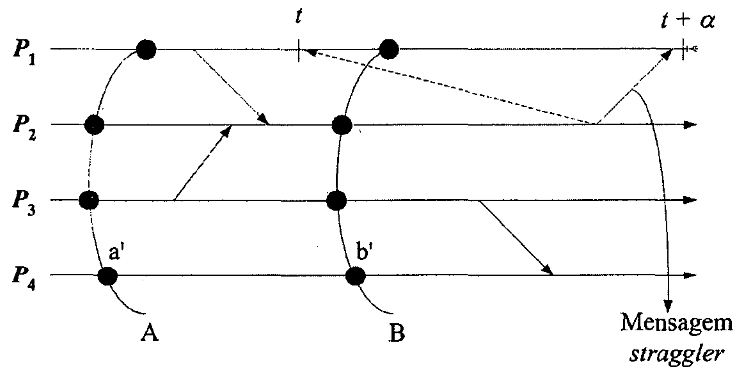

Figura 4.5: Exemplo de linhas de recuperação

É importante destacar que no protocolo *Rollback* Solidário um *checkpoint* pode participar de várias linhas de recuperação. O conceito de visões progressivas permite construir mecanismos de obtenção das linhas de recuperação de forma incremental e na simulação distribuída deve-se permitir o aproveitamento de um mesmo *checkpoint* em várias linhas de recuperação. Portanto, um conjunto de linhas de recuperação é formado por diversas linhas e um único *checkpoint* pode fazer parte de mais de uma linha de recuperação.

4.4 Utilizando *Checkpoints* Síncronos

Como já discutido no capítulo anterior, *checkpoints* síncronos exigem que os processos orquestrem seus *checkpoints* em conjunto para formar um *checkpoint* global consistente. As próximas seções apresentam extensões para os algoritmos *Synch-and-Stop* e *Chandy-Lamport*, devido à simplicidade destes métodos.

4.4.1 Algoritmo *Sync-and-Stop* Estendido

Para utilizar o algoritmo *Sync-and-Stop* (SNS) no protocolo *Rollback* Solidário, foram realizadas alterações para auxiliar na identificação das linhas de recuperação e facilitar a gerência no uso da memória.

Em princípio, cada procedimento de *chekpointing* produz uma linha de recuperação. No entanto, conforme discutido na seção anterior, para melhorar o desempenho do pro-

tocolo, faz-se necessário permitir que um *checkpoint* local pertença a mais de um *checkpoint* global consistente. Para este fim, cada processo mantém um vetor lógico de n posições, onde n é o número de processos que compõem a aplicação. Este vetor informa se houve troca de mensagens. Quando um processo envia ou recebe uma mensagem, ele registra esta informação no vetor. Ao responder ao coordenador com a mensagem `ConfirmaSNS`, os processos adicionam este vetor lógico para que o coordenador possa identificar as dependências entre os *checkpoints* e, assim sendo, localizar novos *checkpoints* globais consistentes, além daquele que é, naturalmente, criado durante o procedimento de *checkpointing*.

Além do vetor lógico, os processos da simulação acrescentam os seus respectivos LVTs na resposta para o coordenador. Durante o cálculo do GVT, o coordenador seleciona o menor LVT e verifica as dependências desse valor com as linhas de recuperação, devolvendo a mensagem de confirmação com o valor do GVT. É importante recordar que não existem mensagens interceptando o corte correspondente à linha de recuperação sendo criada no algoritmo SNS, o que facilita o cálculo do GVT por não haver tratamento de mensagens transientes.

O diagrama 4.6 apresenta o comportamento do algoritmo SNS estendido. No diagrama existem dois fluxos de execução que representam as atividades do processo coordenador e dos demais processos. As setas tracejadas representam as mensagens que são trocadas entre eles. A listagem deste algoritmo pode ser encontrada na seção A.5 do apêndice A.

Apesar da necessidade de sincronizar os processos durante o procedimento de *checkpointing*, esse algoritmo tem a vantagem de ser relativamente simples e exigir pouca memória para o seu funcionamento, uma vez que o cálculo do GVT é continuamente refeito.

4.4.2 Algoritmo *Chandy-Lamport* Estendido

Da mesma forma como foi realizado no algoritmo SNS, para se estender o algoritmo de *Chandy-Lamport*, no intuito de utilizá-lo no protocolo *Rollback* Solidário, será necessário retornar ao coordenador as informações dos LVTs de todos os processos participantes da computação. Todavia, deve-se recordar que nem todos os processos se comunicam diretamente com o coordenador, de maneira que o procedimento de cálculo do GVT será um pouco diferente daquele apresentado no algoritmo *Sync-and-Stop* Estendido.

O diagrama da figura 4.7 demonstra o funcionamento do algoritmo *Chandy-Lamport* estendido. O processo coordenador inicia o procedimento enviando a mensagem `IniciaCL` para todos os processos que possuem canal de comunicação com ele. Durante o envio da mensagem `IniciaCL` todos os processos rotulam a mensagem com o valor do seu

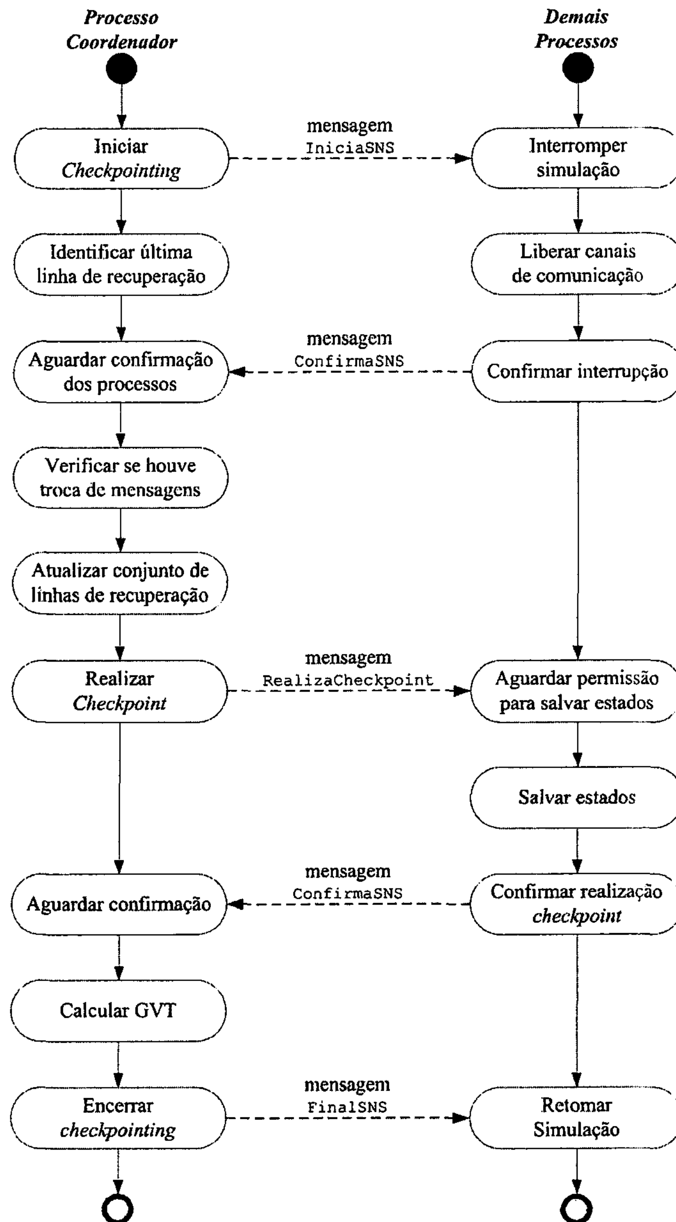

Figura 4.6: Comportamento do algoritmo SNS estendido

LVT. Quando todas as mensagens `IniciaCL` chegam pelos canais de comunicação de um processo, ele calcula o GVT em relação a esses processos. Os processos encerram esta etapa enviando a mensagem `FinalProcessoCL` para o coordenador com os respectivos GVTs locais. O processo coordenador pode, então, calcular o GVT geral e encerrar o procedimento de *checkpointing* através da mensagem `FinalCL` que contém o valor do GVT.

É importante lembrar que, enquanto os processos aguardam a mensagem `IniciaCL`, se alguma mensagem chegar pelos respectivos canais ela deverá ser armazenada.

Como nem todos os processos possuem comunicação direta com o coordenador o envio do vetor lógico, que sinaliza a troca de mensagens para atualização das linhas de recuperação, deve ser feito através de uma matriz quadrada de ordem n, que representa o número de processos. Cada linha desta matriz representa o vetor de um processo. Ao enviar a mensagem `IniciaCl` cada processo atualiza a sua respectiva linha. Desta forma, o coordenador pode atualizar o conjunto de linhas de recuperação como foi explicado no algoritmo SNS estendido.

O detalhamento do algoritmo *Chandy-Lamport* estendido se encontra seção A.6 do apêndice A.

4.4.3 Tratamento dos *Rollbacks* na Abordagem Síncrona

Quando um processo recebe uma mensagem *straggler*, ele envia o *timestamp* da mensagem para o coordenador avisando-o através do uma mensagem de *rollback*. O processo coordenador identifica a respectiva linha de recuperação em seu conjunto, enviando, em seguida, uma mensagem a todos os processos com o vetor de dependências entre os *checkpoints*. O vetor de dependências identifica o *checkpoint* global consistente em que a simulação deve retomar o seu processamento.

O diagrama da figura 4.8 detalha as ações realizadas durante o procedimento de *rollback*. Ao receber o vetor de dependências, cada processo restaura o seu estado correspondente, isto é, o estado que foi armazenado pelo *checkpoint* especificado pelo processo observador. Destaca-se o fato de que o procedimento de *rollback* também é sincrono e todos os processos ficam bloqueados até a sua finalização pelo processo coordenador.

O detalhamento do algoritmo se encontra na seção A.7 do apêndice A.

4.5 *Rollback* Solidário com mecanismos de *Checkpoints* Semi-Síncronos

A base para construção do protocolo *Rollback* Solidário, considerando a abordagem semi-síncrona para obtenção dos *checkpoints*, será o algoritmo FDAS (*Fixed Dependency After Send*), mas é possível utilizar qualquer algoritmo desta abordagem, desde que seja garantida a propriedade *Rollback-Dependency Trackability*.

Inicialmente, para a construção do conjunto de linhas de recuperação será utilizado um processo observador responsável pela montagem das linhas. Este processo tem uma postura passiva durante a simulação, com exceção dos procedimentos de *rollback* quando assume a tarefa de identificar o *checkpoint* global consistente para onde a simulação deve

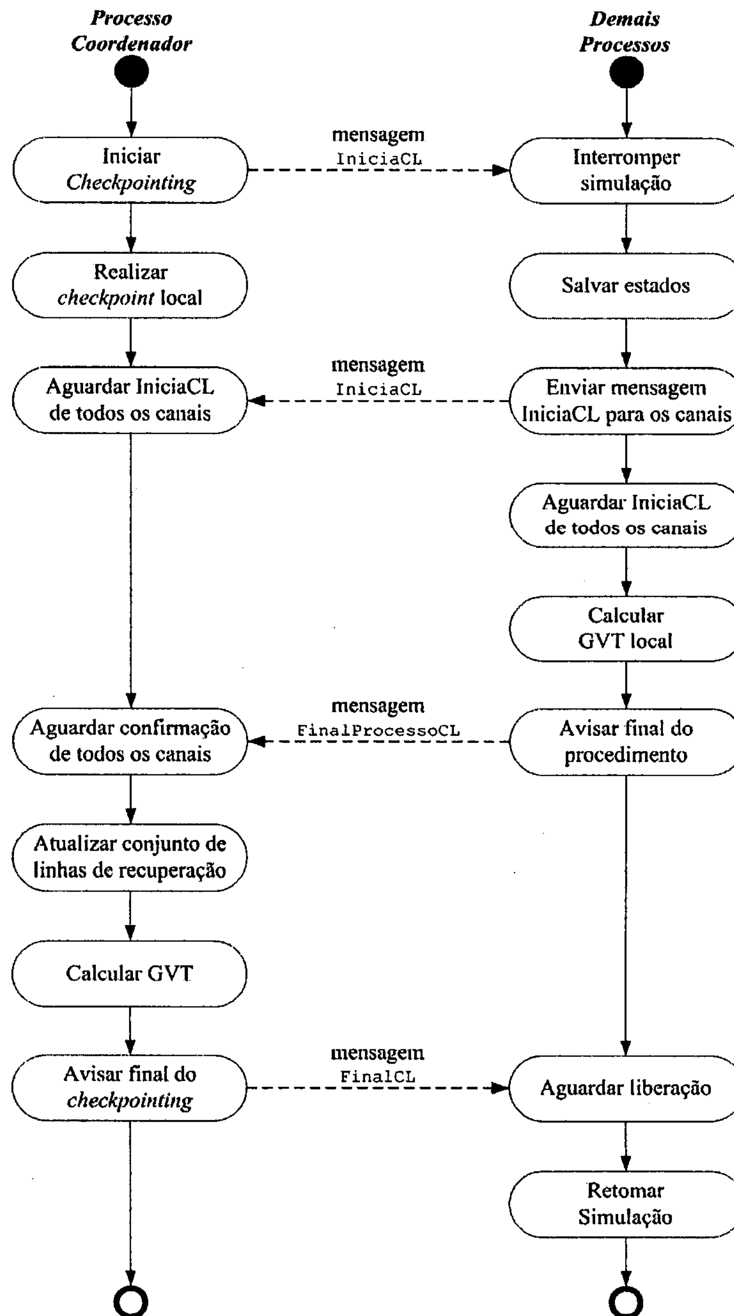

Figura 4.7: Comportamento do algoritmo Chandy-Lamport estendido

retornar.

Quando um processo da simulação realiza um *checkpoint*, ele, imediatamente, envia uma mensagem para o observador com o seu vetor de dependências entre *checkpoints*. O processo observador, por sua vez, mantém uma matriz quadrada de ordem n (matriz M), sendo n o número de processos da simulação. Cada linha i da matriz M representa o último vetor de dependências recebido do processo p_i, ou seja, é a informação das

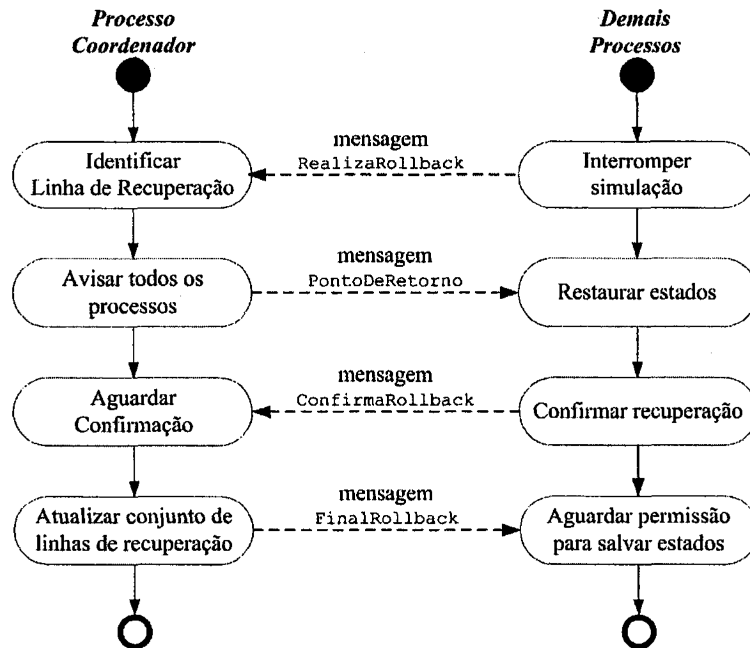

Figura 4.8: *Rollback* Solidário Síncrono ou Coordenado

dependências do último *checkpoint* realizado pelo processo p_i. Deste modo, assim que o processo observador recebe uma mensagem avisando da ocorrência de um *checkpoint* em p_i, ele realiza a seguinte operação:

$$\forall k : 1 \leq k \leq n : M[p, k] \leftarrow m.VD[k].$$

A matriz M é iniciada como uma matriz identidade, representando os *checkpoints* iniciais de cada processo. A cada atualização que M recebe, o vetor que representa a diagonal principal de M poderá identificar uma nova linha de recuperação com o número cronológico dos *checkpoints* de cada processo. Para isso, em uma ordenação de eventos, a diagonal principal deve sempre possuir os maiores valores, representando o último *checkpoint* de cada processo.

Quando o observador recebe uma mensagem do processo p_i com informações mais atualizadas do processo p_j $(i \neq j)$, esta situação pode ser identificada analisando-se as colunas da matriz M. Em cada coluna, os elementos que não fazem parte da diagonal devem ser menores ou iguais ao elemento pertencente a diagonal, ou seja:

$$\forall k : 1 \leq k \leq n \quad \wedge \quad k \neq j : M[k, j] \leq M[j, j]$$

Posto que, cada processo lógico possui a informação mais atualizada a respeito de si mesmo. Se a modificação da matriz levar a uma situação que contradiz essa regra,

então a informação sobre uma nova linha de recuperação ainda não está disponível e o processo observador deve aguardar a chegada de novas mensagens para identificar um novo *checkpoint* global consistente. A listagem A.8, no apêndice A, apresenta os detalhes deste procedimento.

As linhas de recuperação extraídas da matriz M são mantidas em uma estrutura de dados lista encadeada para serem utilizadas durante os *rollbacks*.

O diagrama da figura 4.9 ajuda ilustrar o comportamento do processo observador na obtenção das linhas de recuperação.

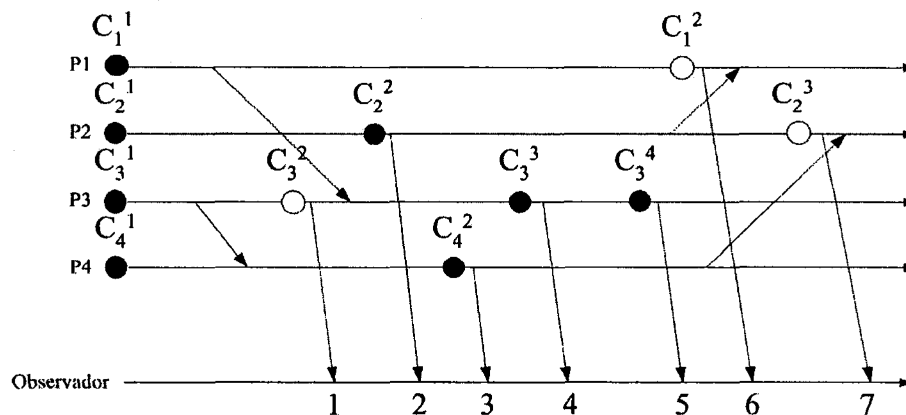

Figura 4.9: Mensagens para o processo observador

O observador inicia M como uma matriz identidade, para representar o primeiro *checkpoint* de cada processo:

$$M_{4\times4} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Ele também registra a primeira linha de recuperação representada pelo vetor $[1, 1, 1, 1]$. Ao receber a primeira mensagem, o observador é informado da ocorrência do *checkpoint* C_3^2 no processo P_3. A linha 3 da matriz M é substituída pelo vetor de dependências recebido e uma nova linha de recuperação é extraída da matriz:

$$M_{4\times4} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \mathit{0} & \mathit{0} & \mathit{2} & \mathit{0} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Este procedimento se repete para os *checkpoints* C_2^2 e C_4^2, porém ao receber a informação da ocorrência do *checkpoint* C_3^3, através da quarta mensagem, a matriz M passa a valer:

$$M_{4\times4} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ \mathit{1} & \mathit{0} & \mathit{3} & \mathit{0} \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

Neste momento, a diagonal da matriz M ainda não está representando uma nova linha de recuperação, pois o *checkpoint* C_3^3 tem relação de dependência com o *checkpoint* C_1^1. Isto pode ser identificado na matriz M toda vez que os elementos de uma coluna que não fizerem parte da diagonal principal possuírem valores iguais ou maiores do que o elemento pertencente à diagonal, como ocorre neste instante, uma vez que $M[1,1] = M[3,1] = 1$. Somente ao receber a sexta mensagem, o processo observador será capaz de identificar a linha de recuperação $[2,2,4,2]$, pois a matriz estará valendo:

$$M_{4\times4} = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 1 & 0 & 4 & 0 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

Devido ao tratamento de consistência das mensagens, o mecanismo apresentado nem sempre identifica todas as linhas de recuperação existentes na simulação. No exemplo da figura 4.9 a linha de recuperação $[2,2,3,2]$ não foi obtida, pois quando o processo observador recebeu a informação do *checkpoint* C_1^2, ele já tinha substituído a linha 3 da matriz com a informação do *checkpoint* C_3^4. Como foi explicado no início do capítulo, esta limitação pode prejudicar o desempenho da simulação devido ao aumento da distância percorrida pelos processos durante um *rollback*. A sua vantagem consiste em disponibilizar uma linha de recuperação rapidamente toda vez que o sistema precisa retornar para um *checkpoint* global consistente.

Para resolver esta limitação pode-se utilizar listas encadeadas ao invés da matriz M. O observador manterá uma lista encadeada para cada processo do sistema e toda vez que receber a informação de um novo *checkpoint*, ele deverá inserir o vetor de dependências na lista do respectivo processo. Quando um aviso de *rollback* chegar, o observador deve percorrer as listas de cada processo para identificar os *checkpoints* adequados para a constituição da linha de recuperação. Entretanto, esta abordagem tem a desvantagem de atrasar a resposta do observador a um pedido de *rollback*.

Para maior eficiência da simulação, as listas devem armazenar os vetores de dependências em ordem decrescente, ou seja, o último *checkpoint* sempre será identificado como

o primeiro elemento da lista. A listagem A.9 (apêndice A) apresenta a definição da estrutura de dados para as listas e o procedimento que busca identificar a linha de recuperação máxima que contém um determinado *checkpoint*. O procedimento faz com que o ponteiro `Atual`, referente a cada processo, aponte para nó com o vetor de dependências dos respectivos *checkpoints* de retorno da linha de recuperação máxima. É interessante notar que o procedimento trabalha utilizando um vetor de ponteiros, mas o efeito é de uma matriz quadrada de ordem n, sendo cada linha representada pelo vetor de dependências endereçado pela variável `Atual`.

A figura 4.10 ilustra o uso de listas encadeadas para armazenar os vetores de dependências dos *checkpoints* da computação representada pelo diagrama da figura 4.9.

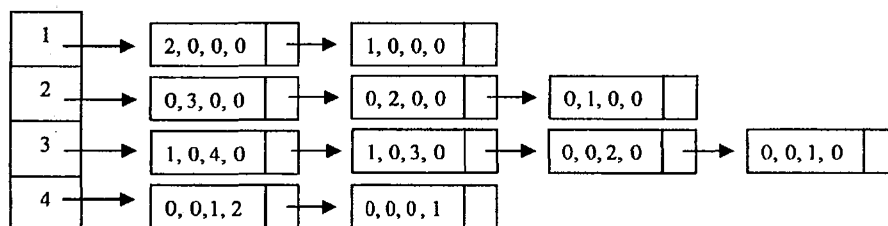

Figura 4.10: Listas encadeadas que armazenam os vetores de dependências dos *checkpoints*

4.5.1 Tratamento dos *Rollbacks* na Abordagem Semi-Síncrona com Observador

O mecanismo de *rollback* da abordagem semi-síncrona é semelhante ao mecanismo da abordagem síncrona, uma vez que o processo observador possui funções semelhantes àquelas do processo coordenador, durante o procedimento de *rollback*. A diferença está nas informações propagadas por estes processos. No primeiro, o coordenador envia um vetor com relógios lógicos, já no segundo mecanismo, o observador propaga um vetor com os respectivos *checkpoints* para onde os processos devem retroceder. É responsabilidade de cada processo atualizar o seu LVT durante a restauração dos estados. Além disso, nesta abordagem, não há necessidade de sincronizar o final do *rollback* entre o processo observador e os demais processos da simulação e, por conseguinte, o tratamento do *rollback* pelo processo observador encerra-se com o envio da mensagem `CheckpointDeRetorno`.

O diagrama 4.11 apresenta as atividades utilizados pelo processo observador durante um *rollback* e a relação de sincronismo com os processos da simulação.

A abordagem semi-síncrona para a obtenção dos *checkpoints* necessita de um tratamento especial para as mensagens da aplicação, pois, como não existe sincronização dos processos durante a obtenção dos *checkpoints*, é provável que existam mensagens interceptando os cortes correspondentes às linhas de recuperação. No exemplo da seção

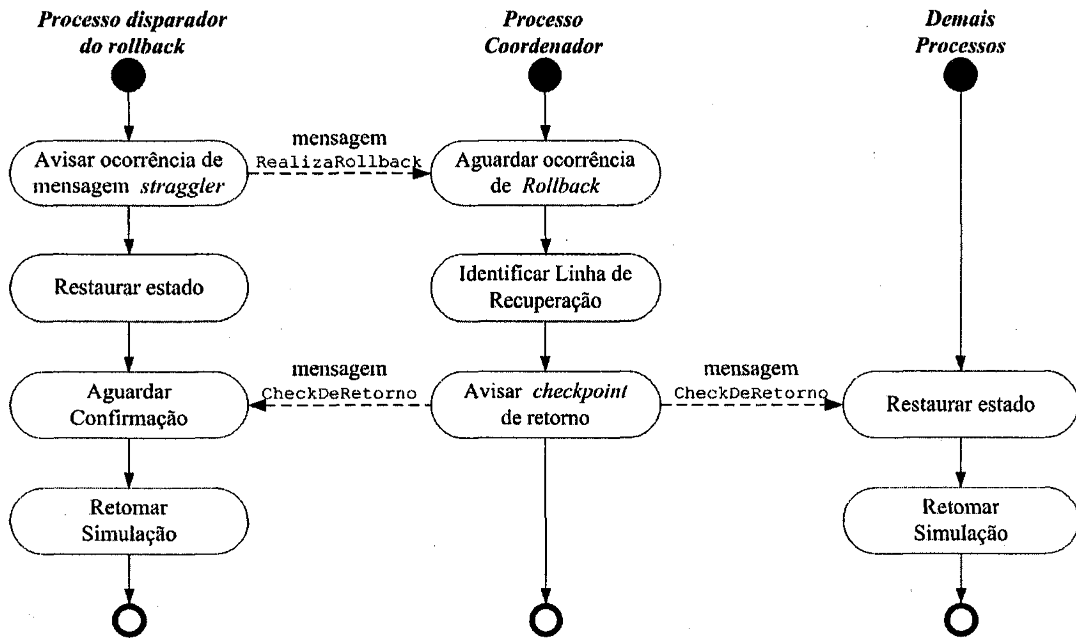

Figura 4.11: Comportamento do processo observador durante um *rollback*

anterior, o sistema identificou a linha de recuperação $[1, 1, 2, 1]$, entretanto, existe uma mensagem entre os *checkpoints* C_3^2 e C_4^1 enviada pelo processo P_3 com destino ao processo P_4. Se esta mensagem não for armazenada pelo processo P_4, uma reconstituição do sistema nesta linha de recuperação irá produzir uma imagem incompleta da computação e, portanto, inconsistente, pois existirá uma mensagem que foi enviada pelo processo P_3, mas que não foi recebida pelo processo P_4.

Em sistemas tolerantes a falhas, onde os mecanismos de *checkpoints* assíncronos ou semi-síncronos são utilizados, há tratamento especial às mensagens do sistema que, em algumas soluções, são armazenadas em arquivos próprios (STROM; YEMINI, 1985).

No protocolo *Rollback* Solidário a solução é semelhante à utilizada no protocolo *Time Warp*. Neste protocolo, as mensagens de entrada não são descartadas após terem sido processadas. Quando ocorre *rollback*, o processo atualiza o ponteiro da lista de eventos futuros com o endereço do primeiro elemento da lista com LVT maior ou igual ao seu ponto de retorno e re-executa os eventos. A diferença consiste no fato de que, no protocolo *Rollback* Solidário, nem todas as mensagens devem ser preservadas, uma vez que outros processos poderão ter sido afetados pelo *rollback* e, desta forma, é possível que algumas mensagens devam ser, realmente, descartadas.

No tratamento das mensagens, o processo que está realizando *rollback* deve identificar o processo emissor (*id*) de cada mensagem recebida com LVT de criação maior que o seu ponto de retorno e comparar se o valor do vetor de dependências, na posição pertinente ao processo emissor, ou seja, $VD[id]$, é menor do que o tempo de criação da mensagem.

Neste caso, a mensagem deve ser descartada.

4.6 Problemas com o Protocolo *Rollback* Solidário

O protocolo *Rollback* Solidário, da forma como foi apresentado, possui algumas limitações que serão tratadas nesta seção. A primeira delas será denominada "Anomalia do Retorno Desnecessário". Esta situação ocorre quando um processo é forçado a retroceder mesmo quando não recebeu mensagens dos processos envolvidos no *rollback* e, portanto, o seu estado atual não possui relação de dependência com os estados desses processos. Este problema é uma desvantagem do protocolo *Rollback* Solidário em relação ao protocolo *Time Warp*, pois exige que todos os processos realizem *rollback*, mesmo aqueles que, em situação idêntica, não receberiam anti-mensagens no protocolo *Time Warp*.

O processo observador, quando recebe uma mensagem de *rollback*, localiza a linha de recuperação adequada para o retorno dos processos da simulação, conforme exposto nas seções anteriores. Mesmo quando um processo não é afetado por um *rollback*, ele receberá a mensagem `CheckpointDeRetorno` para que recupere o estado pertencente à linha de recuperação especificada pelo observador. Dessa forma, ele será forçado a retornar para o seu último *checkpoint*, ou melhor, para o último *checkpoint* que o observador teve conhecimento.

Outro problema, que está relacionado com a "Anomalia do Retorno Desnecessário", é que se todos os processos retornam para a última linha de recuperação identificada pelo processo observador, o evento que provocou o *rollback* deixa de existir e poderá ser gerado novamente quando a simulação retomar a sua execução. Neste caso, o sistema entrará em um ciclo de *rollbacks* e não progredirá, situação conhecida como *livelock*.

Em adição, existe a possibilidade de acontecer *rollbacks* simultâneos quando mais de um processo recebe mensagens *stragglers* e iniciam, concorrentemente, o procedimento de *rollback*. Esta situação poderá conduzir a simulação para condições de inconsistência devido à perda de informações, dependendo da ordem em que são tratados os *rollbacks* simultâneos pelo processo observador.

As próximas seções discutem algumas soluções para estes problemas.

4.6.1 O Problema da "Anomalia do Retorno Desnecessário"

A solução para o problema da "Anomalia do Retorno Desnecessário" considera que, quando o observador envia o vetor com os pontos de retorno de cada processo, ele deve acrescentar a identificação do processo que provocou o *rollback*. Assim que cada processo recebe esta mensagem, ao invés de restaurar, imediatamente, o estado representado pelo

identificador do *checkpoint*, ele verifica se o seu vetor de dependências atual possui relação causal com o estado para o qual o processo que provocou o *rollback* irá retornar. Desta forma, supondo que um processo p_i recebeu uma mensagem *straggler* e deva retornar para o estado representado pelo *checkpoint* C_i^x, ele, imediatamente, avisa o processo observador desta situação. O processo observador irá localizar uma linha de recuperação que contenha C_i^x e enviar o respectivo vetor de dependências (VD) para todos os processos informando, também, quem disparou o procedimento de *rollback*, ou seja, p_i. Ao receber o vetor VD, cada processo verifica se o seu estado atual tem relação causal com o estado para o qual p_i retornou. Isso é realizado com a comparação da posição i do vetor de dependências local com o valor x do processo p_i. Se o valor encontrado for igual ou maior do que x, o processo deverá retornar para o ponto indicado pelo processo observador, caso contrário, poderá continuar o seu processamento. Essa continuação é consistente, pois o *checkpoint* C_i^x é o ponto de referência para a composição da linha de recuperação, mas o processo observador não possui informações atualizadas sobre cada processo, uma vez que se trata de uma computação assíncrona.

Quando um processo não realiza *rollback*, o processo observador passa a ter uma informação incorreta da linha de recuperação para onde retornou a computação. Por causa disto, quando um processo não retorna, após o aviso do observador, ele deve realizar um *checkpoint* e informar o processo observador, que passa a considerar este valor na linha de recuperação do sistema, após a mensagem *straggler*.

A solução para este problema impõe uma outra dificuldade: como cada processo necessita saber os pontos de retorno dos demais, para decidir sobre o descarte ou validação das mensagens já recebidas, a visão desatualizada da linha de recuperação pelo observador leva os processos a um impasse. Resolver este novo problema pode provocar temporariamente a diminuição do otimismo da simulação após cada *rollback*. Uma alternativa é permitir que os processos continuem suas execuções até o instante em que necessitem tratar uma mensagem que foi enviada antes do rollback (época passada). Obviamente, esta mensagem não foi descartada, pois não se tratava de uma mensagem transiente. Por conseguinte, o processo deve aguardar pela linha de recuperação definitiva que será identificada pelo observador após ele encerrar o procedimento de *rollback*. Desta forma, o processo observador deve enviar mais uma mensagem, ratificando ou corrigindo a linha de recuperação. Cada *rollback* irá alterar a época em que se encontra o sistema. Este mecanismo será discutido no tratamento para o problema dos *rollbacks* simultâneos.

4.6.2 O Problema dos *Rollbacks* Simultâneos

O problema dos *rollbacks* simultâneos pode levar a simulação para diversas inconsistências, devido à possibilidade de estar sendo considerada linhas de recuperação diferentes

nos *rollbacks*.

Ao tratar um pedido de *rollback*, o processo observador também restaura o seu respectivo estado. Ao receber outro pedido de *rollback*, o tempo lógico pode não estar condizente com o tempo para o qual retrocedeu a simulação durante o tratamento do primeiro *rollback* e o processo observador não será capaz de identificar se o pedido de *rollback* aconteceu após a recuperação do estado geral do sistema ou não.

Como os processos executam de maneira assíncrona, é impossível impedir que *rollbacks* simultâneos aconteçam. Entretanto, é possível apresentar uma solução para o problema. A solução descrita é semelhante à técnica utilizada para resolver o problema da computação órfã em ambientes cliente×servidor, denominada "Reencarnação". Quando a simulação inicia ou re-inicia, todos os processos, inclusive o observador, começam uma nova época. Esta época é numerada partindo do ordinal 1.

Durante o pedido de *rollback*, o processo envia a identificação da sua época atual. O processo observador verifica se o pedido de *rollback* faz parte da mesma época em que ele se encontra. Neste caso, o procedimento de *rollback* ocorre normalmente e o processo observador pede aos demais processos para incrementarem as suas respectivas épocas, inclusive os processos que não serão afetados pelo referido *rollback*.

Quando o processo observador recebe um pedido de *rollback* referente à uma época anterior a atual, ele verifica se os processos envolvidos foram afetados pelo último *rollback* e, neste caso, descarta a mensagem, caso contrário, ele procede o tratamento do *rollback*. Este procedimento apresenta o inconveniente de não ser compatível com a solução apresentada para o problema da "Anomalia do Retorno Desnecessário", uma vez que, na solução deste problema, alguns processos não respeitam a linha de recuperação indicada pelo observador.

Para compatibilizar os dois problemas, devem ser feitas algumas melhorias na solução apresentada. Uma alternativa é fazer com que o observador aguarde por uma confirmação da realização do procedimento *rollback* por todos os processos. Assim, ele terá noção dos processos que retornaram e daqueles que continuaram sua execução. Lembrando que o processo que não foi afetado pelo *rollback* deve realizar um *checkpoint* e avisar o observador. Por conseguinte, esta mensagem também representa uma das respostas que o observador aguarda. Durante este procedimento, o processo observador não dará tratamento a nenhum pedido de *rollback*, colocando os novos pedidos em uma fila de espera.

Um detalhe que auxilia no tratamento deste problema é que os processos que recebem uma mensagem *straggler*, necessariamente, realizam *rollback*. Como são os processos que enviam o ponto de retorno para o observador, eles devem ficar aguardando a linha de recuperação identificada por ele. Se o ponto de retorno for o mesmo pedido pelo

processo ou representar um estado posterior, o processo restaura o estado identificado pelo observador avisando-o, em seguida, para desconsiderar o pedido de *rollback* da época anterior, avançando a sua época. Caso contrário, ele deverá avisar o observador que a linha de recuperação especificada não atende às suas necessidades e, deverá continuar aguardando uma solução por parte do observador. É importante lembrar que, na solução da "Anomalia do Retorno Desnecessário", o processo observador identifica o causador do *rollback* na mensagem que ele envia para todos os processos, facilitando aos que receberam a mensagem *straggler* identificar se o tratamento dado pelo observador se refere ao seu *rollback*. Em adição, o tratamento dado ao primeiro *rollback* pode provocar o retorno do processo que produziu a mensagem *straggler* do segundo *rollback*, eliminando a necessidade de tratamento deste.

O observador, por sua vez, só terá concluído o procedimento de *rollback* quando receber a resposta de todos os processos da simulação. Se isto ocorrer, havendo pedidos de *rollbacks* referentes a épocas anteriores, eles poderão ser descartados. Se durante a espera pelas respostas dos processos o observador receber pedidos de reconsideração, ele tratará destes pedidos em seqüência, após receber a confirmação dos demais processos.

4.6.3 O Problema da Mensagem Transiente

A utilização da técnica da "Reencarnação", para resolver o problema dos *rollbacks* simultâneos, também auxilia na solução do problema da mensagem transiente.

Esta situação ocorre quando um processo p_i qualquer escalona um evento para ser tratado pelo processo p_j, com $i \neq j$, no tempo t. Uma mensagem m é, então, enviada do processo p_i para o processo p_j, porém, antes que a mensagem chegue, o processo p_j é notificado sobre a ocorrência de *rollback* e restaura um estado armazenado anteriormente. Após este procedimento, p_j recebe a mensagem m tratando-a normalmente. Entretanto, se o processo p_i realizou *rollback* para um tempo anterior à criação da mensagem, a mensagem m deveria ser eliminada.

Para resolver este problema, será necessário rotular cada mensagem com a respectiva época do processo emissor. Com isso, toda mensagem que chegar em um processo tendo como *timestamp* uma época anterior, irá exigir um tratamento especial por parte do processo receptor, antes que seja processada ou descartada. Este tratamento implica em comparar o LVT de criação da mensagem com o *checkpoint* para onde processo emissor retornou. Novamente, mensagens com tempo lógico de criação maior do que o ponto de retorno de seus criadores deverão ser descartadas.

Na solução deste problema o processo envolvido necessita da linha de recuperação e, portanto, se o observador ainda não tiver concluído o procedimento de *rollback* ele deverá

aguardar até o recebimento da mensagem `FinalizaRollback`.

4.7 Estrutura das Mensagens no Protocolo *Rollback* Solidário

Devido às soluções propostas para os problemas apresentados, o cabeçalho de uma mensagem, no protocolo *Rollback* Solidário, deve conter a estrutura da figura 4.12. O campo *Kind* identifica o tipo de mensagem que trafega no sistema, considerando a abordagem semi-síncrona. Os tipos possíveis são:

- `Normal`: Mensagem utilizada pelos processos nas comunicações pertinentes à aplicação.
- `RealizaRollback`: Mensagem enviada para o observador sempre que um processo recebe uma mensagem *straggler* e deve realizar um *rollback*.
- `CheckpointDeRetorno`: Mensagem de resposta do processo observador a um pedido de *rollback*. Esta mensagem carrega o vetor de dependências com a linha de recuperação em que a computação deve ser retrocedida.
- `ConfirmaRollback`: Mensagem enviada para o observador quando o processo retorna para a linha de recuperação especificada.
- `ConfirmaContinuação`: Mensagem enviada para o observador quando o processo não precisa retornar para a linha de recuperação indicada. O processo envia o vetor de dependências, possibilitando ao observador identificar se ele continuou o processamento ou retornou ao ponto designado.
- `DesconsideraPedidoDeRollback`: Mensagem utilizada pelos processos durante a situação de *rollbacks* simultâneos, quando a solução de um *rollback* automaticamente resolve pedidos anteriores.
- `ReconsideraPedidoDeRollback`: Mensagem utilizada para pedir reconsideração do processo observador nos pedidos de *rollbacks* simultâneos, quando a solução de um *rollback* não resolve outros pedidos.
- `FinalizaRollback`: Mensagem utilizada para encerrar o procedimento de *rollback*.

Os campos *LVT Criação* e *LVT Evento* representam o tempo lógico da criação do evento e o tempo definido para o seu escalonamento, respectivamente. A distinção entre o tempo de criação e o tempo para o qual o evento foi escalonado é devido a

necessidade de descartar as mensagens que foram enviadas em um tempo lógico superior ao ponto de retorno dos processos que realizaram *rollback*. O campo *Época* corresponde à informação da época em que a mensagem foi enviada. *VD* é o vetor de dependências entre *checkpoints*, e *Tag* é um campo adicional que pode ser utilizado pela aplicação para armazenar informações pertinentes à simulação. Este campo também é usado para identificar o processo que provocou *rollback*, quando o observador envia o vetor com os *checkpoints* de retorno de cada processo.

Kind	LVT Criação	LVT Evento	Época	VD	Tag

Figura 4.12: Estrutura de uma mensagem no protocolo *Rollback* Solidário

4.8 *Rollback* Solidário sem o Processo Observador

Inicialmente, o protocolo *Rollback* Solidário foi proposto utilizando um processo observador, conforme descrição das seções anteriores, entretanto, é possível obter resultados semelhantes sem a presença daquele. Neste caso, cada processo passa a ser responsável em avisar aos demais sobre a ocorrência do *rollback* e gerenciar o procedimento quando recebem uma mensagem *straggler*.

Ao receber uma mensagem que fere a relação de causa e efeito, o processo restaura o estado imediatamente anterior ao tempo lógico da mensagem e envia para os demais processos da simulação o vetor de dependências do *checkpoint* restaurado, em uma mensagem que sinaliza o *rollback*.

Quando um processo p_j recebe uma mensagem `RealizaRollback` do processo p_i, p_j verifica se o valor do seu vetor de dependências na posição i ($VD_j[i]$) é menor do que o valor do vetor recebido na mesma posição ($m.VD[i]$), ou seja, o processo p_j verifica se o seu estado atual tem relação de dependência com o estado recuperado pelo processo p_i e, portanto, não podem estar juntos em uma mesma linha de recuperação. Se esta situação acontece, p_j continua o seu processamento, caso contrário, ele deve restaurar o último estado salvo tal que o valor da posição i do seu vetor de dependências seja menor do que o valor correspondente no vetor de dependências do processo p_i.

O processo que informou sobre a ocorrência do *rollback* assume as funções do observador, por conseguinte, ele passa a coordenar o procedimento de restauração da simulação. Após realizar o seu *rollback*, o processo coordenador envia a mensagem `RealizaRollback` para os demais processos da simulação. Cada processo, após verificar se necessita realizar *rollback*, responde ao coordenador com a mensagem `ConfirmaRollback` ou `ConfirmaContinuação`, conforme tenha sido a sua decisão. O coordenador, por sua vez, deve construir

um vetor que representará a linha de recuperação com os estados para onde a computação retrocedeu. Cada posição deste vetor é atualizado com o número que identifica o respectivo *checkpoint* de cada processo. É importante destacar o fato de que se um processo não precisou retornar ele realizou um *checkpoint* que será considerado pelo coordenador como parte da linha de recuperação.

A figura 4.13 ilustra o que foi exposto. Após ter recebido uma mensagem *straggler* do processo P_3, o processo P_2 necessita realizar um *rollback* e assume a coordenação do procedimento. Inicialmente ele restaura o estado representado pelo *checkpoint* C_2^2, que possui o vetor de dependências valendo $[1,2,0,0]$ e, em seguida, envia a mensagem `RealizaRollback` para todos os outros processos. O processo P_1 ao receber a referida mensagem compara o valor da posição 2 do seu vetor de dependências com o valor da posição 2 do vetor de dependências recebido na mensagem, que corresponde ao vetor recuperado pelo processo P_2. O vetor de dependências entre *checkpoints* do processo P_1 está, neste momento, valendo $[2,2,1,1]$ e, portanto, a posição 2 está com o mesmo valor da posição 2 do vetor do processo coordenador. Isto implica em relação causal entre o estado recuperado do processo P_2 e o estado atual do processo P_1, assim P_1 é forçado a retornar para o último *checkpoint* que ele tenha armazenado cujo vetor de dependências, na posição 2, tenha valor menor do que 2. Por conseguinte, ele restaura o estado representado pelo *checkpoint* C_1^2 e retorna a mensagem `ConfirmaRollback` para o processo P_2. O processo P_3, ao receber `RealizaRollback` compara o seu vetor de dependências com o vetor recebido na posição 2 e, como o seu vetor de dependências está valendo $[0,0,3,0]$ não precisa retornar. Portanto, o processo P_3 salva o estado atual (*checkpoint*) C_3^4 e envia a mensagem `ConfirmaContinuação` contendo o vetor de dependências valendo $[0,0,4,0]$. Situação semelhante ocorre no processo P_4 que não necessita realizar *rollback*. Quando o processo P_2 recebe todas as resposta ele é capaz de identificar a linha de recuperação em que a computação foi reconstituída, ou seja, $[2,2,4,3]$. Finalmente, esta linha é enviada para todos os demais processos, através da mensagem `FinalizaRollback`, e a época é avançada. Destaca-se o fato de que as mensagens recebidas pelo processo P_2, antes do *rollback* não necessitam ser descartadas, pois, os processos responsáveis pelo envio não realizaram *rollback*, inclusive a própria mensagem *straggler*.

O procedimento descrito possui uma falha sutil que pode ser melhor compreendida através de um exemplo. Supondo que o diagrama da figura 4.13 seja alterado para acrescentar as mensagens m_1, m_2 e m_3, além de ser retirado o *checkpoint* C_3^4, conforme ilustra a figura 4.14. Neste caso, os processos P_3 e P_4, quando receberem a mensagem `RealizaRollback` enviada pelo processo P_2, estarão com o vetor de dependências entre *checkpoints* valendo $[2,2,2,2]$ e, por conseguinte, deverão realizar *rollback* para o último *checkpoint* armazenado tal que a posição 2 dos seus vetores de dependências esteja valendo, no máximo, 1. Assim, os processos P_3 e P_4 retornam, respectivamente, para os *checkpoints*

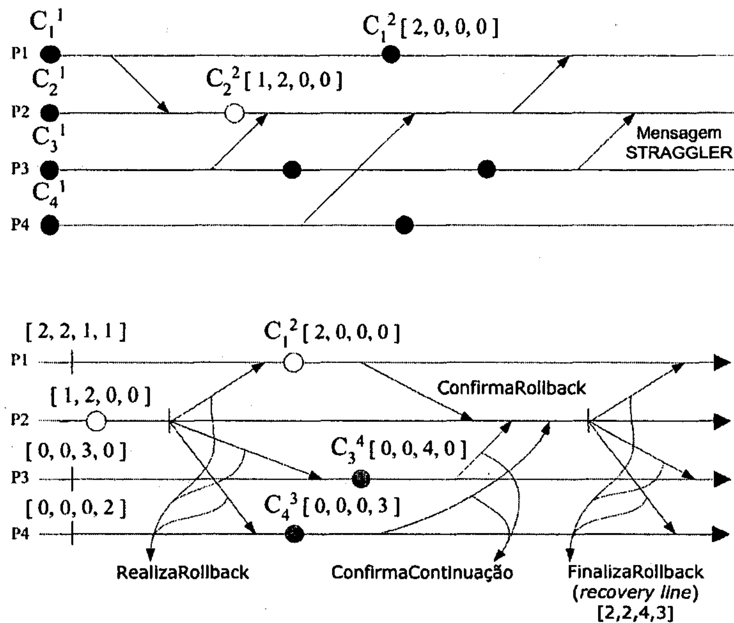

Figura 4.13: Comportamento do protocolo *Rollback* Solidário sem observador durante um *rollback*

C_3^2 e C_4^2. Entretanto, como só foi avaliada a relação de dependência com o *checkpoint* do processo P_2, o método não considerou a possibilidade de dependências entre os *checkpoints* dos demais processos que foram obrigados a realizar *rollback*. No exemplo, o *checkpoint* C_4^2 é dependente do *checkpoint* C_3^2 devido à mensagem m_1.

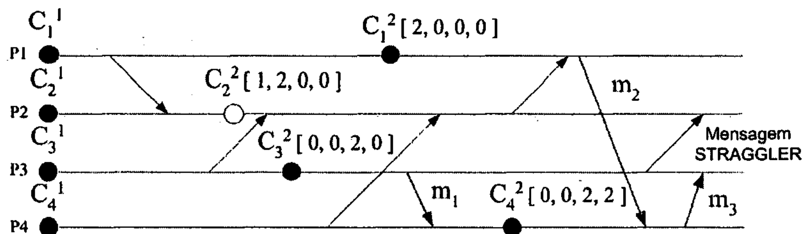

Figura 4.14: Ilustração do problema existente na solução anterior

É importante destacar o fato de que todos os processos que não realizam *rollback* obtém um novo *checkpoint* e, obviamente, estes *checkpoints* não possuem relação causal entre si. Como o algoritmo FDAS satisfaz a propriedade RDT, o conjunto formado pelos novos *checkpoints*, ou seja, os *checkpoints* obtidos pelos processos que não precisaram voltar, mais o *checkpoint* de retorno do processo que disparou o procedimento formam um conjunto que pode fazer parte de um *checkpoint* global consistente. Faz-se necessário, então, encontrar os *checkpoints* dos demais processos, ou seja, daqueles que foram obrigados a realizar *rollback*, que possam formar um *checkpoint* global consistente.

Wang (1997) apresentou algoritmos para obtenção de linhas de recuperação máximas

e mínimas envolvendo um conjunto de *checkpoints*, sendo um por cada processo. Obviamente se deseja encontrar o *checkpoint* global consistente máximo para o retorno da simulação, pois, este conjunto corresponde àquele de prejuízo menor para a computação. Infelizmente, o método apresentado por Wang se baseia na construção de um *R-graph*, que são grafos dirigidos em que nós representam *checkpoints* e arestas representam dependências entre *checkpoints*. A construção do *R-graph* aumenta o custo computacional durante um *rollback*, pois é preciso que os processos da simulação troquem informações a respeito dos seus *checkpoints*.

Desta forma, existem duas opções para tratar o problema. A primeira consiste em aceitar que o sistema construa um *checkpoint* global consistente mínimo, a partir do conjunto dos *checkpoints* dos processos que não retornaram mais o *checkpoint* do processo responsável pelo *rollback*. Wang apresentou um método simples para a obtenção da linha de recuperação mínima R com base em um conjunto S de *checkpoints*. R pode ser calculada com base nos vetores de dependência dos *checkpoints* em S. Para cada processo p_i, basta utilizar o maior valor dos vetores de dependências dos *checkpoints* em S na posição i. O método de Wang se baseia em intervalos de *checkpoints* rotulados à direita e, desta maneira, o vetor de dependências somente é incrementado após a obtenção do *checkpoint* que, no seu registro, não armazena o número do seu *checkpoint*. Por conseguinte, o vetor de dependências associado ao primeiro *checkpoint* do processo $P1$ em uma computação com três processos tem o valor $[0, 0, 0]$, mas imediatamente após a obtenção do *checkpoint* o sistema incrementa o vetor de dependências na posição 1. Voltando ao exemplo do diagrama 4.14, como todos os processos foram obrigados a retornar, somente o vetor de dependências do processo responsável pelo *rollback* pode ser utilizado para identificar a linha de recuperação mínima que será $[1, 1, 0, 0]$.

A segunda opção permite identificar a linha de recuperação máxima que contém o conjunto S. O processo responsável pelo *rollback* (p_r) constrói uma matriz M em que cada linha i corresponde ao vetor de dependências do *checkpoint* para o qual o processo p_i retornou. Nesta matriz o processo p_r pode identificar a existência de *checkpoints* com relação de dependência entre si, verificando se os elementos da diagonal principal representam os maiores valores em cada coluna da matriz, semelhante ao método utilizado na abordagem com observador. Quando a matriz apresentar processos com estados dependentes, p_r deverá enviar uma mensagem para um dos processos envolvidos, ou seja, um processo p_d que deverá restaurar outro estado que não possua dependências com os demais estados restaurados. Neste instante p_d refaz a matriz M. É possível que existam outros processos na mesma situação, ou que passam a ser afetados pela escolha de p_d. Então, p_d escolhe outro processo envolvido e envia a matriz M corrigida. Este procedimento termina quando, ao restaurar um novo estado, a matriz M não identifica mais dependências entre estados. Neste momento, o processo de posse da matriz M

deve retorná-la para p_r que, através da diagonal principal de M, identificará a linha de recuperação para finalizar o procedimento de *rollback* através do envio da mensagem `FinalizaRollback`, com a respectiva linha de recuperação para os demais processos da simulação.

Para ilustrar o que foi explicado, no exemplo da figura 4.14, quando o processo P_2 recebe os pontos de retorno dos demais, ele monta a matriz M com os valores:

$$M_{4\times4} = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 2 & 2 \end{pmatrix}$$

e identifica a dependência entre estados dos processos P_3 e P_4. Neste momento ele envia a matriz M para o processo P_4, uma vez que o estado deste processo depende causalmente do estado do processo P_3. Quando P_4 recebe a matriz M, ele procura por outro estado armazenado que não possua dependência com os demais *checkpoints* dos outros processos. Assim ele é obrigado a retornar para o *checkpoint* C_4^1 que possui o vetor de dependências $[0,0,0,1]$. Ao reconstituir a matriz M, P_4 verifica que a linha de recuperação máxima foi encontrada e retorna a matriz M para o processo responsável que irá finalizar o procedimento de *rollback* com a mensagem `FinalizaRollback`. A linha de recuperação será formada pelos *checkpoints* C_1^1, C_2^2, C_3^2 e C_4^1.

O diagrama da figura 4.15 apresenta os atividades que os processos realizam durante o recebimento das mensagens *straggler* e de *rollback*.

Nesta abordagem, todos os processos assumem as funções que eram do processo observador para o tratamento dos problemas apresentados anteriormente. Naturalmente, o problema da "Anomalia do Retorno Desnecessário" não ocorre, visto que cada processo verifica a necessidade de retornar ou não quando recebe um aviso de *rollback*. O tratamento para os problemas da "Mensagem *Straggler*" é semelhante ao exposto anteriormente.

Devido ao comportamento descentralizado do protocolo *Rollback* Solidário sem Observador durante o tratamento dos *rollbacks*, a solução adotada para o problema dos *rollbacks* simultâneos não pode ser a mesma do mecanismo com observador, uma vez que vários processos responsáveis irão disparar o procedimento de *rollback*. Para tratar deste problema e evitar os inconvenientes de se permitir o tratamento simultâneo de *rollbacks* distintos será preciso utilizar algum método de exclusão mútua (TANENBAUM, 1992; SHAY, 1996).

Ricart e Agrawala (apud TANENBAUM, 1995) apresentaram um método para a obtenção de exclusão mútua através de um algoritmo distribuído. Quando um processo deseja entrar na região crítica, ele constrói uma mensagem contendo o nome da região crítica que ele

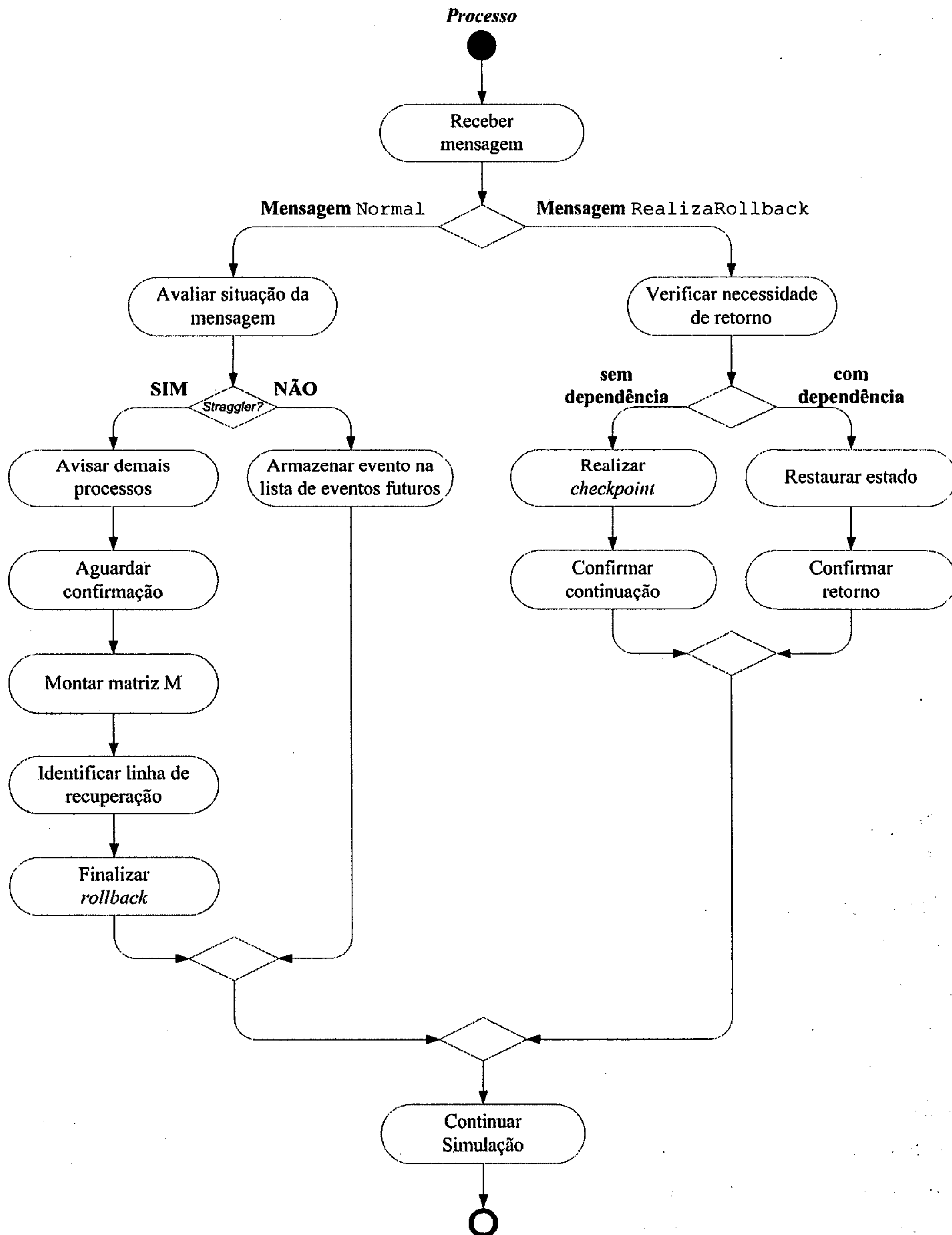

Figura 4.15: *Rollback* Solidário sem observador

deseja utilizar, o número de identificação do processo, e o tempo corrente. Esta mensagem é enviada para todos os outros processos, inclusive ele próprio.

Ao receber uma mensagem de requisição de outro processo, a ação do processo receptor

depende de seu estado com relação a região crítica descrita na mensagem. Três casos devem ser distinguidos:

1. Se o receptor não está na região crítica e não deseja entrar nela, ele envia de volta uma mensagem de OK para o emissor.

2. Se o receptor está na região crítica, ele não responde a mensagem, ao contrário, ele coloca a requisição em um fila de espera.

3. Se o receptor quer entrar na região crítica, mas ainda não conseguiu, ele compara o *timestamp* da mensagem que chegou com o *timestamp* da mensagem que ele enviou. A menor vence. Se a mensagem que chegou é menor, o receptor envia um mensagem OK para o processo emissor. Se a sua mensagem é a que possui o menor *timestamp*, o receptor coloca na fila a mensagem de requisição e não responde ao emissor.

Após enviar todas as mensagens de requisição da região crítica, o processo fica bloqueado aguardando a resposta de todos os outros processos. Assim que todas as mensagens de permissão chegarem, ele pode utilizar a região crítica.

Quando o processo sai da região crítica ele envia uma mensagem OK para todos os processos em sua fila de espera, retirando-os da fila.

Este algoritmo pode ser adaptado para ser usado no protocolo *Rollback* Solidário. Neste caso, o procedimento de *rollback* passa a exigir um sincronismo maior entre os processos havendo necessidade de interrupção da computação, mesmo para os processos não envolvidos no *rollback*.

A mensagem de requisição é a própria mensagem `RealizaRollback`. Quando um processo p_i recebe esta mensagem e não disparou um procedimento de *rollback* ele identifica o seu *checkpoint* de retorno ou realiza um *checkpoint* forçado, conforme já discutido, e responde ao processo responsável. Esta resposta autoriza o controle do *rollback* por parte do processo p_i. O processo p_i interrompe a sua execução até receber a mensagem `FinalizaRollback`. Se o processo p_i receber outra mensagem `RealizaRollback` ele repete o procedimento, fazendo a análise da dependência para o novo estado.

Quando um processo recebe a mensagem `RealizaRollback` e já assumiu o procedimento de *rollback*, ou seja, já se encontra na região crítica, ele não responde a mensagem, colocando-a em uma fila de espera. Se o processo que recebeu a mensagem também disparou o procedimento de *rollback*, ele compara o LVT da mensagem que chegou com o seu LVT. Novamente, a menor vence. Se o pedido que chegou tem um LVT menor, então o processo receptor envia o seu vetor de dependências. Mas, para agilizar o procedimento, a resposta já considera o seu pedido anterior, ou seja, ele devolve o vetor que atende aos

dois pedidos, o seu e o atual. Se o seu LVT é menor ele coloca o pedido na fila e não responde ao emissor.

Quando o processo que está na região crítica termina o procedimento de *rollback* ele envia a mensagem `FinalizaRollback` com a respectiva linha de recuperação. A mensagem `FinalizaRollback` funciona como liberação para os processos que desejam entrar na região crítica. Quando um processo que realizou o pedido de *rollback* simultâneo recebe a mensagem `FinalizaRollback` ele verifica se a linha de recuperação também satisfaz o seu pedido e, neste caso, também envia `FinalizaRollback`, caso contrário, procede a obtenção de um novo *checkpoint* global consistente.

Os processos retornam a sua execução quando receberem uma quantidade de mensagens `FinalizaRollback` proporcional ao número de mensagens `RealizaRollback`.

Finalmente, cada processo deve avançar a época da simulação quando receber uma mensagem *straggler* e for obrigado a sinalizar uma situação de *rollback*. A época é avançada em números de *rollbacks* realizados.

4.9 *Reutilização da Memória*

O protocolo otimista consome muita memória para armazenar os estados salvos. Coleta de lixo (*Garbage Collection*) é a remoção das informações que não tem possibilidade de serem utilizadas em uma situação de *rollback*. Normalmente, o procedimento de coleta de lixo é realizado em sintonia com o cálculo do GVT, ou seja, o GVT é o menor dos LVTs e garante que nenhum processo retornará para um tempo anterior ao GVT, visto que nenhum processo irá produzir mensagens de eventos passados. Por conseguinte, toda informação armazenada com tempo menor que o GVT pode ser descartada.

As técnicas usadas no protocolo *Time Warp* para o cálculo do GVT podem ser diretamente aplicadas no protocolo *Rollback* Solidário, mas como exposto na seção 2.3, essa tarefa não é assíncrona e consome recursos do sistema.

Em sistemas tolerantes a falhas um *checkpoint* obsoleto é aquele que não pode fazer parte de qualquer linha de recuperação futura, mesmo após os *rollbacks* (SCHMIDT et al., 2005). Na simulação distribuída, um *checkpoint* obsoleto é aquele que foi armazenado em um tempo lógico inferior ao GVT atual da simulação.

No protocolo proposto com a utilização do processo observador, o cálculo do GVT pode ser simplificado, uma vez que cada mensagem carrega o vetor de dependências e o LVT do processo. O observador pode, então, identificar o valor do LVT sobre cada *checkpoint* e manter uma relação dos LVTs de cada processo nas respectivas linhas de recuperação. Assim, quando o processo observador for avisado da ocorrência de um *rollback*, ele pode,

juntamente com a linha de recuperação, enviar o valor do GVT. Assim, o GVT será continuamente corrigido e, a cada *rollback*, os processos podem realizar a coleta de lixo na memória.

Na abordagem sem observador, como não existe uma relação direta entre o LVT e o número dos *checkpoints* realizados, não é possível fazer uma associação entre o vetor de dependências e os respectivos LVTs. Assim, uma solução, como exposta anteriormente, aumentaria a carga de cada processo se eles assumissem a função do observador. Uma alternativa para sistemas com poucos processos é alterar o vetor de dependências para que cada posição armazene uma tupla contendo o contador dos *checkpoints* e o respectivo LVT. Isso dobra o tamanho do vetor nas mensagens do sistema, mas agiliza o cálculo do GVT. Para considerar esta solução é preciso avaliar o impacto de manter dados que não serão mais utilizados, devido a um valor desatualizado do GVT, com a necessidade de aumentar o tamanho das mensagens do sistema.

4.10 Considerações Finais

O protocolo *Rollback* Solidário oferece uma forma alternativa para a utilização de protocolos otimistas em simulação distribuída. Este protocolo, que foi discutido neste capítulo, apresenta diversas vantagens em relação ao tradicional *Time Warp* (JEFFERSON, 1985) e suas variantes (STEINMAN, 1992, 1993a; RAJAEL; AYANI; THORELLI, 1993; FERSCHA, 1995; TAY; TEO; KONG, 1997; KALANTERY, 2004), tais como, economia de memória, pois não é necessário armazenar anti-mensagens, maior facilidade para o cálculo do GVT e maior eficiência para retornar a um estado consistente no momento em que ocorre um *rollback*.

Basicamente, o *Rollback* Solidário aplica as técnicas definidas para os Sistemas Distribuídos Tolerantes a Falhas, tais como, *checkpoints* globais consistentes e linhas de recuperação, para obter sincronismo na Simulação Distribuída.

Durante o processo de criação e desenvolvimento do protocolo *Rollback* Solidário foi possível identificar algumas possibilidades de implementação. Essas possibilidades permitem identificar uma hierarquia para as variantes desse protocolo, conforme ilustra a figura 4.16.

O protocolo pode ser dividido em *Rollback* Solidário Síncrono ou Coordenado e *Rollback* Solidário Semi-síncrono. Neste capítulo são apresentados dois algoritmos que coordenam o procedimento de *checkpointing*, demonstrando-se a possibilidade de uso deste tipo de abordagem para o desenvolvimento de programas de simulação distribuídos. Outros algoritmos podem ser utilizados, desde que respeitem as particularidades de um programa para simulação. A grande vantagem desta abordagem é, sem dúvida, o reduzido espaço

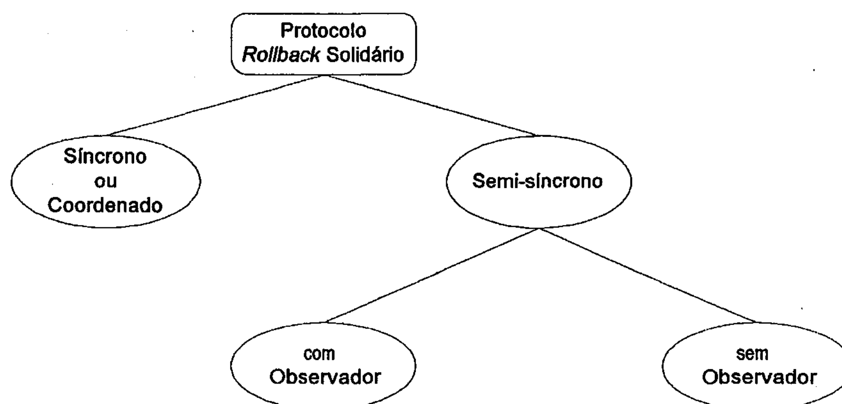

Figura 4.16: Hierarquia de Implementação para o protocolo proposto

de memória necessário para o funcionamento do protocolo.

A abordagem semi-síncrona para obtenção de estados globais consistentes é o foco principal deste trabalho e, neste nível, o protocolo *Rollback* Solidário pode ser implementado utilizando um processo observador para a obtenção das linhas de recuperação e controle do procedimento de *rollback* ou sem a ajuda dele.

O protocolo sem observador impõem um alto custo para os processos da simulação, mas aproveita melhor os elementos de processamento, uma vez que nenhum deles é usado, exclusivamente, para o controle do procedimento de *rollback*. Em compensação, a abordagem com observador, apresenta um mecanismo simples para a obtenção das linhas de recuperação e facilita o tratamento dos problemas discutidos neste capítulo.

Outro detalhe que deve ser ressaltado é a utilização de dois relógios lógicos na simulação. O vetor de dependências entre *checkpoints* apresenta uma ordenação seqüencial independente do tempo lógico de cada processo. Evidentemente, existe uma relação direta entre o LVT e os *checkpoints* dos processos da simulação. Entretanto, esta separação dos relógios facilita a utilização de algoritmos de *checkpoints* semi-síncronos já especificados, sem a necessidade de adaptar esses métodos ao protocolo de sincronização do programa de Simulação. Assim, observa-se que, a princípio, poderia ser utilizado duas abordagens: adoção de um único relógio lógico, para o tratamento do tempo lógico da simulação e da consistência para a obtenção dos *checkpoints*, ou a utilização de dois relógios distintos. Cada uma dessas abordagens apresenta vantagens e desvantagens. A utilização de dois relógios torna o algoritmo para obtenção dos *checkpoints* globais consistentes mais simples, visto que sua aplicação é direta. Desta forma, optou-se por essa abordagem devido à essa relativa simplicidade, uma vez que diversos outros aspectos deveriam ser tratados para se utilizar um único relógio lógico.

O próximo capítulo apresenta uma especificação para implementação do protocolo *Rollback* Solidário, com base na UML (*Unified Modeling Language*). Os diagramas da

UML permitem visualizar detalhadamente o funcionamento do protocolo proposto, fornecendo um roteiro para a implementação da camada de "Protocolo de Simulação" da arquitura para ambientes de simulação distribuída.

Capítulo 5

Especificação para Implementação do Protocolo *Rollback* Solidário

O protocolo *Rollback* Solidário é uma nova opção de protocolo para o usuário da simulação distribuída. O capítulo anterior descreveu o comportamento do protocolo, além de expor alternativas nas soluções de alguns problemas que, tradicionalmente, surgem neste tipo de aplicação. Este capítulo apresenta o projeto para sua implementação com base na UML (*Unified Modeling Language*), que é uma linguagem padrão para a elaboração da estrutura de projetos de *software* (BOOCH; RUMBAUGH; JACOBSON, 2000). A UML é adequada para a modelagem de sistemas, pois é uma linguagem muito expressiva, abrangendo as visões necessárias ao desenvolvimento e implantação desses sistemas.

O capítulo anterior apresentou uma arquitetura em camadas separando o projeto de um ambiente de simulação distribuída em três níveis. O objetivo deste capítulo é apresentar a modelagem para o projeto da segunda camada, conforme ilustra a figura 5.1, considerando a abordagem semi-síncrona para a obtenção dos *checkpoints*. A especificação apresentada neste capítulo demonstra a viabilidade de implementação do protocolo proposto.

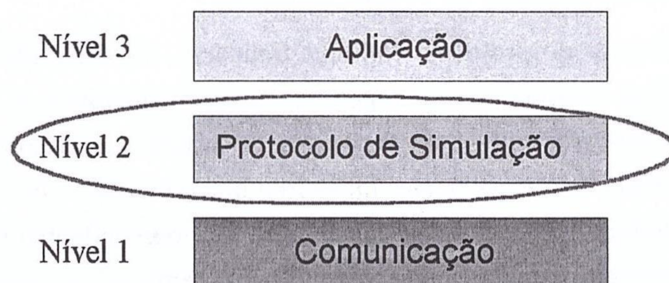

Figura 5.1: Camada onde se encontra o protocolo *Rollback* Solidário

A UML é uma linguagem padrão para a elaboração da estrutura de projetos de

software. Ela inclui nove diagramas: *classes, objetos, casos de uso, seqüência, colaborações, estados, atividades, componentes* e *implantação*. Essa não é uma lista completa de diagramas. Algumas ferramentas podem usar a UML para fornecer outros tipos, apesar de que, certamente, estes nove diagramas são os mais encontrados na prática (BOOCH; RUMBAUGH; JACOBSON, 2000).

Na maioria das situações não há necessidade de se utilizar todos os diagramas para modelar um sistema orientado a objetos. Alguns diagramas possuem funcionalidades semelhantes e podem ser usados para representar a mesma estrutura de software, como por exemplo, diagramas de classes e objetos, diagramas de seqüência e colaboração, etc.

Para descrição do projeto serão utilizados três diagramas: *classes, seqüências* e *estados*. Os diagramas de classes exibem um conjunto de classes, interfaces e colaborações, bem como seus relacionamentos. Os diagramas de seqüências modelam as interações dos objetos dando ênfase à ordenação temporal das mensagens que são trocadas entre os objetos, ou seja, na seqüência das chamadas aos métodos das classes. Os diagramas de gráficos de estados exibem uma máquina de estados, formada por estados, transições, eventos e atividades. Estes últimos abrangem a visão dinâmica de um sistema.

Ressalta-se o fato de que o diagrama de atividades foi utilizado, no capítulo anterior, para descrever o comportamento dos processos lógicos nas diversas situações tratadas pelo protocolo *Rollback* Solidário.

5.1 O Diagrama de Classes

As classes são os blocos de construção mais importantes de qualquer sistema orientado a objetos. A classe é uma abstração de um conjunto de objetos que possuem os mesmos tipos de características e comportamentos. Em outras palavras, uma classe é a matriz pela qual os objetos são criados e descreve um molde que especifica as propriedades e o comportamento para um conjunto de objetos similares.

O diagrama de classes do protocolo *Rollback* Solidário, representado pela figura 5.2, apresenta as principais classes que o compõe. A classe `Processo` é a base do modelo e trata-se de uma classe abstrata, uma vez que a declaração do método `executar()` é apenas um compromisso de que as classes filhas irão implementá-lo. As classes concretas `ProcessoSemObservador` e `ProcessoComObservador` são descendentes diretas da classe `Processo` e implementam o método `executar()`. Em especial, a classe `ProcessoSemObservador` possui o método `procederRollback()`, uma vez que, na abordagem sem observador, cada processo é responsável pelo procedimento de *rollback*.

A classe `Observador` possui a estrutura necessária para implementar o processo obser-

vador da abordagem que utiliza este tipo de processo. Destacam-se os métodos `extrair-LinhaDeRecuperação()`, `obterLRecuperação()` e `procederRollback()` que implementam as atividades que foram descritas no capítulo anterior.

Ambas as classes, `Processo` e `Observador`, possuem o método `ObterMensagem()` para facilitar a interação com o objeto da classe `Receptor`.

Além da classe `Processo`, a classe `Checkpoint` também é uma classe abstrata. Isto se deve ao fato de que diferentes algoritmos semi-síncronos para *checkpointing* podem ser utilizados. Para cada algoritmo possível, deve-se criar uma nova classe descendente de `Checkpoint` e implementar o algoritmo através dos métodos `checkpointing()`, `aoChegarMensagem()` e `aoEnviarMensagem()` que representam, respectivamente, o método para salvar os estados dos processos durante o procedimento de *checkpoint* e os métodos para atualizar o vetor de dependências e realizar os *checkpoints* forçados durante a troca de mensagens entre os processos lógicos do programa de simulação.

A classe `Estado` representa os atributos de cada processo da simulação. A separação destes atributos da classe `Processo` é devido à necessidade de armazenar os estados durante o procedimento de *checkpointing*. Além disso, esta classe pode ser remodelada de acordo com a necessidade do usuário sem grandes impactos no sistema, o que é sempre desejável em qualquer sistema orientado a objetos. A principal característica desta classe é a existência dos métodos `serializar()` e `desserializar()` que são responsáveis pela conversão dos atributos dos objetos da classe em uma cadeia de caracteres e vice-versa. Isto facilita o procedimento de *checkpointing* e a restauração dos respectivos estados durante um *rollback*.

As classes `Lista` e `Data` são classes que implementam as estruturas de dados necessárias para o gerenciamento das listas e filas de eventos futuros da simulação. Várias classes do modelo utilizam a estrutura de dados lista encadeada, como por exemplo, as classes `Checkpoint`, com o atributo `listaStrings`, e `Observador`, com o atributo `linhasRecuperação`. Por conseguinte, a classe `Data` é uma classe-template, que é um elemento parametrizado. Em linguagens como C++ e Ada, é possível escrever classes-template, cada uma definindo uma família de classes. O resultado da instanciação de uma classe-template é uma classe concreta que pode ser empregada da mesma forma que outras classes comuns. Os objetos da classe `Data` deverão ser instanciados de acordo com o tipo de dados que a classe `Lista` irá manipular.

Definição 5.1.1 (Classe Ativa) *Classes ativas são classes que representam comportamentos concorrentes do mundo real, utilizadas para criar um modelo que use os recursos do sistema o mais eficientemente possível.*

O diagrama de classes apresenta duas classes ativas: `Emissor` e `Receptor`. Estas

classes são responsáveis pela comunicação com as rotinas do nível 1 da arquitetura em camadas do ambiente de simulação, ou seja, os objetos desta classe são responsáveis pelo tratamento das mensagens que são enviadas e recebidas pelos processos lógicos do sistema. A existência destas classes especiais permite a implementação de linhas de controle (*threads*) independentes para o tratamento das mensagens do sistema, permitindo um desempenho superior da simulação. Além disso, a classe `Receptor` é modelada como classe amiga (<<`Friend`>>) das classes `ProcessoSemObservador` e `ProcessoComObservador` para facilitar a comparação dos tempos lógicos das mensagens com o estado do objeto.

Na modelagem, a comunicação entre objetos ativos é descrita utilizando eventos, sinais e mensagens. Um evento é algo que ocorre no sistema ou no ambiente, como a chegada de uma mensagem na rede de comunicação. Os sinais são um caso especial de eventos nomeados que podem ser suspensos. Na classe `Receptor`, o sinal `ChegadaDeMensagem` avisa a chegada de uma mensagem na rede de comunicação.

As duas classes ativas se relacionam com a classe `Message` que contém a estrutura da mensagem de comunicação. Os atributos desta classe foram definidos no capítulo anterior.

Finalmente, a classe `Ambiente` mantém informações a respeito das estações do ambiente de simulação. A existência desta classe permite estender o protocolo para possibilitar o balanceamento de carga e a migração dinâmica de processos.

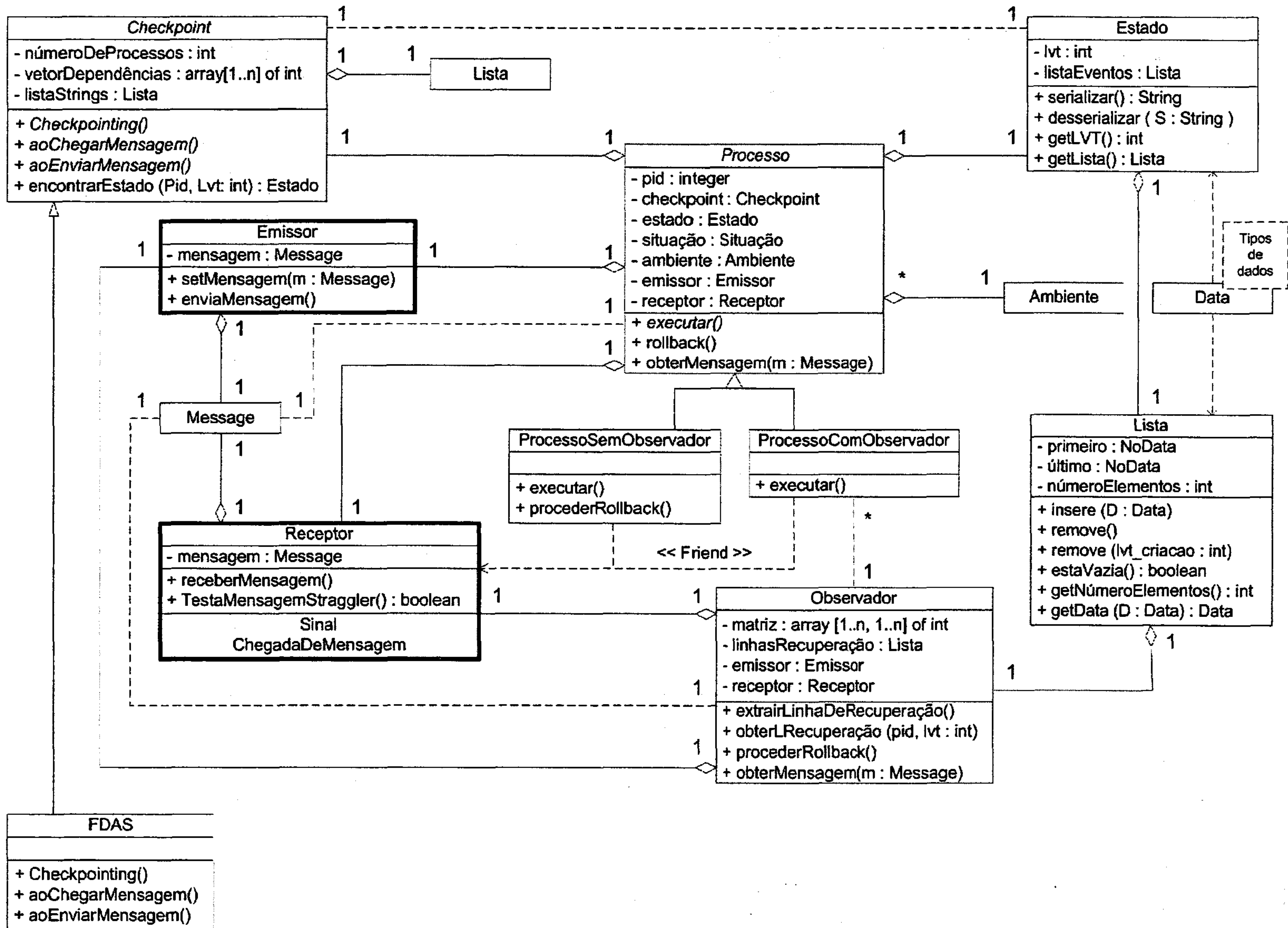

Figura 5.2: Diagrama de classes do protocolo *Rollback* Solidário

5.2 Diagramas de Seqüência

O diagrama de classes permite uma visão estática do sistema. Enxergar um sistema apenas pelo seu lado estático é um problema, pois as estruturas de dados e as funções que devem ser executadas pelo sistema são uma pequena parte do modelo e do próprio sistema.

Representar interações é permitir que se saiba como um determinado conjunto de operações permitirá que seja realmente executado o que se deseja do sistema. É muito improvável que apenas a constatação empírica de que haja estruturas de dados suficientes seja razoável para se confirmar a exatidão na execução de tais operações (MATOS, 2002).

Sem a representação de interações, a modelagem dos sistemas torna-se incompleta, haja vista que não se conhece como uma tarefa está sendo executada e nem como um objeto dispara e finaliza uma tarefa. Na UML, os diagramas de Interação são divididos em Diagramas de Colaboração e Diagramas de Seqüência. Os diagramas de Interação podem seguir dois enfoques distintos:

- baseando-se na ordem temporal das mensagens;
- baseando-se no contexto e na familiaridade de um conjunto de objetos

Particularmente, a ordem temporal é representada pelo Diagrama de Seqüência e o outro enfoque é representado pelo Diagrama de Colaboração.

Na modelagem do protocolo *Rollback* Solidário, será utilizado apenas o Diagrama de Seqüência, pois deseja-se demonstrar o comportamento das mensagens enviadas aos objetos das classes, definidas no diagrama de classes, e a seqüência dessas chamadas.

O diagrama de classes contemplou as duas possibilidades de implementação do protocolo *Rollback* Solidário em um mesmo diagrama, considerando a abordagem semi-síncrona para obtenção dos *checkpoints*, ou seja, *Rollback* Solidário sem Observador e *Rollback* Solidário com Observador. Para o diagrama de seqüência, tal objetivo não é possível de ser alcançado devido às diferenças entre as mensagens enviadas aos objetos.

5.2.1 *Rollback* Solidário sem Observador

Para facilitar o entendimento do diagrama de seqüência ele será subdivido em três partes. A figura 5.3 representa a primeira parte. Este diagrama ilustra o comportamento dos objetos durante o envio e o recebimento das mensagens dos processos lógicos durante o processamento normal da simulação distribuída.

Quando um evento é criado no sistema e um processo lógico, que está representado no sistema através de um objeto `:Processo`, necessita enviar uma mensagem para outro processo, ele realiza uma chamada ao método `setMensagem()` do objeto `:Emissor` que transfere o pedido para a rotina responsável na camada de Comunicação. Durante o processamento desta rotina acontece a chamada ao método `aoEnviarMensagem()` do objeto `:Checkpoint` que, dependendo do algoritmo para *checkpointing*, irá avaliar a necessidade de se obter um *checkpoint* forçado. Quando a mensagem chega, o sinal `ChegadaDeMensagem` avisa o objeto `:Receptor` através da chamada ao método `rece-berMensagem()`. Este método testa se a mensagem fere a relação de causa e efeito. Por isso, a classe `Receptor` é uma classe amiga das classes `ProcessoSemObservador` e `Proces-soComObservador`. Se a mensagem for *straggler* será chamado o método `rollback()` do objeto `:Processo`, caso contrário será chamado o método `obterMensagem()`. Além disso, o método `receberMensagem()` envia a mensagem `aoChegarMensagem()` para o objeto `:Checkpoint` que irá verificar a necessidade de realizar o salvamento de estados de acordo com o algoritmo semi-síncrono para a obtenção dos *checkpoints*.

A segunda parte do diagrama de seqüência, representado pela figura 5.4, apresenta a seqüência de ações que são realizadas durante a ocorrência de um *checkpoint* básico. O método `checkpointing()` realiza a contagem dos eventos para identificar quando foi concluído o intervalo entre os *checkpoints*, realizando os *checkpoints* básicos, neste caso.

A última parte do diagrama de seqüência representa a seqüência de ações durante a ocorrência de um *rollback* (figura 5.5). Neste diagrama, o processo que recebeu a mensagem *straggler* identifica o *checkpoint* de retorno através da chamada ao método `encontrarEstado()` do objeto `:Checkpoint` que, por sua vez, identifica o estado do processo através da chamada `getData()` do objeto `:Lista`. Os valores dos atributos do estado do processo são reconstituídos através da chamada ao método `desserializar()` do objeto `:Estado`. Em seguida, o objeto `:Processo`, que representa o processo lógico que recebeu a mensagem *straggler*, envia uma mensagem para os demais processos lógicos avisando da ocorrência do *rollback*. Esta ação é realizada através da chamada ao método `setMensagem()`.

Quando o procedimento de *rollback* está encerrado, o objeto `:Processo` retorna às atividades normais da simulação através do método `executar()`.

É importante considerar que os diagramas de seqüência apresentados contemplam a troca de mensagens entre os processos lógicos da simulação distribuída. Esta estrutura semântica não faz parte da estrutura padrão do diagrama de seqüência, mas permite visualizar a invocação de métodos remotos, através das mensagens do sistema.

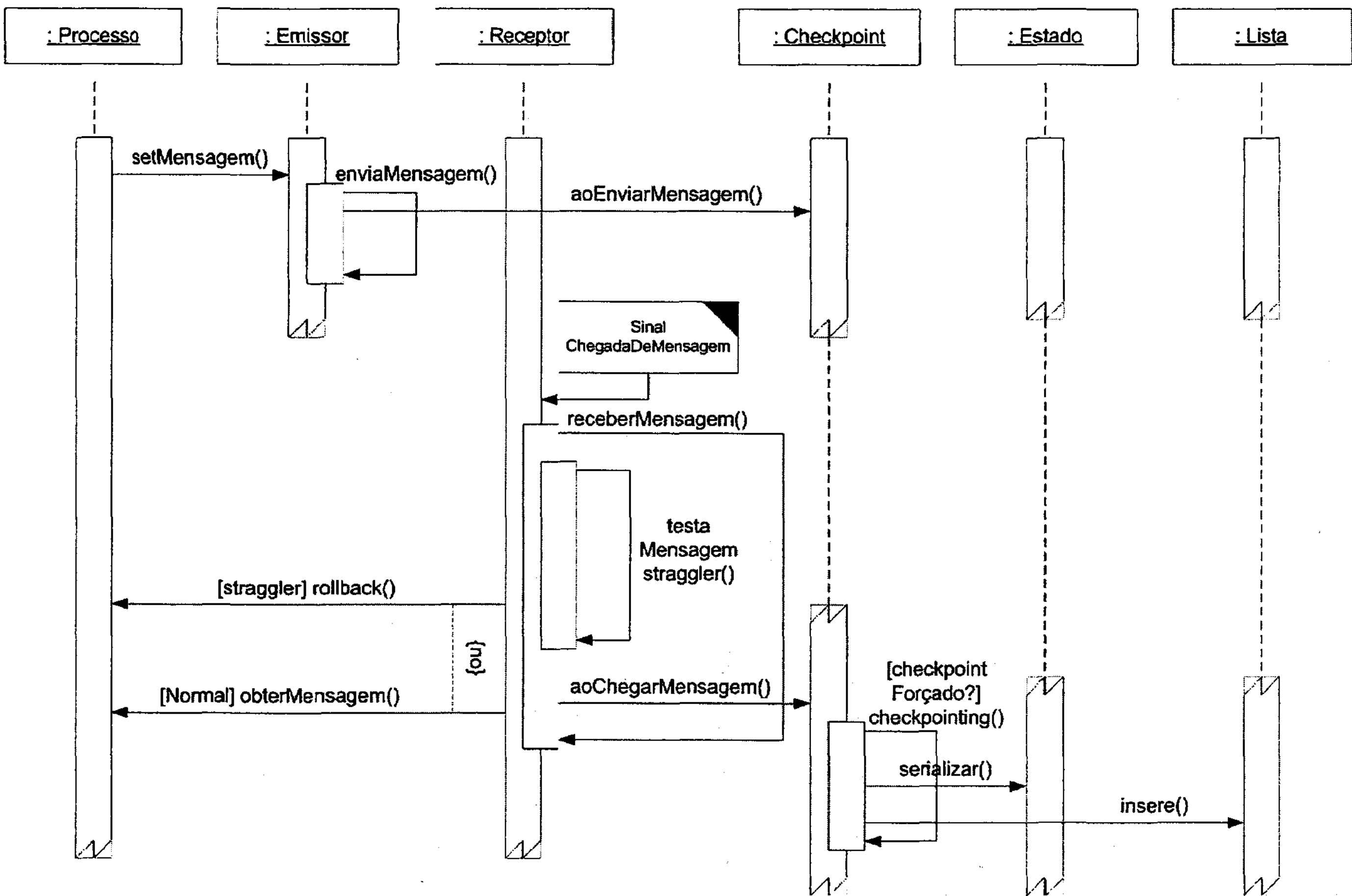

Figura 5.3: Diagrama de seqüência do protocolo *Rollback* Solidário

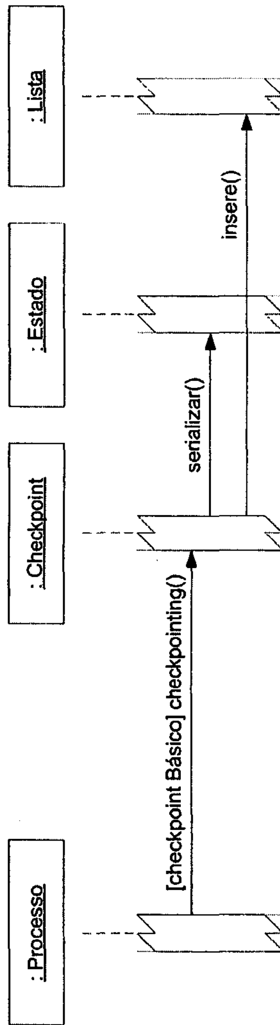

Figura 5.4: Diagrama de seqüência durante um *checkpoint* básico

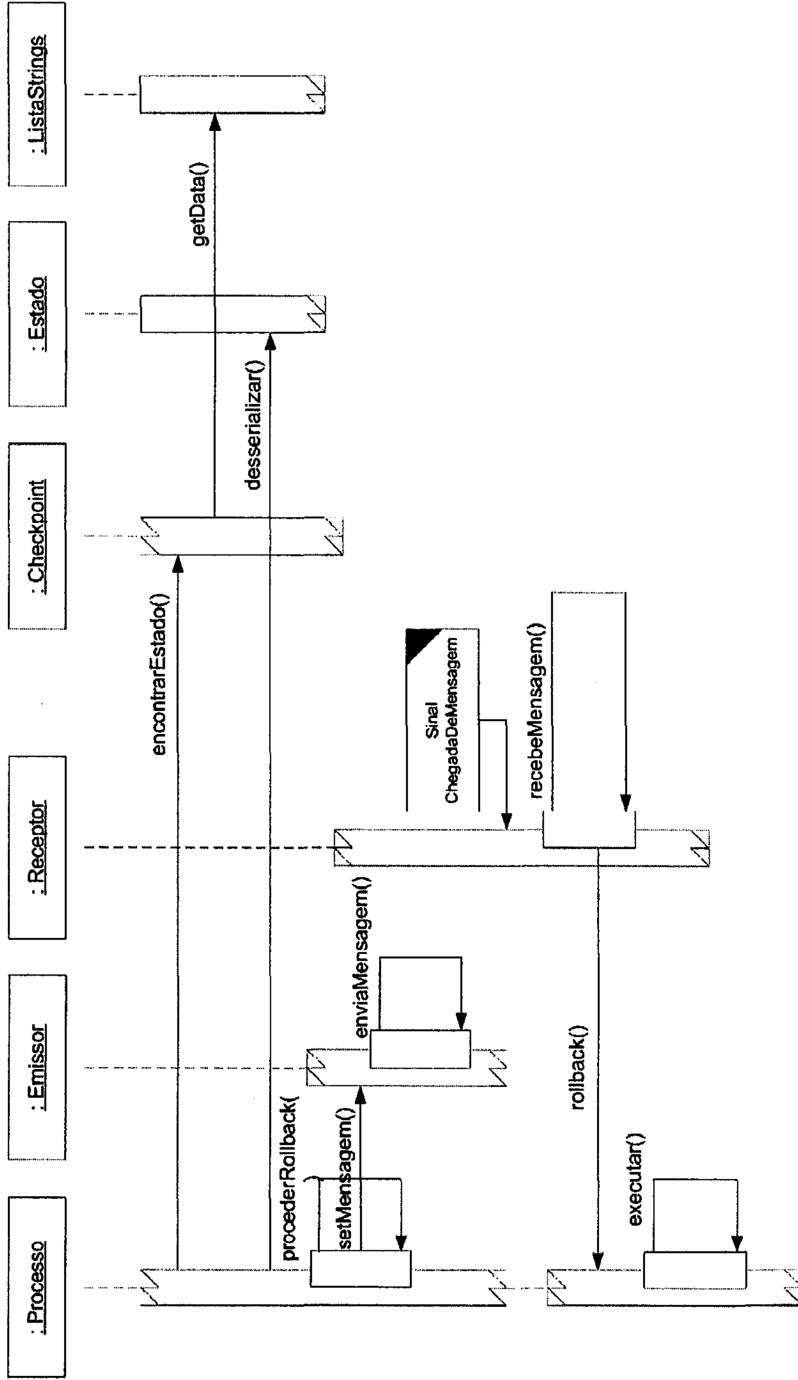

Figura 5.5: Diagrama de seqüência durante um *rollback* no protocolo *Rollback* Solidário

5.2.2 *Rollback* Solidário com Observador

Os diagramas de seqüência para abordagem com observador são semelhantes aos diagramas anteriores. As diferenças acontecem durante um *checkpoint*, que precisa ser notificado ao processo Observador, e quando ocorre um *rollback*. Portanto, somente a segunda e terceira partes do diagrama de seqüência serão novamente descritos.

Durante um *checkpoint* básico ou forçado, o processo lógico deve avisar o processo observador, através de uma mensagem. Quando o sinal de uma nova mensagem chega ao objeto `:Receptor` do processo observador, ele imediatamente realiza uma chamada ao método `extrairLinhaDeRecuperação()` do objeto `:Observador`, que irá atualizar o atributo `matriz` e verificar se uma nova linha de recuperação pode ser identificada. Neste caso, esta linha de recuperação será inserida na lista `linhasRecuperação`. O diagrama 5.6 ilustra esta seqüência.

Quando um processo lógico recebe a noticificação da chegada de uma mensagem *straggler*, imediatamente após a identificação e restauração do estado para onde o processo deve retornar, ele envia uma mensagem ao processo observador que, por sua vez, irá realizar uma chamada ao seu método `procederRollback()` (figura 5.7). Este método irá realizar os procedimentos descritos no capítulo anterior.

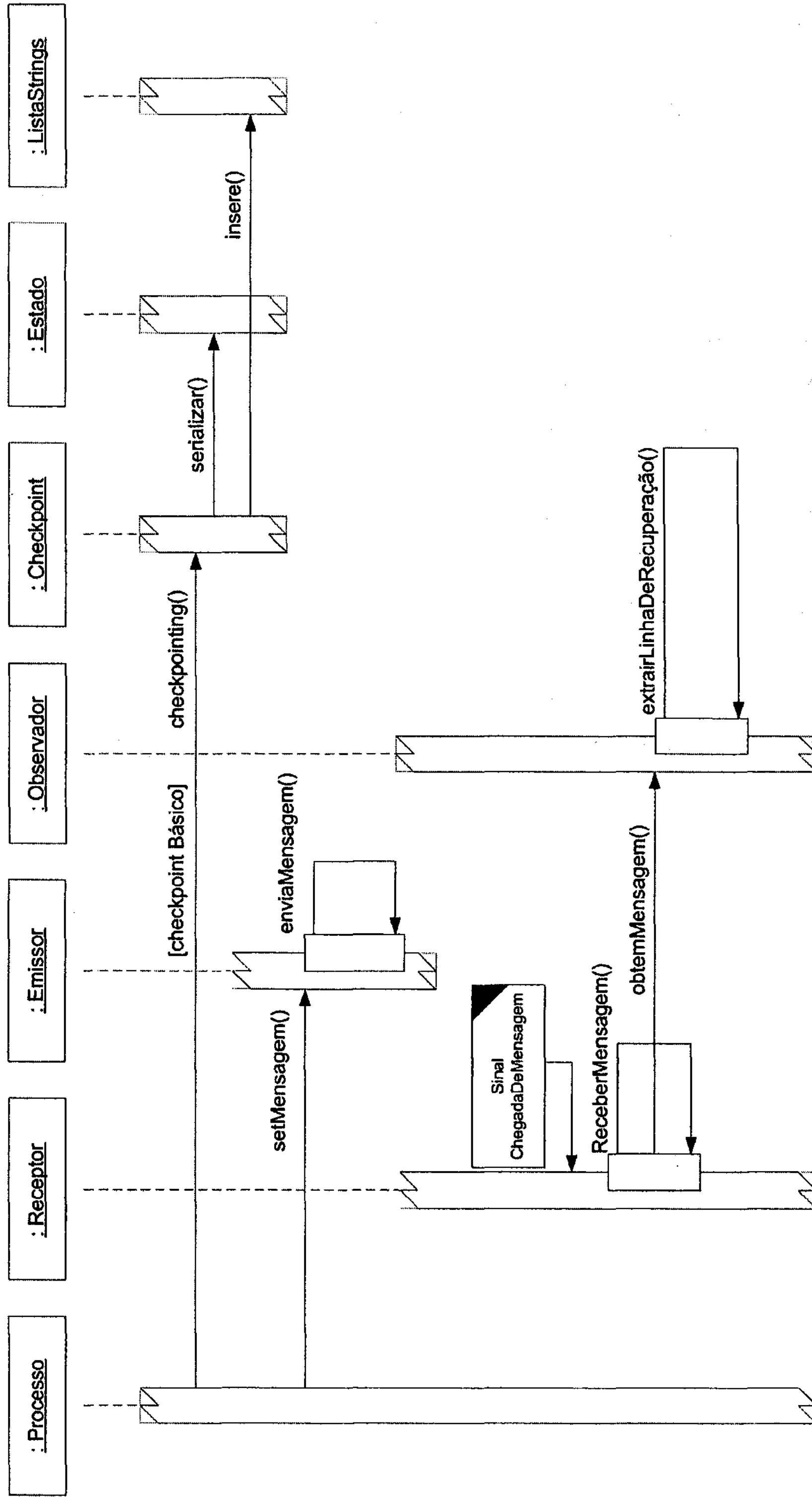

Figura 5.6: Diagrama de seqüência durante um *checkpoint* básico na abordagem com observador

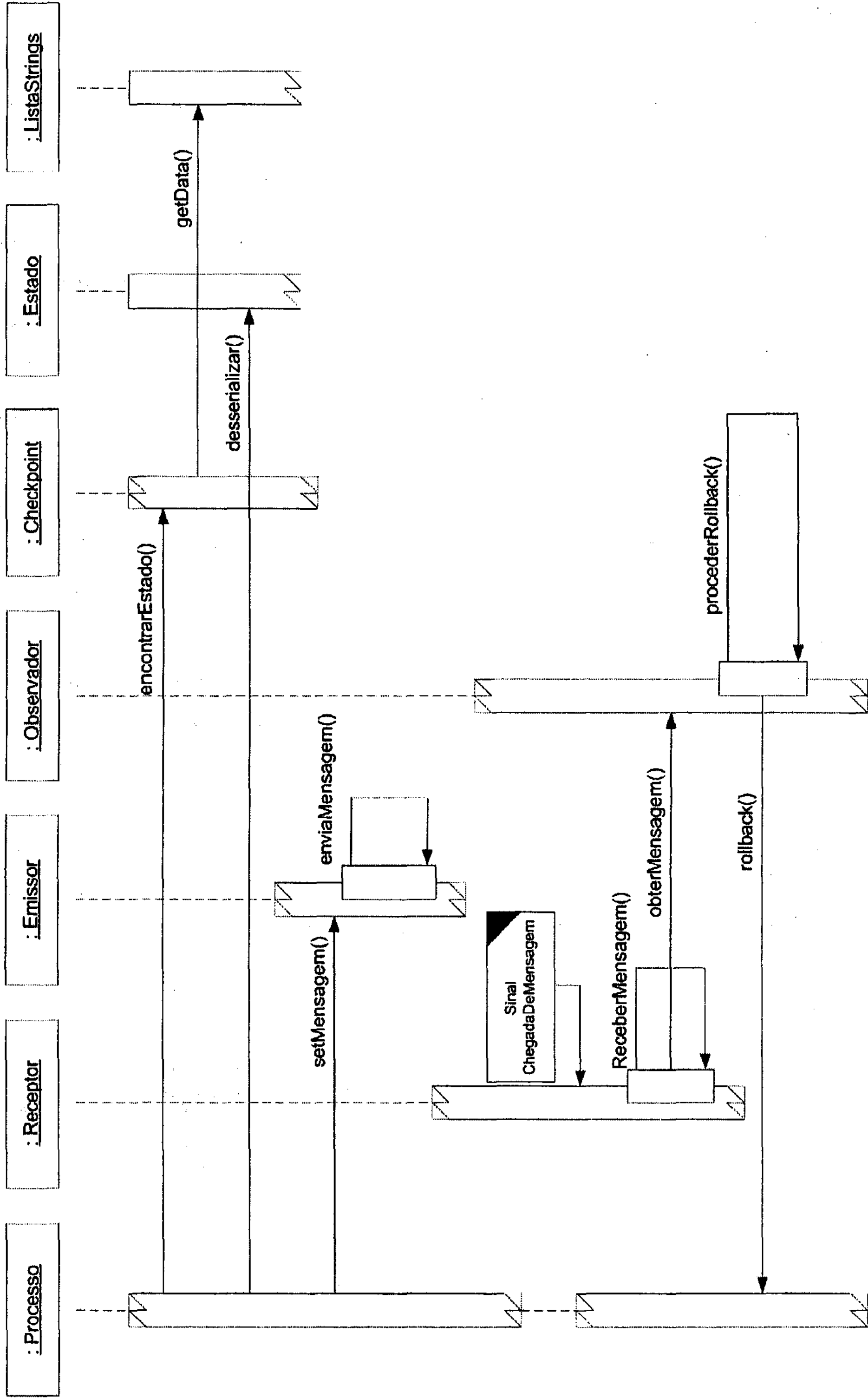

Figura 5.7: Diagrama de seqüência durante um *rollback* da abordagem com observador

5.3 Diagramas de Estados

Diagramas de estados são usados para descrever o comportamento de um sistema. Eles descrevem todos os estados possíveis em que um objeto particular pode estar e como o estado do objeto muda como resultado de eventos que o atingem.

Na maioria das técnicas de orientação a objetos, os diagramas de estados são projetados para uma classe única visando mostrar o comportamento ao longo do tempo de vida de um único objeto. Um estado é uma condição ou situação na vida de um objeto durante a qual o objeto satisfaz alguma condição, realiza alguma atividade ou aguarda um evento.

Existem várias formas de diagramas de estados, cada uma com uma pequena diferença semântica. O estilo UML é baseado no *statechart* de Harel (1987).

Na modelagem do protocolo *Rollback* Solidário, somente serão considerados os estados das classes `Processo`, `Emissor`, `Receptor` e `Observador`, pois tratam-se das principais classes do sistema, sendo, também, aquelas que dão a precisa idéia do comportamento do protocolo.

O diagrama de estados da classe `Processo` possui quatro estados (figura 5.8). Na fase inicial (estado **Iniciando**), o objeto `:Processo` deve instanciar todos os objetos que o compõe. Quando o objeto estiver preparado para executar as funções da simulação haverá a transição para o estado **Aguardando**. A partir deste instante, os objetos das classes `Emissor` e `Receptor` estarão trabalhando em concorrência e cooperação com o objeto `:Processo`. O estado **Aguardando** é um estado transitório e o objeto só mudará de estado novamente quando existir eventos para serem processados.

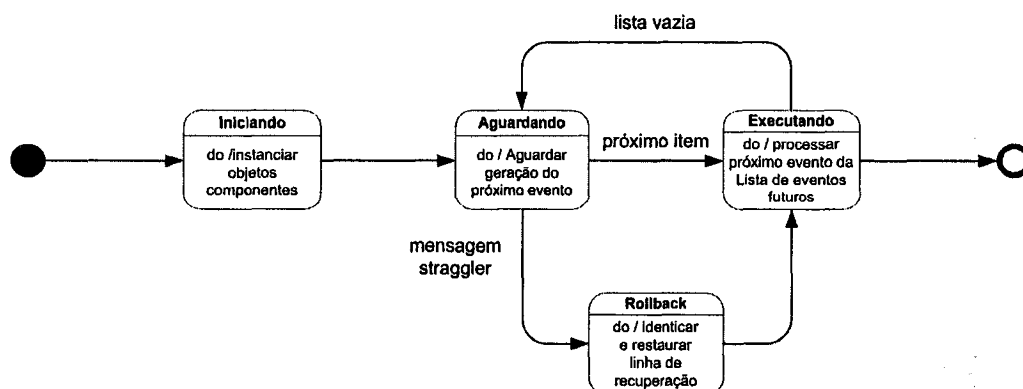

Figura 5.8: Diagrama de estados da classe Processo

Os primeiros eventos também poderão ser criados no estado **Iniciando**. Neste caso, a mudança do estado **Aguardando** para o estado **Executando** é automática devido à existência de eventos para serem escalonados.

Os objetos da classe `Receptor` podem estar em um dos quatro estados representados

pelo diagrama da figura 5.9. Os objetos ficam aguardando a chegada de uma nova mensagem pela rede de comunicação para mudar de estado (**Aguardando**→ **Tratando**). No estado **Tratando** é identificada a situação da mensagem recebida, normal ou *straggler*, para identificar a nova transição de estado. Quando uma mensagem *straggler* chega ao sistema, ocorre a mudança de estados no objeto `:Receptor` que, por sua vez, provoca a mudança do estado do objeto `:Processo` para **Rollback**, quando o sistema irá tratar da recuperação da computação.

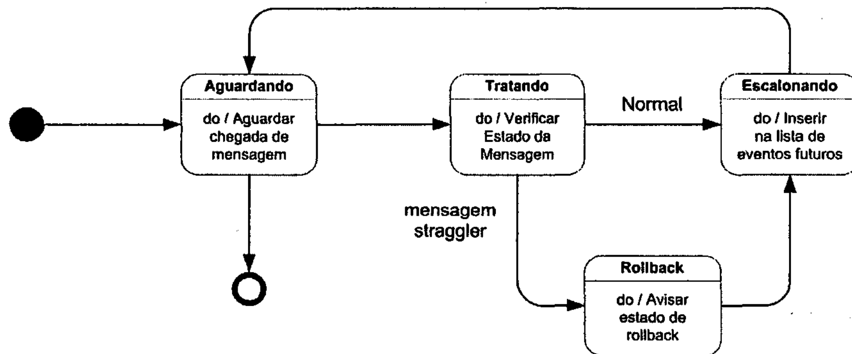

Figura 5.9: Diagrama de Estados da classe Receptor

O diagrama de estados do objeto `:Emissor` (figura 5.10) é mais simples, pois contém apenas dois estados: **Aguardando** e **Enviando**, ao contrário do objeto `:Receptor` que precisar tratar a mensagem que chega pela rede de comunicação.

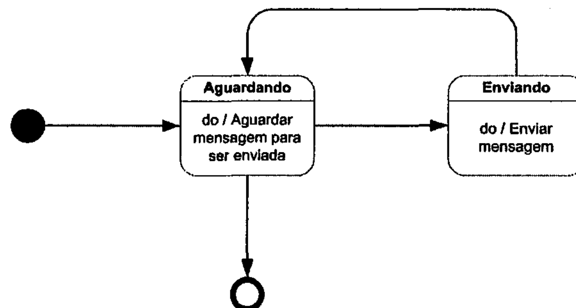

Figura 5.10: Diagrama de Estados da classe Emissor

O diagrama de estados do objeto `:Observador` é semelhante ao diagrama de estados do objeto `:Processo` (figura 5.11). A maior parte do tempo o objeto se encontra **Aguardando** a chegada de mensagens. As mensagens que apenas sinalizam a ocorrência de um *checkpoint* alteram o estado do objeto para **Controle**. Neste estado, o objeto atualiza a matriz de dependências e verifica se é possível extrair uma nova linha de recuperação. Após o tratamento do novo *checkpoint* o objeto retorna para o estado **Aguardando**. Quando uma mensagem de aviso de *rollback* chega, o estado é alterado para **Rollback** e são realizadas as ações necessárias para recompor a simulação. Quando o *rollback* estiver terminado, o objeto retorna para o estado de **Aguardando**.

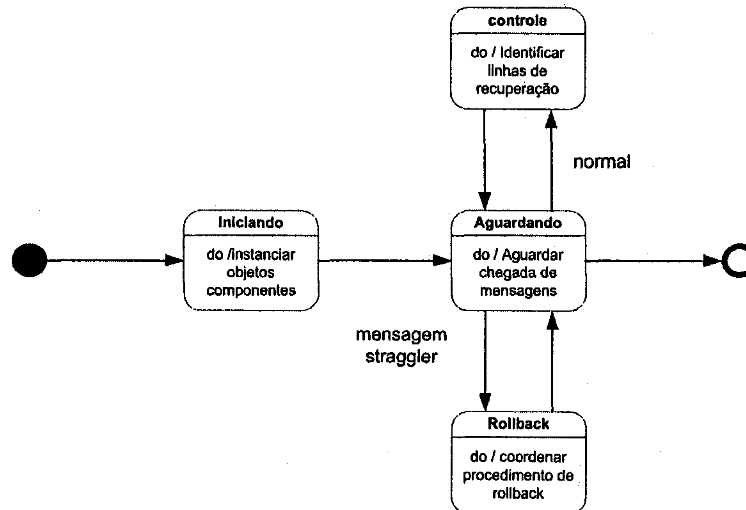

Figura 5.11: Diagrama de Estados da classe Observador

5.4 Considerações Finais

Os diagramas apresentados neste capítulo dão uma visão detalhada do funcionamento do protocolo proposto e, juntamente com os diagramas de atividades do capítulo anterior e os algoritmos do apêndice A, fornecem um roteiro para a implementação da camada de "Protocolo de Simulação" da arquitura para ambientes de simulação distribuída.

O próximo capítulo apresenta uma análise comparativa entre os protocolos *Rollback* Solidário e *Time Warp*.

Capítulo 6

Análise Comparativa

Este capítulo apresenta um estudo comparativo entre os protocolos *Rollback* Solidário e *Time Warp*. Este estudo tem por objetivo avaliar o comportamento do protocolo *Rollback* Solidário com relação aos critérios desempenho e utilização mais adequada da memória. Para realizar este estudo, o protocolo *Time Warp* será apresentado na variação *Sparse State Saving*, devido à semelhança no procedimento de armazenagem dos estados. Entretanto, como existem estudos comparativos entre os mecanismos de salvamento de estados para o protocolo *Time Warp*, através dos dados obtidos, pode-se prever como o protocolo *Rollback* Solidário se comportará em relação aos demais métodos de salvamento de estados (RöNNGREN et al., 1996). Outros estudos comparam os protocolos conservativos, otimistas e simulação seqüencial, além do custo da comunicação entre os processos (LIN, 1992; CAROTHERS; FUJIMOTO, 1994; SRINIVASAN; REYNOLDS, 1995; CORTELLESSA; QUAGLIA, 1998; QUAGLIA; CORTELLESSA; CICIANI, 1999; NUTARO; SARJOUGHIAN, 2001; SANTORO; QUAGLIA, 2001).

6.1 Comparação de Desempenho

O objetivo principal da simulação distribuída é diminuir o tempo de execução de programas de simulação. O *speedup* é o aumento de velocidade observado quando se executa um determinado programa em p processadores em relação à execução deste mesmo programa em apenas um processador. O ideal seria que o valor do *speedup* tendesse a p, porém, segundo Almasi (1994) e Quinn (1987), vários fatores dificultam alcançar esse índice como, por exemplo, a sobrecarga da comunicação entre os processadores, partes do código executável estritamente seqüenciais e o nível de paralelismo utilizado em virtude do uso de granulação inadequada à arquitetura.

Nesta análise, o fator principal para comparação do desempenho entre os protocolos *Time Warp* e *Rollback* Solidário será o tempo desperdiçado durante a simulação, que

corresponde ao período necessário para computar os eventos desfeitos durante os *rollbacks* dos protocolos otimistas ou ao tempo em que um processo fica bloqueado nos protocolos conservativos.

O número de eventos desfeitos em um *rollback*, assim como o número de *rollbacks* que ocorrem durante a simulação, está intimamente relacionado com o modelo que o programa de simulação está representando, a velocidade dos processadores, o balanceamento da carga entre eles, a interação entre os processos, etc. Portanto, encontrar um modelo matemático, de âmbito geral, que represente o desperdício de processamento em um programa de simulação não é trivial. Por conseguinte, será considerado que o número de eventos desfeitos durante um *rollback* segue uma distribuição uniforme e que, após o envio de uma mensagem, a probabilidade de ocorrer *rollbacks* em cascata, envolvendo k processos, segue uma distribuição binomial, conforme as definições 6.1.1 e 6.1.2, respectivamente.

Definição 6.1.1 (Distribuição Uniforme) *Uma distribuição uniforme é uma distribuição de probabilidade em que todos os valores da variável aleatória são igualmente prováveis.*

Definição 6.1.2 (Distribuição Binomial) *Distribuição binomial é aquela em que os termos da expansão do binômio (ou multinômio) correspondem às probabilidades de todos os eventos possíveis do espaço amostral. O binômio (ou multinômio) é formado pelas probabilidades de cada acontecimento elevado ao número total de ocorrências.*

Os experimentos binomiais têm a característica de apresentarem exatamente dois resultados complementares e satisfazem as seguintes condições (TRIOLA, 1999; MONTGOMERY; RUNGER, 2003):

1. O experimento deve comportar um número fixo de provas.
2. As provas devem ser independentes (O resultado de qualquer prova não afeta as probabilidades das outras provas).
3. Cada prova deve ter todos os resultados classificados em duas categorias.
4. A probabilidade de sucesso que, geralmente é denotada por p, é a mesma em cada ensaio. A probabilidade de falha será denotada por $1 - p$.

Para um experimento que consiste na realização de n ensaios de Bernoulli, o espaço amostral pode ser considerado como o conjunto das n-uplas de comprimento n, em que cada posição há um sucesso (S) ou uma falha (F) (DANTAS, 2000).

Pelas condições 2) e 4) vê-se que a probabilidade de um ponto amostral com sucessos nos k primeiros ensaios e falhas nos $n-k$ ensaios seguintes é $p^k(1-p)^{n-k}$.

Observa-se que o evento $[X = k]$ ocorre se for observado um ponto amostral que tenha k sucessos e $n - k$ falhas. O número de pontos do espaço amostral que satisfaz essa condição é igual ao número de maneiras com que é possível escolher k ensaios dentre os n para a ocorrência de sucesso, pois nos $n-k$ restantes deverão ocorrer falhas. Este número é igual ao número de combinações de n elementos tomados k a k, ou seja $\binom{n}{k}$.

Decorre do que foi exposto que, para $k = 0, 1, \ldots, n$:

$$P[X = k] = \binom{n}{k} p^k (1-p)^{n-k} \tag{6.1}$$

A fórmula 6.1 é denominada distribuição binomial com parâmetros n e p, onde n é o número de ensaios e p a probabilidade de sucesso em cada ensaio.

A palavra *sucesso*, tal como usada aqui, é arbitrária e não descreve necessariamente um resultado desejado. Qualquer uma das duas categorias possíveis pode ser chamada um sucesso (S), desde que a probabilidade correspondente seja identificada como p.

Na simulação distribuída, n é o número de processos do sistema que podem ser afetados por uma única mensagem *straggler* m e k é a quantidade de processos envolvidos no procedimento de um *rollback*, ou seja, k representa o número de *rollbacks* em cascata afetados pela mensagem m. Obviamente, $n-k$ é o número de falhas do experimento que, na realidade, representa a quantidade de processos que não realizaram *rollback*.

6.1.1 Desperdício de Processamento no *Time Warp*

A comparação que será apresentada tem como base a ocorrência de *rollbacks*. Supondo, portanto, que uma mensagem *straggler* m, no protocolo *Time Warp*, force um processo p_i a restaurar o seu estado α, que foi armazenado no tempo lógico t, então, como já exposto, p_i desfaz o processamento que foi realizado com tempo lógico maior do que t, restaurando o respectivo estado armazenado. Na situação mais simples, quando a ocorrência do *rollback* primário não provoca *rollbacks* secundários, o tempo necessário para recompor a simulação (t_r) é:

$$t_r = \overrightarrow{t}(m) + t(\alpha) \tag{6.2}$$

onde $\overrightarrow{t}(m)$ representa o tempo médio em que a mensagem m leva para chegar ao seu destino e $t(\alpha)$ é o tempo necessário para restaurar o estado α.

Então, se n_e é o número médio de eventos desfeitos durante os *rollbacks*, o tempo

médio desperdiçado pelo processo p_i, para restaurar o estado α, pode ser descrito pela equação:

$$td_{p_i} = t_r + n_e \cdot t(e) \tag{6.3}$$

com $t(e)$ representando o tempo gasto pelo processador para a execução do evento e, que, nesta análise, será tratado como um valor constante para o processamento de qualquer evento da simulação e, para simplificar as notações, a expressão $n_e \cdot t(e)$ será substituída pela variável λ nas próximas equações.

Se a ocorrência de um *rollback* primário no processo p_i provoca *rollback* em p_j, então, além do desperdício representado pela equação 6.2, que no processo p_j representa o tempo de deslocamento da respectiva anti-mensagem mais o tempo para restaurar o estado correspondente, o desperdício de tempo em p_j será acrescido do tempo td_{p_i}, pois durante a ocorrência da mensagem *stragger* em p_i, o processo p_j continuou a sua execução. Desta forma, o desperdício de processamento em p_j pode ser descrito pela equação:

$$td_{p_j} = t_r + td_{p_i} \tag{6.4}$$

Por indução, tem-se que, após a ocorrência de um *rollback* primário no processo p_i, se acontecerem n *rollbacks* secundários, e sendo p_n o último processo dessa série, então, o desperdício de processamento em p_n será:

$$td_{p_n} = n \cdot t_r + \lambda \tag{6.5}$$

É importante considerar que o número de eventos desfeitos em um *rollback* não é acumulativo, apesar do efeito cascata dos *rollbacks*, pois, no protocolo *Time Warp*, os *rollbacks* sempre ocorrem para o futuro e, para efeito de comparação, está sendo considerado uma distribuição uniforme com relação ao número médio de eventos perdidos durante um *rollback*.

A partir da equação 6.5, pode-se calcular o desperdício total de processamento provocado pela mensagem *straggler* m (Td_m) quando ocorrerem n *rollbacks*, sendo 1 primário e $n-1$ secundários. Nesta situação, Td_m é o somatório dos desperdícios de cada processo envolvido:

$$Td_m = \sum_{i=1}^{n} td_{p_i} \tag{6.6}$$

Vale lembrar que o valor td_{p_1} corresponde ao tempo desperdiçado pelo processo que

realizou o *rollback* primário. Desenvolvendo o somatório da equação 6.6, obtém-se a equação 6.7.

$$\begin{aligned} Td_m &= td_{p_1} + td_{p_2} + td_{p_3} + \ldots + td_{p_n} \\ &= t_r + \lambda + 2t_r + \lambda + 2t_r + \lambda + 3t_r + \lambda + \ldots + n \cdot t_r + \lambda \\ &= (t_r + 2t_r + 3t_r + \ldots + n \cdot t_r) + \underbrace{(\lambda + \lambda + \lambda + \ldots + \lambda)}_{n\ vezes} \\ &= t_r \cdot \left(\frac{n \cdot (n+1)}{2}\right) + n \cdot \lambda \end{aligned} \tag{6.7}$$

Conforme discussão anterior, a ocorrência de *rollbacks* em cascata é causada, principalmente, pelo relacionamento entre os processos do sistema. Entretanto, a dependência do modelo não está sendo considerada devido à generalidade desta análise e, portanto, a distribuição binomial será utilizada, considerando que, para cada processo do sistema, a probabilidade de ocorrer *rollback*, após uma mensagem *straggler*, é p e, para efeito de análise, os *rollbacks* não possuem dependência entre si. Conseqüentemente, a variável aleatória X representa a probabilidade de ocorrência de *rollbacks* após o envio de uma mensagem m no sistema. Assim, $X = 0$ implica na probabilidade de que m não provoque *rollback*, $X = 1$ é a probabilidade de m provocar apenas *rollback* primário, $X = 2$ é a probabilidade de ocorrer um *rollback* primário e um *rollback* secundário e assim por diante. Por conseguinte, o número total de *rollbacks* no sistema é dado em função da esperança matemática da variável X, que, na distribuição binomial, é definida pela equação 6.8.

$$\begin{aligned} E(X) &= \sum_{k=0}^{n} \binom{n}{k} p^k (1-p)^{n-k} \\ &= \sum_{k=0}^{n} \frac{kn!}{k!(n-k)!} p^k (1-p)^{n-k} \\ &= \sum_{k=1}^{n} \frac{n!}{(k-1)!(n-k)!} p^k (1-p)^{n-k} \\ &= np \sum_{j=0}^{n-1} \frac{(n-1)!}{j!(n-j-1)!} p^j (1-p)^{n-j-1} \\ &= np \sum_{j=0}^{n-1} \binom{n-1}{j} p^j (1-p)^{n-j-1} \\ &= np(p + (1-p))^{n-1} = np \end{aligned} \tag{6.8}$$

Na terceira igualdade foi utilizada a expressão $j = k - 1$ e a penúltima igualdade é

conseqüência da expansão do binômio $(p+(1-p))$ à potência $n-1$.

Conclui-se, então, que o desperdício total (Td_{TW}) de processamento durante a simulação, utilizando o protocolo *Time Warp* e com probabilidade p de ocorrer *rollbacks* a cada mensagem do sistema, em média, consiste em:

$$Td_{TW} = \left\{ t_r \cdot \left(\frac{np \cdot (np+1)}{2} \right) + np \cdot \lambda \right\} \cdot Msg \tag{6.9}$$

sendo Msg o número total de mensagens que ocorreram durante a simulação.

Esta análise mostra que a complexidade do protocolo *Time Warp* nos piores casos é da ordem de $O(n^2)$, ou seja, em situações onde a quantidade de processos envolvidos a cada *rollback* é grande, o desperdício de processamento chega a ser quadrático em função do número de processos envolvidos.

6.1.2 Desperdício de Processamento no *Rollback* Solidário sem observador

Admitindo-se que a mesma mensagem *straggler* m, que ocorreu no sistema com o protocolo *Time Warp*, surja em condições similares no protocolo *Rollback* Solidário, utilizando o mecanismo sem observador, forçando o processo p_i a restaurar o seu estado α, como discutido no capítulo 4, o processo p_i irá restaurar o estado armazenado com tempo lógico imediatamente anterior ao LVT da mensagem *straggler* e enviará, para os demais processos da simulação, o respectivo vetor de dependências do *checkpoint* restaurado, através da mensagem `RealizaRollback`. O processo p_i ficará aguardando as mensagens de confirmação por parte dos demais processos.

Mesmo na situação em que a ocorrência do *rollback* primário não cause *rollbacks* secundários, haverá o envio da mensagem de *rollback* para todos os processos da simulação por p_i. Neste caso, supondo que a mensagem seja enviada em *broadcast*, o tempo total desperdiçado pelo protocolo *Rollback* Solidário para reconstituir a simulação após a mensagem *straggler* m, considerando apenas o *rollback* primário, é dado pela equação 6.10.

$$Td_{m1} = \overrightarrow{t}(m) + t(\alpha) + 2 \cdot \overrightarrow{t}(m') \tag{6.10}$$

Ressalta-se que $2 \cdot \overrightarrow{t}(m')$ representa o tempo médio para o envio da mensagem `RealizaRollback` com o vetor de dependências mais o tempo de retorno das mensagens de confirmação.

Se a ocorrência de um *rollback* primário no processo p_i provoca *rollback* em qualquer outro processo, o tempo de desperdício dos demais processos só tem relação com o prejuízo

outro processo, o tempo de desperdício dos demais processos só tem relação com o prejuízo do processo p_i, por conseguinte, o cálculo de td_{p_n}, para o protocolo *Rollback* Solidário, será o resultado da equação 6.11, qualquer que seja n, com $n > 1$.

$$td_{p_n} = 2t_r + \lambda \tag{6.11}$$

Como foi discutido no capítulo anterior, para encontrar a linha de recuperação máxima é preciso aumentar a comunicação entre os processos envolvidos. Mas o protocolo *Rollback* Solidário pode adotar o procedimento que encontra a linha mínima, pois, neste caso, o tratamento é dado apenas pelo processo responsável pelo *rollback*. Assim, a linha de recuperação não será a mesma do protocolo *Time Warp*, porém com o aumento da quantidade de *checkpoints* no sistema, o intervalo entre *checkpoints* diminui e, portanto, isto afeta a quantidade de eventos desfeitos durante o *rollback*, mas não prejudica a análise da complexidade do protocolo.

Da mesma forma como foi calculado para o protocolo *Time Warp*, o desperdício total de processamento provocado pela mensagem *straggler* m é a soma dos desperdícios de cada processo envolvido. A expansão deste somatório resulta na equação 6.12.

$$\begin{aligned} Td_m &= (t_r + \lambda) + \underbrace{(2t_r + \lambda) + \ldots + (2t_r + \lambda)}_{n-1 \ vezes} + Synch \\ &= t_r \cdot (2n - 1) + n \cdot \lambda + Synch \end{aligned} \tag{6.12}$$

O termo *Synch* foi inserido para registrar a existência das mensagens `ConfirmaRollback` e `ConfirmaContinuação`, visto que o processo que iniciou o procedimento de *rollback* fica aguardando as respostas dos demais processos implicando em atraso para o processo responsável. Entretanto, devido às distribuições usadas, *Synch* é um termo constante e possui pouco impacto no sistema quando o número de processos envolvidos no *rollback* é alto.

Novamente, supondo que a ocorrência de *rollbacks*, após o envio de uma mensagem, segue uma distribuição binomial, pode-se, então, deduzir que o desperdício total (Td_{SR}) de processamento durante a simulação, utilizando o protocolo *Rollback* Solidário, em média, consiste em:

$$Td_{SR} = \{t_r \cdot (2np - 1) + np \cdot \lambda\} \cdot Msg \tag{6.13}$$

É importante destacar que, apesar de propagar mais informações de controle em cada mensagem, o protocolo *Rollback* Solidário apresenta complexidade linear durante

a ocorrência dos *rollbacks* em cascata, ao contrário do protocolo *Time Warp* que possui complexidade quadrática.

Se todas as mensagens *straggler* provocarem apenas *rollbacks* primários ou envolverem poucos processos em *rollbacks* secundários, o protocolo *Time Warp* terá um desempenho melhor, uma vez que, no protocolo *Rollback* Solidário, o processo que recebe a mensagem *straggler*, após enviar a mensagem de *rollback* para os demais processos, deve aguardar a confirmação. Entretanto, a medida que o número de *rollbacks* secundários cresce o protocolo *Rollback* Solidário apresentará um desempenho superior.

6.1.3 Desperdício de Processamento no *Rollback* Solidário com observador

Conforme discutido no capítulo 4, a implementação do protocolo *Rollback* Solidário utilizando um processo observador, apresenta algumas vantagens, sendo a principal delas a eficiência na tomada de decisões durante os *rollbacks*. Além disso, o protocolo permite soluções mais simples para os problemas de sincronização que podem surgir.

Quando o número de *rollbacks* no sistema for pequeno, a utilização de um processo observador poderá diminuir o *speedup* da aplicação. Conforme a probabilidade de ocorrer *rollbacks* aumenta, a vantagem do uso deste tipo de processo passa a ser evidente. Entretanto, este tipo de processo aumenta a carga na rede de comunicação.

Em simulações onde a quantidade de *rollbacks* é relativamente baixa, os processos do protocolo *Rollback* Solidário podem implementar um mecanismo preguiçoso para avisar ao observador da ocorrência de novos *checkpoints*. Nesta situação, as mensagens sobre novos *checkpoints* poderão ser enviadas após uma certa quantidade q de *checkpoints*. Entretanto, esta possibilidade exige um procedimento de correção das linhas de recuperação quando os *rollbacks* ocorrem.

Um estudo mais aprofundado do impacto da utilização do processo observador poderá identificar valores adequados para a variável q. Em uma situação ideal, o valor de q pode ser definido em tempo de execução. Isto sugere um mecanismo dinâmico que pode ser desenvolvido para melhorar o desempenho do protocolo *Rollback* Solidário com observador.

6.1.4 Impacto dos *Checkpoints*

A análise realizada nas seções anteriores sugere que o impacto no desempenho, realizado pelo salvamento dos estados (*checkpointing*), é o mesmo nos dois protocolos. Entretanto, a realidade é um pouco diferente, pois, como o protocolo *Rollback* Solidário utiliza um mecanismo semi-síncrono para a obtenção dos estados, ele tende a forçar mais *checkpoints* do que o protocolo *Time Warp*. Essa diferença degrada o desempenho do protocolo e deve ser considerada na análise comparativa.

Para verificar o impacto dos *checkpoints* forçados na simulação é necessário encontrar a diferença entre a quantidade de *checkpoints* obtidos com a utilização de um algoritmo semi-síncrono e a quantidade de *checkpoints* básicos que seriam obtidos em um mecanismo totalmente assíncrono, como é o caso do protocolo *Time Warp*, utilizando a abordagem *Sparse State Saving*, para salvamento de estados. A equação 6.13, portanto, deve acrescentar o número de *checkpoints* a mais que são necessários para o funcionamento do protocolo *Rollback* Solidário, através de uma variável que o represente. A equação 6.14 adiciona a variável χ à equação 6.13.

$$Td_{SR} = \{t_r \cdot (2np - 1) + np \cdot \lambda\} \cdot Msg + \chi \tag{6.14}$$

A variável χ representa o tempo de processamento necessário para realizar os *checkpoints* que são produzidos a mais no protocolo *Rollback* Solidário. Finalmente, a partir das equações 6.9 e 6.14, é possível apresentar uma inequação que pode ser usada para verificar quando o protocolo *Rollback* Solidário supera o protocolo *Time Warp*:

$$\chi < \left(\frac{(np)^2 - 3np + 2}{2}\right) \cdot t_r \cdot Msg \tag{6.15}$$

Como estudado no Capítulo 3, os mecanismos para *checkpoints* semi-síncronos induzem a realização dos *checkpoints* com base nos padrões das comunicações realizadas pela aplicação. É evidente que, quando um processo realiza um *checkpoint* forçado, o intervalo entre os *checkpoints* diminui e, por conseguinte, o número de *checkpoints* na aplicação tende a ser maior. Desta forma, em uma comparação com o protocolo *Time Warp*, é importante analisar o impacto que a indução de *checkpoints* provoca no protocolo *Rollback* Solidário, uma vez que no *Time Warp* o mecanismo *Sparse Sate Saving* realiza os salvamentos de estados com base, apenas, nos intervalos entre os *checkpoints*, possuindo somente *checkpoints* básicos.

Para se obter a comparação entre os algoritmos de *checkpointing*, foi desenvolvida uma aplicação que permite simular o comportamento dos algoritmos sobre um diagrama de Espaço × Tempo. A ferramenta constrói o diagrama gerando aleatoriamente os eventos

que ele representa. É possível estender a ferramenta para utilizar um arquivo de *trace*, que pode ser obtido a partir de um sistema de computação distribuído real, para que sejam identificados, automaticamente, os parâmetros de configuração.

O Diagrama de Fluxo de Dados (DFD) da figura 6.1 demonstra, em alto nível, o comportamento da ferramenta. Inicialmente, o usuário configura os parâmetros que definem o número de eventos do diagrama de Espaço × Tempo, o número de processos e o intervalo entre os *checkpoints*. Esse intervalo pode ser definido para uma faixa de valores. Assim, a ferramenta simula o mesmo algoritmo para intervalos de *checkpoints* diferentes, sem que o usuário tenha que repetir o procedimento.

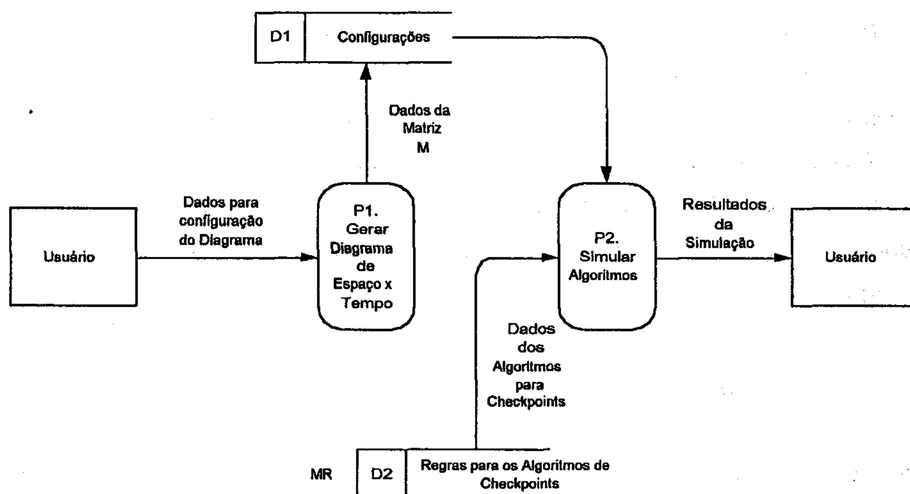

Figura 6.1: DFD representando a ferramenta para análise de algoritmos para *checkpointing*

A figura 6.2 apresenta a interface que permite configurar os dados para a criação do diagrama de Espaço × Tempo. É importante observar que o usuário deve fornecer os valores, em porcentagem, da probabilidade de ocorrer eventos internos, externos ou neutros. Os eventos externos representam o envio e o recebimento de uma mensagem. Ao escolher um evento de envio, o programa também sorteia um processo para o destino da mensagem. O sistema mantém uma fila FIFO (*First-In-First-Out*) para cada processo. Esta estrutura é utilizada para armazenar as mensagens recebidas. Quando o sistema escolhe um evento de *receive*, ele retira a primeira mensagem da fila. Se a fila estiver vazia, o evento de recebimento de mensagem é substituído por um evento interno. A ferramenta simula uma rede de comunicação completa, pois todos os processos podem enviar mensagens diretamente para qualquer outro processo. Não há opção para perda de mensagens que são entregues na mesma ordem de envio.

Com essas informações, o processo P1 (Gerar Diagrama de Espaço × Tempo) atribui valores iniciais a uma matriz M de ordem $n \times m$, onde n representa o número de processos e m representa o tempo lógico na computação distribuída. A figura 6.3 apresenta a matriz

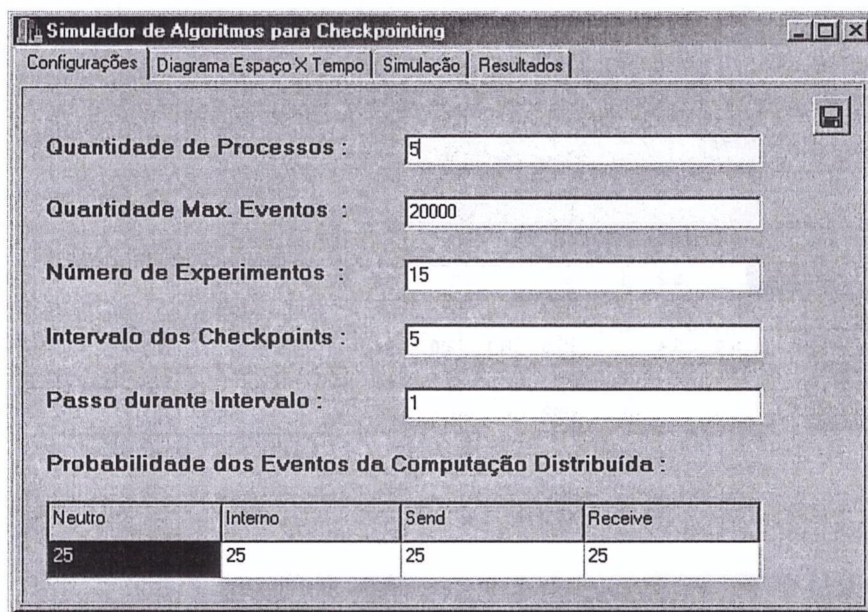

Figura 6.2: Interface da ferramenta apresentando a tela de configuração do diagrama de Espaço × Tempo

após o seu preenchimento pelo programa.

Cada elemento da matriz M é iniciado com um dos seguintes valores:

- `'I'`: Representando um evento interno da computação distribuída.
- `'S:`n`'`: Representando o envio de uma mensagem para o processo n.
- `'R:`n`'`: Representando o recebimento de uma mensagem do processo n.

A célula ainda pode receber um valor nulo, representando que um novo evento ainda não ocorreu no processo. Este é o valor neutro e significa, simplesmente, que o último evento ainda não foi concluído no respectivo processo.

Após a geração da matriz, todos os algoritmos de *checkpointing*, que possuem suas representações armazenadas (figura 6.4), podem ser simulados. A figura 6.5 apresenta a tela com os resultados da simulação.

Os testes realizados com a ferramenta foram divididos nas quatro classes representadas pela tabela 6.1. O objetivo desta definição é criar uma escala de situações que envolvam aplicações fortemente *CPU-Bound*, até aplicações que façam uso instenso de comunicação (*IO-Bound*).

Os experimentos foram realizados considerando as matrizes de 2 × 20.000, 4 × 20.000, 6 × 20.000 e 8 × 20.000, ou seja 2, 4, 6 e 8 processos, com cada processo realizando, no

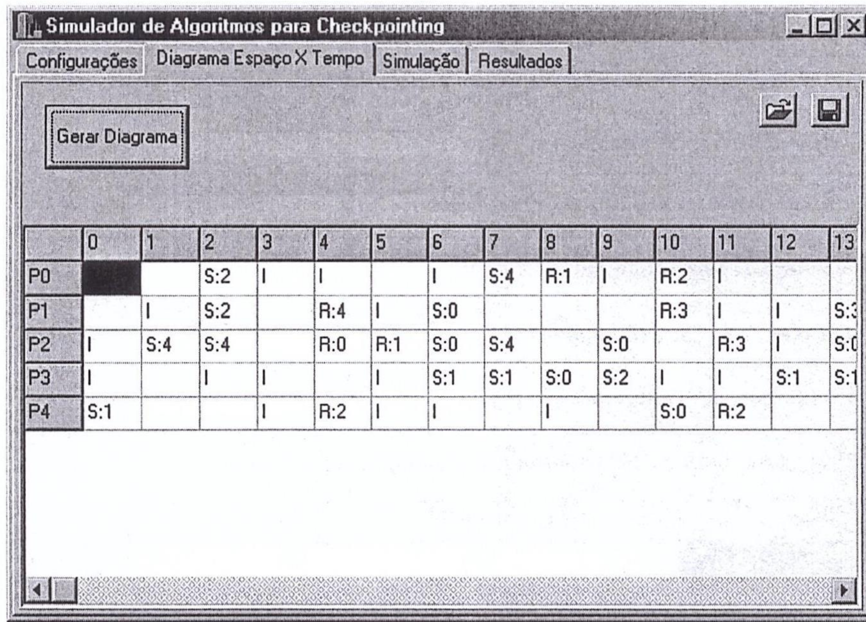

Figura 6.3: Interface da ferramenta apresentando a matriz que representa o diagrama de Espaço × Tempo

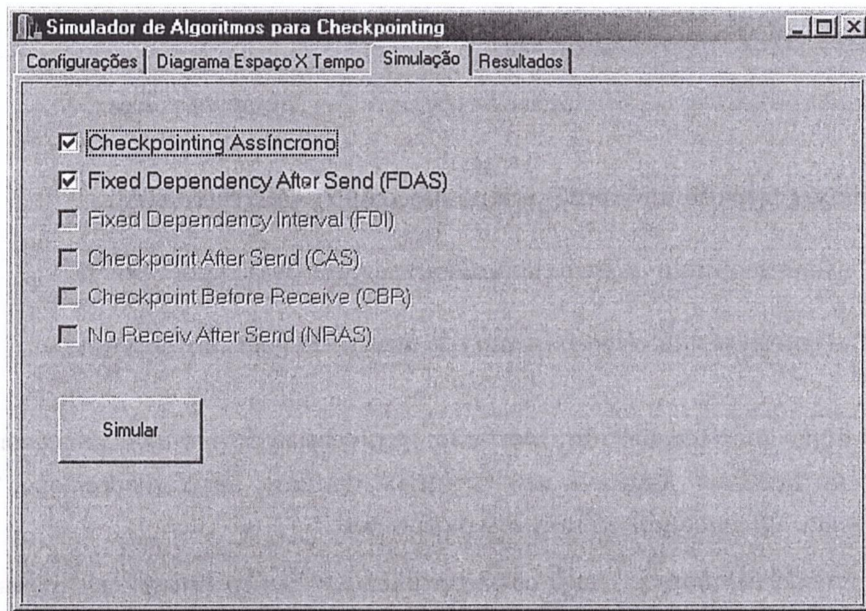

Figura 6.4: Interface da ferramenta para análise de algoritmos para *checkpointing*

máximo, 20.000 eventos. Os intervalos de *checkpoints* variaram de 5 até 25, em passos de 5, isto é, a simulação foi realizada para intervalos de 5, 10, 15, 20 e 25 eventos internos e cada combinação foi repetida 10 vezes.

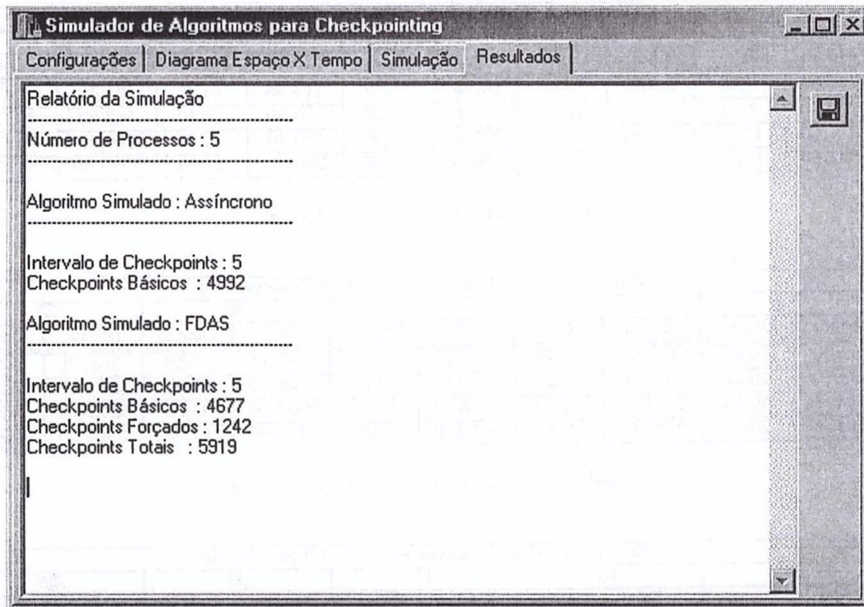

Figura 6.5: Interface da ferramenta para análise de algoritmos para *checkpointing*

Eventos	Classe 1	Classe 2	Classe 3	Classe 4
Neutros	30 %	25 %	20 %	15 %
Internos	60 %	55 %	50 %	45 %
Send	05 %	10 %	15 %	20 %
Receive	05 %	10 %	15 %	20 %

Tabela 6.1: Classes dos testes para o Simulador de Algoritmos para *Checkpointing*

6.1.5 Análise dos Resultados

As tabelas 6.2, 6.3, 6.4 e 6.5 apresentam os resultados encontrados para as classes 1, 2, 3 e 4, respectivamente. Estes dados representam a diferença de *checkpoints* entre o algoritmo FDAS (*Fixed Dependency After Send*) e o mecanismo assíncrono, nas respectivas classes. Para cada uma das tabelas, apresenta-se os resultados e o respectivos desvios padrões obtidos através da execução dos eventos na simulação dos algoritmos. Nas tabelas 6.6, 6.7, 6.8 e 6.9 são apresentados os resultados relativos para as classes 1, 2, 3 e 4, respectivamente.

	Diferença dos Intervalos entre Checkpoints									
	5		10		15		20		25	
p	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
2	120,00	9,35	147,20	13,07	161,90	15,32	174,80	10,08	173,40	7,82
4	265,70	9,89	361,40	11,40	435,40	17,37	490,90	18,40	543,80	19,87
6	411,60	17,77	569,30	25,06	685,00	27,36	799,70	21,77	890,20	24,05
8	557,50	19,79	765,80	19,66	940,00	23,80	1086,80	27,86	1218,60	21,69

Tabela 6.2: Resultados da classe 1

	Diferença dos Intervalos entre Checkpoints									
	5		10		15		20		25	
p	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
2	392,90	16,86	421,80	15,93	423,00	12,34	412,50	20,88	383,70	12,88
4	986,20	15,58	1224,20	24,38	1399,50	29,97	1538,00	35,89	1662,30	39,45
6	1548,30	36,53	1964,60	39,36	2315,80	42,62	2588,00	44,90	2821,20	44,76
8	2101,30	25,75	2706,80	40,41	3212,00	55,85	3635,00	64,65	3992,50	55,97

Tabela 6.3: Resultados da classe 2

	Diferença dos Intervalos entre Checkpoints									
	5		10		15		20		25	
p	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
2	730,00	19,80	705,50	22,84	659,80	28,76	598,30	27,99	536,90	20,33
4	1975,30	40,30	2332,70	45,41	2623,10	35,80	2853,60	31,33	3041,40	47,42
6	3121,20	50,13	3832,10	54,73	4408,00	65,25	4880,60	78,16	5265,80	80,01
8	4313,50	65,61	5379,20	63,76	6245,00	67,36	6965,70	66,94	7526,90	56,12

Tabela 6.4: Resultados da classe 3

	Diferença dos Intervalos entre Checkpoints									
	5		10		15		20		25	
p	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
2	1060,90	34,58	933,10	26,88	803,20	35,86	695,90	39,37	601,90	24,03
4	3120,70	31,48	3608,60	51,40	4014,20	67,30	4351,90	76,03	4629,30	89,60
6	5077,30	56,24	6150,90	61,54	7012,20	66,07	7653,30	71,73	8126,30	75,21
8	6966,90	58,47	8553,50	83,84	9796,60	93,29	10706,70	97,13	11357,50	94,50

Tabela 6.5: Resultados da classe 4

	Diferença dos Intervalos entre Checkpoints									
	5		10		15		20		25	
p	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
2	2,50%	0,20%	6,14%	0,57%	10,14%	0,99%	14,59%	0,89%	18,10%	0,84%
4	2,77%	0,11%	7,53%	0,24%	13,62%	0,57%	20,48%	0,80%	28,36%	1,07%
6	2,86%	0,13%	7,91%	0,35%	14,28%	0,58%	22,23%	0,64%	30,94%	0,87%
8	2,90%	0,10%	7,98%	0,21%	14,70%	0,39%	22,67%	0,60%	31,77%	0,59%

Tabela 6.6: Resultados relativos da classe 1

	Diferença dos Intervalos entre Checkpoints									
	5		10		15		20		25	
p	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
2	8,91%	0,39%	19,13%	0,73%	28,79%	0,87%	37,44%	1,91%	43,54%	1,45%
4	11,21%	0,19%	27,84%	0,58%	47,75%	1,07%	69,98%	1,69%	94,56%	2,36%
6	11,73%	0,28%	29,77%	0,60%	52,65%	1,02%	78,47%	1,40%	106,96%	1,76%
8	11,93%	0,16%	30,74%	0,47%	54,72%	0,95%	82,58%	1,43%	113,42%	1,57%

Tabela 6.7: Resultados relativos da classe 2

	Diferença dos Intervalos entre Checkpoints									
	5		10		15		20		25	
p	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
2	18,25%	0,49%	35,29%	1,19%	49,51%	2,28%	59,88%	2,92%	67,18%	2,66%
4	24,61%	0,53%	58,15%	1,22%	98,11%	1,51%	142,35%	1,80%	189,69%	3,36%
6	25,92%	0,49%	63,65%	1,09%	109,85%	1,95%	162,24%	3,00%	218,83%	3,85%
8	26,93%	0,45%	67,18%	0,93%	117,02%	1,47%	174,07%	2,00%	235,17%	2,19%

Tabela 6.8: Resultados relativos da classe 3

	Diferença dos Intervalos entre Checkpoints									
	5		10		15		20		25	
p	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	
2	29,39%	0,96%	51,72%	1,84%	66,79%	3,09%	77,18%	4,16%	83,49%	3,29%
4	43,22%	0,41%	100,00%	1,36%	166,90%	2,78%	241,34%	4,13%	320,97%	6,16%
6	46,88%	0,60%	113,63%	1,29%	194,38%	2,23%	282,89%	3,19%	375,57%	4,36%
8	48,26%	0,50%	118,53%	1,38%	203,71%	2,32%	296,89%	3,16%	393,79%	3,88%

Tabela 6.9: Resultados relativos da classe 4

A figura 6.6 apresenta os gráficos das médias e dos resultados relativos representando as respectivas tabelas. Nestes gráficos têm-se a diferença entre o número de *checkpoints*, obtidos no algoritmo FDAS em relação a abordagem assíncrona, em função do tamanho do intervalo de *checkpoints*. Nota-se que, independente da classe de aplicações, o comportamento do algoritmo foi semelhante.

É importante observar que as classes 3 e 4 representam situações especiais onde as comunicações entre os processos ocorrem cerca de 30% a 40% do tempo total de processamento. Nessas aplicações, a probabilidade de acontecer caminhos-Z é alta e, por conseguinte, os algoritmos que tentam prevenir esse tipo de ocorrência, como o FDAS, são obrigados a obter um número significativo de *checkpoints* forçados para garantir a existência das linhas de recuperação. Entretanto, é preciso considerar que, em um programa de simulação distribuída, quanto maior a quantidade de mensagens trocadas entre os processos lógicos, maior é a probabilidade de ocorrer *rollbacks*. Em adição, com o aumento no número dos *checkpoints*, devido aos *checkpoints* forçados, o intervalo diminui e, desta forma, a quantidade de eventos desperdiçados durante o *rollback* também. Além disso, simulações com muita troca de mensagens necessitam de um intervalo entre os *checkpoints* de tamanho pequeno para diminuir a fase *coast forward*.

Segundo o estudo realizado por Vieira (2001), o número de *checkpoints* forçados pelos algoritmos do padrão ZPF não é influenciado significativamente pelo tamanho dos intervalos de *checkpoints*. Entretanto, esse estudo foi realizado em comparação com outros algoritmos semi-síncronos para *checkpointing*. A análise desta tese comparou o algoritmo FDAS com a abordagem assíncrona o que, de certa forma, não é adequado, uma vez que a abordagem assíncrona não garante a utilidade dos *checkpoints*. Mesmo assim, a comparação foi feita, pois o protocolo *Time Warp* utiliza a abordagem assíncrona com o mecanismo *Sparse State Saving* que não precisa impedir a ocorrência de caminhos-Z, pois, como discutido no capítulo 2, utiliza anti-mensagens para recompor a simulação ao

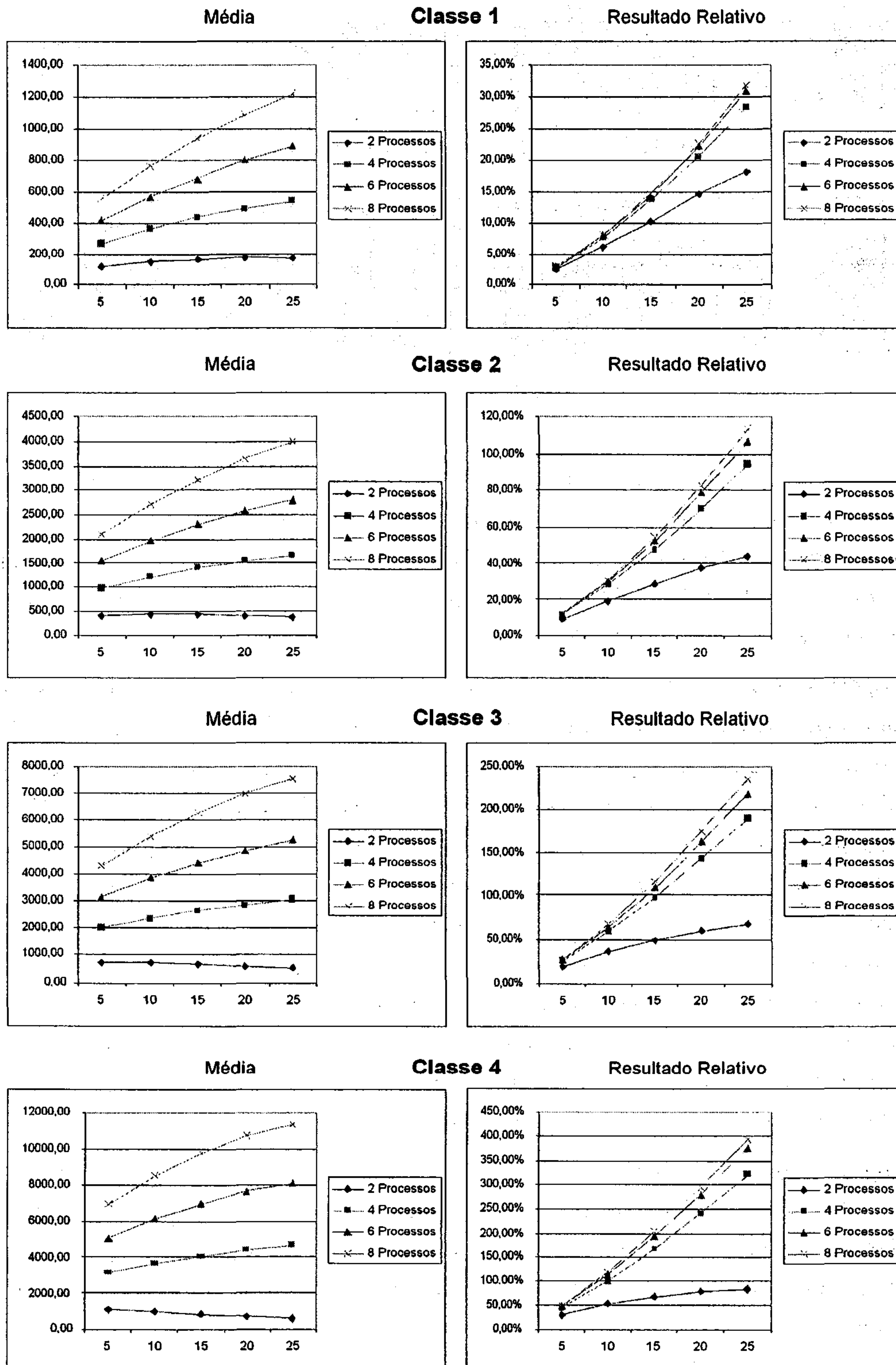

Figura 6.6: Gráficos dos dados absolutos e relativos das 4 classes

invés das linhas de recuperação usadas no protocolo *Rollback* Solidário.

Através deste estudo ainda é possível observar que, quando o número de processos aumenta, tem-se, incialmente, um crescimento considerável na diferença de *checkpoints* entre os métodos analisados. Entretanto, este crescimento se estabiliza conforme pode ser observado na figura 6.7. Cabe ressaltar que a medida que o intervalo entre *checkpoints* aumenta o número de *checkpoints* diminui, tanto para a abordagem semi-síncrona (FDAS) quanto para a abordagem assíncrona, o que, intuitivamente, espera-se que aconteça. É preciso destacar, entretanto, que os gráficos desta análise representam o valor absoluto das diferenças entre as duas abordagens que, evidentemente, justifica o comportamento observado.

Pela inequação 6.15, o tempo necessário para armazenar o excesso dos *checkpoints* obtidos pela abordagem semi-síncrona, em relação a abordagem assíncrona, deve ser menor que a diferença de desempenho entre os protocolos *Time Warp* e *Rollback* Solidário, para que este supere aquele em desempenho. Entretanto, conforme o número de processos e mensagens aumenta no sistema, a probabilidade de ocorrer *rollbacks* secundários também aumenta. No *Time Warp* este aumento tem uma complexidade quadrática, mas, como pode ser observado nos gráficos desta seção, o aumento de χ não segue a mesma tendência. Desta forma, os resultados obtidos em relação ao aumento do número de *checkpoints* torna o protocolo *Rollback* Solidário ainda mais atrativo quando o número de processos da simulação aumenta.

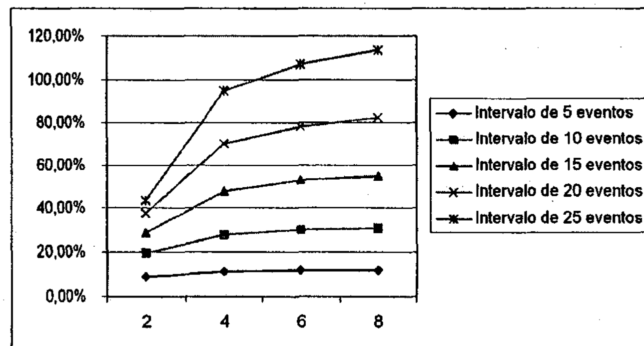

Figura 6.7: Gráfico considerando a diferença relativa de *checkpoints* com conforme varia o número de processos

6.2 FDAS para Simulação Distribuída

Preiss, MacIntyre e Loucks (1992) conduziram um estudo experimental cujos resultados indicaram que o comportamento dos *rollbacks*, na abordagem *Sparse State Saving* do protocolo *Time Warp*, é afetado pela freqüência da obtenção dos *checkpoints*. Assim, Lin

et al. (1993) apresentaram um modelo que deriva o intervalo de *checkpoints* a partir do comportamento dos *rollbacks*. Como foi discutido no capítulo 2, a sobrecarga para garantir a realização dos *rollbacks*, em um sistema otimista, é o resultado de dois componentes: o armazenamento dos estados (*checkpointing*) e a sua restauração, ou seja, o próprio *rollback*. Por conseguinte, quanto menor o intervalo entre os *checkpoints* menor será o custo da recuperação e maior o número de *checkpoints*. Além disso, quanto menor o número de mensagens, menor é a probabilidade de ocorrer *rollbacks* e também será pequeno o número de *checkpoints* forçados, que serão induzidos pela comunicação.

Com base nos estudos de Lin et al. (1993) e Palaniswamy e Wilsey (1993), Rönngren e Ayani (1994) propuseram um método adaptativo para a definição dos intervalos entre os *checkpoints* para o protocolo *Time Warp*. O método refina o valor do intervalo dos *checkpoints* de forma iterativa. A equação 6.16 obtém o intervalo entre *checkpoints* que minimiza o tempo de execução, onde R_{obs} é o número de execuções de eventos observados, k_{obs} é o número de *rollbacks* obtido durante o intervalo de observação R_{obs}, δ_s é o tempo médio para salvar um estado do processo lógico e δ_c é a média do tempo de execução para a fase *coast forward*.

$$\Delta\chi_{min} = \left\lceil \sqrt{2\frac{R_{obs} \cdot \delta_s}{k_{obs} \cdot \delta_c}} \right\rceil \tag{6.16}$$

A partir da equação 6.16, Rönngren e Ayani (1994) apresentaram um procedimento iterativo para refinar o valor de $\Delta\chi_{min}$. A próxima iteração de $\Delta\chi_{min}$ após o n-ésimo intervalo observado é dado pela equação 6.17.

$$\begin{aligned} \Delta\chi_n &= \text{if } (n = 0) \text{ then } \Delta\chi_{inicial} \\ &= \text{else if } (K_{obs} = 0) \text{ then } \lceil (1-\rho)\Delta\chi_{n-1} + \rho\Delta\chi_{max} \rceil \\ &= \text{else max } (1, \lceil (1-\rho)\Delta\chi_{n-1} + \rho \text{ min } (\Delta\chi_{min}, \Delta\chi_{max}) \rceil) \end{aligned} \tag{6.17}$$

Nos testes experimentais de Rönngren e Ayani (1994) foram utilizados os seguintes valores para os parâmetros deste método: R_{obs} igual a execução de 200 eventos, o intervalo máximo entre os *checkpoints* $\Delta\chi_{max}$ foi 30, $\Delta\chi_{inicial}$ foi 4 e $\rho = 0,4$. O sistema utilizado pelos autores define os valores de δ_s e δ_c durante os primeiros 200 eventos executados de cada processo lógico ou após ter alcançado, pelo menos, 30 medidas para δ_c.

Em um protocolo otimista, o processo com o menor LVT possui a maior probabilidade de enviar mensagens *stragglers*. Entretanto, ele não pode ser afetado por este tipo de mensagem, uma vez que os eventos enviados pelos outros processos estarão sendo escalonados para o futuro deste processo. Se a implementação do protocolo *Rollback*

Solidário mantiver um vetor com os relógios lógicos, conforme discutido na seção 4.9, é possível utilizar estas informações para refinar o valor de $\Delta\chi$ aumentando o grau de acerto da equação 6.17. O processo mais atrasado pode aumentar o seu intervalo entre *checkpoints*, pois tem uma probabilidade menor de realizar *rollback*, enquanto os processos mais adiantados devem diminuir o intervalo. Isto aumenta o otimismo do processo mais atrasado e diminui o otimismo do processo mais adiantado, criando um equilíbrio entre os processos e minimizando a distância a ser percorrida em caso de *rollback*.

Como pode ser analisado na equação 6.17, ρ determina a velocidade com que o intervalo entre os *checkpoints* se aproxima do valor $\Delta\chi_{max}$. Assim, o valor de ρ pode ser alterado dinamicamente de acordo com os valores dos LVTs dos processos da simulação. Seja ΔLVT_{max} a diferença do maior LVT com relação ao menor LVT entre todos os processos, então o valor de ρ para cada processo pode ser obtido pela equação 6.18. Nesta equação, ΔLVT é a diferença entre o maior LVT e o LVT do processo que está obtendo o próximo intervalo entre os *checkpoints*.

$$\rho = \frac{\Delta\text{LVT}}{\Delta\text{LVT}_{max}} \tag{6.18}$$

Outra melhoria que pode ser implementada, com relação à quantidade de *checkpoints* forçados na aplicação, tem como base o trabalho de Manivannan e Singhal (1996). O algoritmo proposto pelos autores substitui um *checkpoint* básico pelo último *checkpoint* forçado do processo, caso ele exista. Para implementar esse algoritmo, deve-se manter duas variáveis para contar os eventos em cada intervalo entre *checkpoints*. A primeira variável (`Normal`) funciona como contador do método assíncrono, identificando os intervalos que seriam obtidos se a aplicação não induzisse *checkpoints* forçados. O segundo contador (`Forçado`) é zerado toda vez que um *checkpoint* ocorre, seja ele forçado ou básico. Quando a primeira variável (`Normal`) identifica o fim do intervalo, o sistema compara as duas variáveis. Caso elas estejam iguais o sistema obtém um *checkpoint* básico. Neste instante, as duas variáveis são reiniciadas com zero.

6.3 Complexidade do Espaço de Armazenamento

Esta seção discute o relacionamento da complexidade de espaço de memória entre os protocolos da simulação distribuída *Time Warp*, utilizando o mecanismo *Sparse State Saving*, e o protocolo proposto nesta tese.

Definição 6.3.1 (Estado Local no protocolo *Time Warp*) *O estado local do processo p_i, no tempo lógico τ, pode ser definido através da tupla ordenada $\sigma_i^\tau = (X_i^\tau, I_i^\tau, O_i^\tau)$ onde:*

X_i^τ: *conjunto dos estados salvos do processo* p_i *tal que* $X_i = \{x \in X_i \mid GVT(\tau) \leq LVT(x) \leq \tau\}$;

I_i^τ: *conjunto dos eventos na fila de entrada, no tempo lógico* τ, *e os eventos que já foram enviados por outros processos para* p_i, *mas que ainda não foram recebidos por* p_i;

O_i^τ: *conjunto dos eventos na fila de saída, no tempo lógico* τ, *representando a lista de anti-mensagens do processo* p_i;

A definição de estado local, fundamenta-se na Teoria dos Conjuntos e contempla as mensagens transientes do sistema. A partir dessa definição, obtém-se a definição de estado global para o protocolo *Time Warp* (definição 6.3.2).

Definição 6.3.2 (Estado Global no protocolo *Time Warp*) *O estado global do sistema, no tempo lógico* τ, *é a união dos estados locais de todos os processos da simulação, ou seja:*

$$\Sigma(\tau) = \bigcup_{1 \leq i \leq n} \sigma_i^\tau \tag{6.19}$$

Para efeito de comparação, o espaço de armazenamento utilizado pelo protocolo *Time Warp* durante a execução da simulação é dado pela maior área de memória ocupada durante a simulação. Portanto, o espaço de armazenamento pode ser representado pela equação 6.20.

$$\Delta_{TW} = \max_{\forall \tau} \mid \Sigma(\tau) \mid \tag{6.20}$$

No protocolo *Rollback* Solidário, o estado local apresenta algumas diferenças em relação ao protocolo *Time Warp*. A principal delas é a inexistência da fila de saída. Portanto, o conjunto O_i^τ não faz parte do protocolo e representa um ganho no seu gerenciamento de memória, entretanto ele possui a desvantagem de armazenar o vetor de dependências entre os *checkpoints*. Na prática, esse vetor faz parte do conjunto X_i^τ, pois representa um estado interno do processo, porém, para facilitar a comparação, ele será tratado separadamente na definição de estado local do protocolo *Rollback* Solidário, conforme definição 6.3.3.

Definição 6.3.3 (Estado Local no protocolo *Rollback* Solidário) *O estado local do processo* p_i, *no tempo lógico* τ, *pode ser definido através da tupla ordenada* $\sigma_i^\tau = (X_i^\tau, I_i^\tau, V_i^\tau)$ *onde:*

X_i^τ: *conjunto dos estados salvos do processo* p_i *sem o vetor de dependências entre checkpoints, tal que* $X_i = \{x \in X_i \mid GVT(\tau) \leq LVT(x) \leq \tau\}$;

I_i^τ: *conjunto dos eventos na fila de entrada, no tempo lógico* τ, *e os eventos que já foram enviados por outros processos para* p_i, *mas que ainda não foram recebidos por* p_i;

V_i^τ: *conjunto unitário, contendo o vetor de dependências entre os checkpoints;*

Como os conjuntos X_i^τ e I_i^τ, pelas definições, possuem a mesma cardinalidade, é direta a percepção de que a diferença entre a complexidade de espaço dos protocolos está no impacto causado pelo armazenamento do vetor de dependências entre *checkpoints* em relação à economia que o protocolo *Rollback* Solidário possui de não armazenar anti-mensagens. Apesar das mensagens que trafegam no sistema serem rotuladas com o vetor de dependências, essa análise trata o conjunto I_i^τ dos dois protocolos com a mesma cardinalidade, pois, quando uma mensagem chega, o vetor de dependências que acompanha a mensagem é imediatamente analisado para a atualização do vetor local, sendo, em seguida, retirado da mensagem, antes desta ser inserida na lista de eventos futuros. Além disso, o conjunto I_i^τ do protocolo *Rollback* Solidário em alguns momentos possui uma quantidade menor de elementos que o mesmo conjunto no protocolo *Time Warp*. Isto ocorre devido à necessidade de eliminar possíveis mensagens cujos processos emissores tenham realizado *rollbacks*.

A definição de estado global no protocolo *Rollback* Solidário é idêntica à definição correspondente do protocolo *Time Warp*.

Definição 6.3.4 (Estado Global no protocolo *Rollback* Solidário) *O estado global do sistema, no tempo lógico* τ, *é a união dos estados locais de todos os processos da simulação, ou seja:*

$$\Sigma(\tau) = \bigcup_{1 \leq i \leq k} \sigma_i^\tau$$

Novamente, a análise comparativa depende do modelo que está sendo simulado, pois o número de anti-mensagens está relacionado com a interação entre os processos.

Um aspecto a ser considerado consiste no fato de que o conjunto V_i^τ é sempre unitário. Assim, o gerenciamento de memória em um ambiente que precise restringir o espaço de memória disponível para a simulação é mais fácil de ser realizado devido à maior constância do espaço de memória necessário a cada passo do programa. Além disso, a cardinalidade do conjunto O_i^τ é dependente do cálculo do GVT e da re-alocação de áreas na memória, o que não acontece com o conjunto V_i^τ.

Uma mensagem no protocolo *Time Warp*, em sua implementação mais simples, armazena o identificador do processo destino da mensagem, o LVT do processo que criou o evento e o tempo para o qual ele foi escalonado. Se for considerado que cada elemento do conjunto O_i^T ocupa o espaço de três posições do vetor de dependências do protocolo *Rollback* Solidário, então, para que o conjunto V_i^T ocupe menos espaço que o conjunto O_i^T, será necessário o armazenamento de, pelo menos, $\lceil (n/3) \rceil$ anti-mensagens, por cada processo do protocolo *Time Warp*. Se a simulação se desenvolve em um ambiente com poucos processos lógicos, percebe-se que esta quantidade de anti-mensagens é rapidamente alcançada. Mesmo em situações em que o número de processos é alto, a quantidade de anti-mensagens pode superar o espaço gasto pelo vetor de dependências se não for realizado um controle adequado de memória através da re-alocação de espaços já utilizados, após o cálculo do GVT. Isto exige que se aumente a freqüência da realização deste cálculo, o que, por sua vez, prejudica o desempenho da simulação.

6.4 Considerações Finais

Uma característica importante do protocolo *Rollback* Solidário é sua melhor escalabilidade em relação ao protocolo *Time Warp*, devido ao comportamento linear, no desperdício de tempo, quando vários processos são afetados pela mesma mensagem *straggler*.

Em um ambiente com poucas mensagens e com um bom particionamento do modelo simulado, o protocolo *Time Warp* tende a ser uma boa opção, uma vez que o espaço de armazenamento é menor. Quanto maior o número de processos, mais espaço será ocupado pelo conjunto V_i^T, entretanto o número de mensagens também deverá aumentar, equilibrando a comparação entre os dois protocolos.

O cálculo do GVT para coleta de lixo é um aspecto importante nos dois protocolos por causa da necessidade de espaço do conjunto X_i^T, mas é mais crítico no protocolo *Time Warp* devido ao crescimento do conjunto O_i^T, o que não ocorre com o conjunto V_i^T.

Apesar de ter sido utilizada uma distribuição de probabilidade hipotética, o procedimento utilizado para a análise permite comparar elementos semelhantes entre os dois protocolos. Isso possibilita a criação de um mecanismo dinâmico de troca de protocolos durante a simulação, pois as variáveis das equações 6.9 e 6.13 podem ser atualizadas em tempo de execução. Assim, a troca entre os protocolos *Rollback* Solidário e *Time Warp* pode ser realizada sem grande impacto no sistema, tornando o mecanismo apresentado mais interessante.

Capítulo 7

Conclusões

O protocolo *Rollback* Solidário, apresentado nesta tese, é um mecanismo para sincronizar processos lógicos de uma aplicação de simulação distribuída, fundamentado no conceito de *checkpoints* globais consistentes. É um protocolo otimista, pois permite que cada processo execute seus eventos de forma independente em relação aos outros processos da simulação. A diferença principal entre este protocolo e os demais protocolos otimistas consiste na identificação das linhas de recuperação associadas aos pontos de retorno da simulação durante o procedimento de *rollback*. Esta característica flexibiliza a sincronização dos processos, evitando a necessidade de armazenar anti-mensagens.

O protocolo *Rollback* Solidário une conceitos das áreas de simulação distribuída e tolerância a falhas. Os *checkpoints* globais consistentes, que na área de tolerância a falhas são usados para garantir que haverá uma linha de recuperação para recompor a computação em caso de falha no sistema, são empregados, neste protocolo, para definir pontos de retorno para a execução dos *rollbacks*.

Com a utilização dos *checkpoints* globais consistentes surgem novas perspectivas de pesquisa na área de Simulação Distribuída, uma vez que a teoria dos *checkpoints*, sejam eles síncronos, assíncronos ou semi-síncronos, está bem consolidada e vários resultados que hoje são utilizados em sistemas tolerantes a falhas podem produzir resultados interessantes em um programa de simulação, em especial utilizando o protocolo *Rollback* Solidário. Além disso, o *Time Warp* é um protocolo em constante evolução desde a sua apresentação 20 anos atrás. Grande parte dos trabalhos desenvolvidos podem ser reavaliados para serem utilizados junto ao protocolo apresentado nesta tese, como é o caso do mecanismo proposto por Santoro (2003), que permite a obtenção dos *checkpoints* em paralelo às rotinas da simulação, evitando a completa interrupção dos processos lógicos durante o salvamento dos seus estados. Desta forma, novos trabalhos podem favorecer uma rápida convergência do protocolo *Rollback* Solidário para um protocolo eficiente.

Durante o desenvolvimento deste trabalho, várias conclusões foram alcançadas, sendo

a principal delas a empregabilidade dos *checkpoints* globais consistentes, para sincronizar processos nos programas de simulação distribuída, possuindo uma aplicação direta na abordagem otimista, devido a uma sincronização mais ágil entre os processos envolvidos em um *rollback* e a eliminação da necessidade do uso de anti-mensagens. Com isso, há uma melhoria no desempenho da simulação e uma diminuição do montante de memória necessário para o funcionamento do programa de simulação.

O mecanismo síncrono para obtenção dos *checkpoints* foi utilizado nesta tese para demonstrar que os cortes consistentes permitem sincronizar os processos durante o procedimento de *rollback* de um protocolo otimista. Esta abordagem realiza o cálculo do GVT de forma direta toda vez que o procedimento de *checkpointing* acontece, tornando o gerenciamento da memória uma tarefa mais simples. Outra característica deste mecanismo, é que ele provoca a limitação do otimismo dos processos da aplicação, favorecendo o desenvolvimento de protocolos híbridos que utilizem técnicas conservativas e otimistas. Além disso, mecanismos dinâmicos para troca de protocolos podem se beneficiar, pois durante a sincronização dos processos é possível obter informações globais para a decisão de mudar o protocolo atual.

Entretanto, em função da necessidade da aplicação estar constantemente obtendo os *checkpoints*, o desempenho é bastante prejudicado com a utilização de algoritmos síncronos. Na prática, o protocolo *Rollback* Solidário é viável com a utilização dos mecanismo semi-síncronos, visto que estes possuem grande flexibilidade e, dependendo do método utilizado, pouco impacto no sistema.

Uma conclusão importante deste trabalho é a constatação de que a identificação das linhas de recuperação, visando ao retorno dos processos durante o procedimento de *rollback*, permite evitar o efeito cascata que ocorre em outros protocolos otimistas, como o *Time Warp*, possibilitando o desenvolvimento de um protocolo eficiente.

A definição do protocolo *Rollback* Solidário contemplou duas abordagens para a identificação dos *checkpoints* globais consistentes, com observador e sem observador. A utilização de um processo observador facilita a localização das linhas de recuperação em tempo de execução, dando maior eficiência para a aplicação durante o procedimento de *rollback*. O processo observador é pouco intrusivo, mas aumenta o número de mensagens na rede de comunicação.

As linhas de recuperação oferecem uma solução natural para o problema do cálculo do GVT. Na abordagem com o processo observador, este cálculo pode ser constantemente atualizado, sem prejuízo para a simulação, visto que os processos da simulação podem receber o valor do GVT em intervalos regulares de tempo, ou durante a ocorrência dos *rollbacks*. As linhas de recuperação, por estarem associadas a um corte consistente da aplicação, podem ser utilizadas para implementar mecanismos de mudança dinâmica de

protocolo e balanceamento de carga com migração de processos. Neste caso, o processo observador pode assumir as funções de um processo monitor, obtendo mais informações a respeito da carga dos processadores.

No entanto, em aplicações distribuídas é desejável que não exista um ponto de gargalo no sistema. Neste contexto, foi discutida uma versão sem observador do protocolo *Rollback* Solidário que apresenta um comportamento descentralizado durante o tratamento dos *rollbacks*. Algumas das soluções apresentadas para esta versão não possuem a mesma simplicidade e eficiência da versão com observador, mas permite que todos os processos da simulação estejam executando tarefas associadas a simulação em si.

A agilidade do procedimento de *rollback* evitando o efeito cascata, proporciona maior escalabidade ao protocolo *Rollback* Solidário. Além disso, a utilização das linhas de recuperação possibilita a criação de diversos mecanismos que necessitam tomar decisões baseadas em informações que são obtidas através de pontos de visualização global da computação, como é o caso dos procedimentos para troca de protocolos e mecanismos para o balanceamento dinâmico da carga no sistema.

7.1 Contribuições deste trabalho

A principal contribuição deste trabalho é a proposta de um novo protocolo para sincronização de processos em Simulação Distribuída. A idealização e o desenvolvimento do protocolo *Rollback* Solidário apresentam diversas fases, sendo que em cada uma delas pode-se identificar contribuições, tanto para a área de Simulação Distribuída como para outras áreas correlatas. A seguir essas fases são discutidas e as contribuições apresentadas:

1. Estudo dos protocolos otimistas e identificação de pontos que podem ser aprimorados:
    - O trabalho apresentado nesta tese discutiu vários aspectos relacionados ao comportamento dos processos durante a sincronização de suas atividades nos protocolos otimistas da simulação distribuída. O surgimento do protocolo *Rollback* Solidário foi o resultado do estudo das formas de sincronização dos eventos que ocorrem em um sistema distribuído. Os mecanismos utilizados para manter uma visão global do tempo, por todos os elementos de processamento que compõem esse tipo de ambiente e as soluções existentes para manter a coerência do sistema foram comparados com as alternativas adotadas na área de simulação distribuída.
    - Durante esta fase foi dada especial atenção aos mecanismos de gerenciamento de memória e ao cálculo do GVT. O desenvolvimento do protocolo, desde

o seu início, foi marcado pela preocupação de se encontrar novas formas de sincronismo que favorecesse o gerenciamento da memória, o que foi possível com a utilização dos *checkpoints* globais consistentes.

2. Nova visão dos *checkpoints* globais consistentes:

    - *Checkpoints* globais consistentes têm sido utilizados em sistemas distribuídos para solucionar problemas que necessitam descrever o progresso da computação ou para a verificação de predicados estáveis. Nesta tese, eles foram utilizados para sincronizar os processos da simulação distribuída durante um *rollback*, por estarem associados a um corte consistente da computação.
    - Para a utilização dos *checkpoints* o conceito de linha de recuperação foi estendido para um conjunto de linhas de recuperação, pois a necessidade de retrocesso da computação não ocorre por falhas no sistema, mas pela ocorrência de eventos que ferem a relação de causalidade entre os processos.

3. Definição do funcionamento do protocolo *Rollback* Solidário:

    - O capítulo 4 descreve o funcionamento do protocolo apresentando as atividades que são realizadas para a obtenção das linhas de recuperação e os procedimentos realizados durante um *rollback*.
    - São apresentadas formas alternativas para a obtenção das linhas de recuperação que favorecem um rápido retorno dos processos durante um *rollback*. Na simulação distribuída, essa exigência é fundamental, uma vez que os erros de causalidade são inerentes aos protocolos otimistas.
    - É apresentado um método para a localização da linha de recuperação máxima com base na criação de uma matriz que circula entre os processos que, no primeiro passo, não se ajustaram ao ponto de retorno definido pelo processo coordenador do procedimento de *rollback*. Este mecanismo pode ser utilizado em sistemas tolerantes a falhas, sem alterações.
    - Além da especificação do protocolo e demonstração da viabilidade do uso de *checkpoints* globais consistentes, para sincronizar os processo durante o procedimento de *rollback*, esta tese propõe soluções para alguns problemas que surgem em programas de computação distribuída, como o problema da mensagem transiente e *rollbacks* simultâneos.

4. Projeto formal para a implementação do protocolo *Rollback* Solidário:

    - O capítulo 5 apresenta o projeto formal para implementação do protocolo utilizando os diagramas de *classes, seqüências* e *estados* da UML (*Unified*

Modeling Language). Este projeto, em conjunto com a definicão do protocolo, favorece a visão do comportamento e das características dos objetos que compõem o modelo, oferecendo um projeto conciso para a implementação do mesmo.

- As listagens que compõem o apêndice A detalham alguns dos diagramas de atividades apresentados no capítulo 4, fornecendo recursos de implementação para aqueles que desejam utilizar o protocolo *Rollback* Solidário.

5. Comparação entre os protocolos *Rollback* Solidário e *Time Warp*:

- Para a comparação entre os protocolos, uma seqüência de passos foi descrita no capítulo 6, criando uma abordagem que poderá ser utilizada para comparar outros protocolos.
- Com base em estudos para a definição dinâmica do intervalo entre *checkpoints* do mecanismo *Sparse State Saving* para o protocolo *Time Warp*, foi proposto um método iterativo que estende o modelo de Rönngren e Ayani (1994), aproveitando as características do protocolo *Rollback* Solidário.
- Para a análise do impacto dos *checkpoints* forçados foi desenvolvida uma ferramenta para simulação dos algoritmos de *checkpointing*, que pode ser adaptada para a análise de outras características dos ambientes distribuídos, por representar a computação distribuída através de um diagrama de espaço×tempo.

7.2 Sugestões para Trabalhos Futuros

Por se tratar de um novo protocolo que utiliza uma abordagem diferente para a realização do sincronismo entre os processos, vários projetos podem ser considerados para dar continuidade ao projeto *Rollback* Solidário, entre eles:

- A implementação do protocolo contemplando as técnicas da orientação a objetos, como apresentado na especificação do protocolo, utilizando a modelagem descrita e as definições dos algoritmos.

- A definição deste protocolo, na abordagem semi-síncrona para a obtenção dos *checkpoints*, teve como base o algoritmo *Fixed Dependency After Send* para gerenciar a obtenção dos *checkpoins* e garantir a utilidade de todos eles. Entretanto, este algoritmo pode ser melhorado considerando as características específicas da aplicação, na tentativa de diminuir a quantidade de *checkpoints* forçados. Propõe-se, então, o estudo e o desenvolvimento de algoritmos semi-síncronos para obtenção dos *checkpoints*, explorando as características dos programas de simulação.

- A realização de um estudo comparativo dos algoritmos de *checkpointing*, semelhante ao realizado em Vieira (2001), mas que procure avaliar o desempenho desses algoritmos junto ao protocolo *Rollback* Solidário. Este estudo deve contemplar o impacto que o aumento no número dos *checkpoints* provoca na simulação, considerando que este aumento diminui o tamanho dos *rollbacks* e, por conseguinte, agiliza o procedimento de recuperação dos processos.

- A comparação do desempenho do protocolo com e sem a utilização do processo observador. O objetivo desta comparação é identificar classes ou padrões que favoreçam ou não o uso desse processo. Pode-se, inclusive, criar mecanismos que possibilitem habilitar ou desabilitar o uso do processo observador em tempo de execução, a partir de uma linha de recuperação.

- O estudo da viabilidade de se desenvolver um mecanismo para troca dinâmica entre os protocolos *Time Warp* e *Rollback* Solidário, conforme descrição apresentada no capítulo 6.

- O estudo da criação de um mecanismo de balanceamento de cargas e migração de processos para os programas de simulação distribuída. O diagrama de classes, apresentado no capítulo 6, considera a existência de uma classe `Ambiente` para permitir balanceamento de carga durante a execução do protocolo. Este ambiente pode criar mais um nível na Arquitetura em Camadas das aplicações de simulação, permitindo implementar mecanismos de migração de processos utilizando, por exemplo, agentes móveis, uma vez que a utilização de *checkpoints* pode auxiliar o procedimento empregado para migrar processos em um sistema distribuído (AGBARIA et al., 2001). Além disso, a migração de processos pode favorecer certos modelos para simulação, uma vez que possibilita o particionamento dinâmico do modelo.

- Um estudo para melhorar o procedimento descrito na solução para o problema dos "Rollbacks Simultâneos", evitando a criação de uma barreira, pois foi utilizado um algoritmo distribuído para garantir exclusão mútua entre os processos que desejam proceder o *rollback*. Entretanto, da forma como foi apresentado, o mecanismo interrompe todos os processos, inclusive aqueles que não precisam utilizar a região crítica.

- Um estudo para permitir o desenvolvimento de um mecanismo preguiçoso de aviso ao processo observador da ocorrência de novos *checkpoints* e que permita a correção das linhas de recuperação quando os *rollbacks* ocorrem, como proposto no capítulo 6.

- A realização de um estudo sistematizado e abrangente de todos os trabalhos que limitam o otimismo ou melhoram o desempenho do *Time Warp*, visando avaliar a

pertinência da aplicação desses trabalhos no *Rollback* Solidário.

Referências Bibliográficas

AGBARIA, A. et al. Quantifying rollback propagation in distributed checkpointing. *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, p. 36–45, October 2001.

AKGUL, T.; MOONEY, V. J. Assembly instruction level reverse execution for debugging. *ACM Transactions on Software Engineering and Methodology*, v. 13, n. 2, p. 149–198, April 2004.

ALLEYNE, P.; TROPPER, C. On the parallel simulation of fixed channel allocation algorithms. *Mobile Networks and Applications*, v. 5, n. 3, p. 209–218, 2000.

ALMASI, G. S. *Highly Parallel Computing*. 2nd. ed. Redwood City: The Benjamin Cummings Publishing Company, 1994. 670 p.

ANCEAUME, E.; HéLARY, J. M.; RAYNAL, M. Tracking immediate predecessors in distributed computations. *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, p. 210–219, 2002.

AXFORD, T. *Concurrent Programming - Fundamental Techniques for Real Time and Parallel Software Design*. Chichester: John Wiley & Sons, 1989.

BABAOGLU, O.; MARZULLO, K. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In: MULLENDER, S. (Ed.). *Distributed Systems*. New York: Addison-Wesley, p. 55–96, 1993.

BAEZNER, D.; LOMOW, G.; UNGER, B. W. A parallel simulation environment based on time warp. *International Journal in Computer Simulation*, v. 4, n. 2, p. 183–207, 1994.

BALDONI, R.; QUAGLIA, F.; CICIANI, B. A vp-accordant checkpointing protocol preventing useless checkpoints. *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, p. 61–67, October 1998.

BALL, D.; HOYT, S. The adaptive time-warp concurrency control algorithm. *Proceedings of the SCS Multiconference on Distributed Simulation*, v. 22, n. 1, p. 174–177, January 1990.

BANKS, J. Introduction to simulation. *Proceedings of The 1999 Winter Simulation Conference*, p. 7–13, 1999.

BELLENOT, S. State skipping performance with the Time Warp operationg system. *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, p. 53–61, January 1992.

BELLENOT, S. Performance of a riskfree Time Warp operating system. *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, p. 155–158, May 1993.

BEN-ARI, M. *Principles of Concurrent and Distributed Programming.* New York: British Library Cataloguing in Publication Data, 1990.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *UML - Guia do Usuário.* Rio de Janeiro: Editora Campus, 2000.

BOUKERCHE, A.; TROPPER, C. Local versus global lookahead in conservative parallel simulations. *Parallel Computing*, v. 27, n. 8, p. 1033–1055, July 2001.

BRIATICO, D. C.; CIUFFOLETTI, A.; SIMONCINI, L. A distributed domino-effect free recovery algorithm. *Proceedings of 4th IEEE Symposium on Reliability in Distributed Software and Database Systems*, p. 207–215, December 1984.

BRUSCHI, S. M. *ASDA - Ambiente de Simulação Distribuída Automático.* Tese (Doutorado) — Universidade de São Paulo, São Carlos, 2003.

BRYANT, R. E. *Simulation of Packet Communication Architecture Computer Systems.* Massachussets, 1977. v. 7(3), 404–425 p. MIT-LCS-TR-188.

CAI, W.; TURNER, S. J. An algorithm for distributed discrete-event simulation – the "carrier null message" approach. *Proceedings of the SCS Multiconference on Distributed Simulation*, v. 22, n. 1, p. 3–8, January 1990.

CAROTHERS, C. D.; FUJIMOTO, R. M. Effect of communication overheads on Time Warp performance: An experimental study. *Proceedings of the eighth workshop on Parallel and distributed simulation*, p. 118–125, 1994.

CAROTHERS, C. D.; PERUMALLA, K. S.; FUJIMOTO, R. M. Efficient optimistic parallel simulations using reverse computation. *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, p. 126–135, May 1999.

CHANDY, K. M.; LAMPORT, L. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, v. 3, n. 1, p. 63–75, February 1985.

CHANDY, K. M.; MISRA, J. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5, n. 5, p. 440–452, September 1979.

CHIU, G.-M.; YOUNG, C.-R. Efficient rollback-recovery technique in distributed computing systems. *IEEE Transactions on Parallel and Distributed Systems*, v. 7, n. 6, p. 565–577, June 1996.

CHUNG, M. J.; XU, J. An overhead reducing technique for Time Warp. *Proceedings of the Sixth IEEE International Workshop on Distributed Simulation and Real-Time Applications*, p. 95–102, October 2002.

CORTELLESSA, V.; QUAGLIA, F. An analysis of the efficiency of optimistically synchronized parallel simulators. *First Conference on Simulation Modeling and Applications*, 1998.

DANTAS, C. A. B. *Probabilidade: Um Curso Introdutório*. São Paulo: Edusp - Editora da Universidade de São Paulo, 2000.

DAS, S.; FUJIMOTO, R. M. An adaptive memory management protocol for Time Warp parallel simulation. *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, v. 22, p. 201–210, May 1994.

DAS, S. et al. Gtw: A Time Warp system for shared memory multiprocessors. *Proceedings of the 26th Conference on Winter Simulation*, p. 1332–1339, December 1994.

DAS, S. R. Adaptive protocols for parallel discrete event simulations. *Proceedings of the 28th Conference on Winter Simulation*, p. 186–193, 1996.

DAS, S. R.; FUJIMOTO, R. M. An empirical evaluation of performance-memory trade-offs in Time Warp. *IEEE Transactions on Parallel and Distributed Systems*, v. 8, n. 2, p. 210–224, February 1997.

DICKENS, P. M.; REYNOLDS, P. F. SRADS with local rollback. *Proceedings of the SCS Multiconference on Distributed Simulation*, v. 22, n. 1, p. 161–164, January 1990.

ELNOZAHY, E. N. et al. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, v. 34, n. 3, p. 375–408, September 2002.

ELNOZAHY, E. N.; JOHNSON, D. B.; ZWAENEPOEL, W. The performance of consistent checkpointing. *Proceedings of the Eleventh Symposium on Reliable Distributed Systems*, p. 39–47, October 1992.

FERSCHA, A. Probabilistic adaptive direct optimism control in Time Warp. *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, p. 120–129, 1995.

FIDGE, C. Logical time in distributed computing systems. *IEEE Computer*, v. 24, n. 8, p. 28–33, August 1991.

FLEISCHMANN, J.; WILSEY, P. A. Comparative analysis of periodic state saving techniques in Time Warp simulators. *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, v. 25, n. 1, p. 50–58, July 1995.

FUJIMOTO, R. M. Parallel discrete event simulation. *communications of ACM*, v. 33, n. 10, p. 31–53, October 1990.

FUJIMOTO, R. M. *Parallel and Distributed Simulation Systems*. New York: John Wiley & Sons, 2000. 300 p.

FUJIMOTO, R. M. Parallel and distributed simulation systems. *Proceedings of the 2001 Winter Simulation Conference*, p. 147–157, 2001.

FUJIMOTO, R. M. Distributed simulation systems. *Proceedings of the 2003 Winter Simulation Conference*, p. 124–134, 2003.

GAFNI, A. *Space Management and Cancellation Mechanisms for Time Warp*. Los Angeles, 1985. TR-85-341.

GAFNI, A. Rollback mechanisms for optimistic distributed simulation systems. *Proceedings of the SCS Multiconference Distributed Simulation*, v. 19, n. 3, p. 61–67, 1988.

GARCIA, I. C. *Visões Progressivas de Computações Distribuídas.* Tese (Doutorado) — Universidade de Campinas, Campinas, 2001.

GARCIA, I. C.; BUZATO, L. E. *Checkpoint Using Local Knowledge about Recovery Lines.* Campinas, 1999. IC-99-22.

GIMAC, R. L. Distributed simulation using hierarchical rollback. *Proceedings of the 1989 Winter Simulation Conference*, p. 621–629, 1989.

GUPTA, B.; LIU, Z.; LIANG, Z. On designing direct dependency - based fast recovery algorithms for distributed systems. *ACM SIGOPS Operating Systems Review*, v. 38, n. 1, p. 58–73, January 2004.

HAGENAUER, H. Global virtual time approximation for split queue Time Warp. *Proceedings of the 4th International ACPC Conference Including Special Tracks on Parallel Numerics and Parallel Computing in Image Processing, Video Processing, and Multimedia: Parallel Computation*, v. 1557, p. 541–548, 1999.

HAREL, D. Statecharts: A visual formalism for comples systems. *Science of Computer Programming*, v. 8, n. 3, p. 231–274, June 1987.

HéLARY, J. M. et al. Preventing useless checkpoints in distributed computations. *Proceedings of the 16th IEEE Symposium on Reliable Distributed Systems*, p. 183–190, October 1997.

ISKRA, K. A.; ALBADA, G. D. V.; SLOOT, P. M. A. Time warp cancellation optimisations on high latency networks. *Proceedings of the Seventh IEEE International Symposium on Distributed Simulation and Real-Time Applications*, p. 128–135, 2003.

JEFFERSON, D. R. Virtual time. *ACM Transactions on Programming Languages and Systems*, v. 7, n. 3, p. 404–425, 1985.

JEFFERSON, D. R. Virtual time ii: The cancelback protocol for storage management in distributed simulation. *Proceedings of the Ninth Annual ACM Symposium Principles of Distributed Computing*, p. 754–90, 1990.

JI, A. et al. Optimizing parallel execution of detailed wireless network simulation. *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, p. 162–169, 2004.

KALANTERY, N. Time Warp - connection oriented. *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, p. 71–77, 2004.

KELLY, O. E. et al. Scalable parallel simulations of wireless networks with WiPPET: Modeling of radio propagation, mobility and protocols. *Mobile Networks and Applications*, v. 5, n. 3, p. 199–208, 2000.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, v. 21, n. 7, p. 558–565, July 1978.

LI, L.; TROPPER, C. Event reconstruction in Time Warp. *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, p. 37–44, 2004.

LIN, Y. B. Memory management algorithms for optimistic parallel simulation. *Proceedings of the SCS Multiconference on Parallel and Distributed Simulation*, p. 43–52, January 1992.

LIN, Y. B.; LAZOWSKA, E. Optimality considerations for Time Warp parallel simulation. *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, p. 35–48, 1990.

LIN, Y. B.; PREISS, B. R. Optimal memory management for Time Warp parallel simulation. *ACM Transaction on Modeling and Computer Simulation*, v. 1, n. 4, p. 283–307, October 1991.

LIN, Y. B. et al. Selecting the checkpoint interval in Time Warp simulation. *Proceedings of the 7th Workshop Parallel and Distributed Simulation*, p. 3–10, May 1993.

LIU, L. Z.; TROPPER, C. Local deadlock detection in distributed simulations. *Proceedings of the SCS Multiconference on Distributed Simulation*, p. 64–69, January 1990.

MACDOUGALL, M. H. *Simulation Computer Systems: Techniques and Tools*. Cambridge: MIT Press, 1987. 293 p.

MADISETTI, V. K.; HARDAKER, D. A.; FUJIMOTO, R. M. The MIMDIX operating system for parallel simulation. *Proc. 6th Workshop on Parallel and Distributed Simulation*, p. 65–74, January 1992.

MANIVANNAN, D.; NETZER, R. H. B.; SINGHAL, M. Finding consistent global checkpoints in a distributed computation. *IEEE Transactions on Parallel and Distributed Systems*, v. 8, n. 6, p. 623–627, June 1997.

MANIVANNAN, D.; SINGHAL, M. A low-overhead recovery technique using quasi-synchronous checkpointing. *Proceedings of the 16th International Conference on Distributed Computing Systems*, p. 100–107, May 1996.

MANIVANNAN, D.; SINGHAL, M. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Transactions on Parallel and Distributed Systems*, v. 10, n. 7, p. 703–713, July 1999.

MARTIN, D. E.; MCBRAYER, T. J.; WILSEY, P. A. Warped: A Time Warp simulation kernel for analysis and application development. *Proceedings of the 29th Hawaii International Conference on System Sciences*, v. 1, p. 383–386, January 1996.

MATOS, A. V. de. *UML Prático e Descomplicado*. São Paulo: Érica, 2002.

MATTERN, F. Virtual time and global states of distributed systems. *Proceedings of the Parallel and Distributed Algorithms*, p. 215–226, 1989.

MATTERN, F. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, v. 8, n. 4, p. 423–434, 1993.

MCAFFER, J. A unified distributed simulation system. *Proceedings of the 22nd Conference on Winter Simulation*, p. 415–422, 1990.

MISRA, J. Distributed discrete-event simulation. *ACM Computing Surveys*, p. 39–65, 1986.

MONTGOMERY, D. C.; RUNGER, G. C. *Estatística Aplicada e Probabilidade Para Engenheiros*. Rio de Janeiro: LTC - Livros Técnicos e Científicos Editora S.A., 2003.

NETZER, R. H. B.; XU, J. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, v. 6, n. 2, p. 165–169, February 1995.

NICOL, D.; FUJIMOTO, R. M. Parallel simulation today. *Annals of Operations Research*, v. 53, p. 249–285, 1994.

NICOL, D. M.; REYNOLDS, P. F. Problem-oriented protocol design. *Proceedings of the 16th Conference on Winter Simulation*, p. 471–474, December 1984.

NUTARO, J.; SARJOUGHIAN, H. Speedup of a sparse system simulation. *Proceedings of the fifteenth Workshop on Parallel and Distributed Simulation*, p. 193–199, 2001.

OLIVEIRA, R. S.; CARISSIMI, A. S.; TOSCANI, S. S. *Sistemas Operacionais*. Porto Alegre: Sagra Luzzatto, 2000.

PALANISWAMY, A. C.; WILSEY, P. A. An analytical comparison of periodic checkpointing and incremental state saving. *ACM SIGSIM Simulation Digest*, v. 23, n. 1, p. 12–20, July 1993.

PARK, A.; FUJIMOTO, R. M.; PERUMALLA, K. S. Conservative synchronization of large-scale network simulations. *Proceedings of th 18th Workshop on Parallel and Distributed Simulation*, p. 153–161, 2004.

PEGDEN, C. D.; SHANNON, R. E.; SADOWSKI, R. P. *Introduction to Simulation using SIMAN*. 2nd. ed. New York: McGraw-Hill International Editions, 1995.

PLANK, J. S. *Efficient Checkpoints on MIMD Architectures*. Tese (Doutorado) — Princeton University, 1993.

PRAKASH, A.; SUBRAMANIAN, R. Filter: An algorithm for reducing cascaded rollbacks in optimistic distributed simulations. *Proceedings of the 24th Annual Simulations Symposium*, p. 123–132, April 1991.

PREISS, B. R.; LOUCKS, W. M. Memory management techniques for Time Warp on a distributed memory machine. *Proceedings of the Ninth Workshop Parallel and Distributed Simulation*, p. 30–39, 1995.

PREISS, B. R.; LOUCKS, W. M.; MACINTYRE, I. D. Effects on the checkpoint interval on time and space in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, v. 4, n. 3, p. 223–253, 1994.

PREISS, B. R.; MACINTYRE, I. D.; LOUCKS, W. M. On the trade-off between time and space in optimistic parallel discrete-event simulation. *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, p. 33–42, January 1992.

QUAGLIA, F.; CORTELLESSA, V. Rollback-based parallel discrete event simulation by using hybrid state saving. *Proceedings of the 9th European Simulation Symposium*, p. 275–279, October 1997.

QUAGLIA, F.; CORTELLESSA, V.; CICIANI, B. Tradeoff between sequential and time Warp based parallel simulation. *IEEE Transactions on Parallel and Distributed Systems*, v. 10, n. 8, p. 781–811, August 1999.

QUINN, M. J. *Designing Efficient Algorithms for Parallel Computers.* New York: McGraw-Hill, 1987.

RAJAEL, H.; AYANI, R.; THORELLI, L. The local Time Warp approach to parallel simulations. *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, p. 119–126, 1993.

RANDELL, B. System structure for software fault-tolerance. *IEEE Transactions on Software Engineering*, SE-1, n. 2, p. 220–232, June 1975.

RAYNAL, M.; SINGHAL, M. Logical time: A way to capture causality in distributed systems. *IEEE Computer*, p. 49–56, 1995. Publication Interne.

REED, D. A.; MALONY, A. D.; MCCDREDIE, B. D. Parallel discrete event simulation using shared memory. *IEEE Transactions on Software Engineering*, v. 14, n. 4, p. 541–553, 1988.

REIHER, P. L. et al. Cancellation strategies in optimistic execution systems. *Proceedings of the 1990 Distributed Simulation Conference*, v. 22, p. 112–121, January 1990.

REIHER, P. L.; JEFFERSON, D.; WIELAND, F. Limitation of optimism in the Time Warp operating system. *Proc. 1989 Winter Simulation Conference*, p. 765–770, December 1989.

REYNOLDS, P. F. A shared resource algorithm for distributed simulation. *Proceedings of the 9th Annual Symposium on Computer Architecture*, p. 259–266, April 1982.

RICART, G.; AGRAWALA, A. K. An optimal algorithm for mutual exclusion in computer networks. *Communication of the ACM*, v. 24, n. 1, p. 9–17, January 1981.

RöNNGREN, R.; AYANI, R. Adaptive checkpointing in Time Warp. *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, p. 110–117, 1994.

RöNNGREN, R. et al. A comparative study of state saving mechanisms for Time Warp synchronized parallel discrete event simulation. *IEEE Proceedings of the 29th Annual Simulation Symposium*, p. 5–14, 1996.

ROBINSON, S. Distributed simulation and simulation practice. *Simulation*, v. 81, n. 1, p. 5–13, January 2005.

RUSSELL, D. L. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6, n. 2, p. 183–194, March 1980.

SAMADI, B. *Distributed Simulation.* Tese (Doutorado) — University of California, Los Angeles, 1985.

SANTORO, A. *Semi-Asynchronous Checkpointing for Optimistic Parallel Simulation.* Tese (Doutorado) — Università Degli Studi di Roma "La Sapienza", 2003.

SANTORO, A.; QUAGLIA, F. Communications and network: Benefits from semi-asynchronous checkpointing for Time Warp simulations of a large state pcs model. *Proceedings of the 33nd Conference on Winter Simulation*, p. 1339–1345, 2001.

SCHMIDT, R. et al. Optimal asynchronous garbage collection for RDT checkpointing protocols. *25th International Conference on Distributed Computing Systems*, June 2005.

SCHMUCK, F. *The Use of Efficient Broadcast in Asynchronous Distributed Systems.* Tese (Doutorado) — Cornel University, 1988. TR88-928.

SCHWARZ, R.; MATTERN, F. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, v. 7, n. 3, p. 149–174, 1994.

SHAY, W. *Sistemas Operacionais.* São Paulo: Makron Books, 1996. 337 p.

SIMMONDS, R.; BRADFORD, R.; UNGER, B. Applying parallel discrete event simulation to network emulation. *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, p. 15–22, 2000.

SKOLD, S.; RONNGREN, R. Event sensitive state saving in Time Warp parallel discrete event simulation. *Proceedings of the 28th conference on Winter simulation*, p. 653–660, December 1996.

SOARES, L. F. G. *Modelagem e Simulação Discreta de Sistemas.* São Paulo: IME-USP, 1990. (VII Escola de Computação).

SOKOL, L. M.; BRISCOE, D. P.; WIELAND, A. P. Mtw: A strategy for scheduling discrete events for concurrent execution. *Proceedings of the SCS Multiconference on Distributed Simulation*, p. 34–42, July 1988.

SOLIMAN, H. M. On the selection of the state saving strategy in Time Warp parallel simulations. *Transactions of The Society for Computer Simulation International*, v. 16, n. 1, p. 32–36, 1999.

SPOLON, R. *Um Método para Avaliação de Desempenho de Protocolos de Sincronização Otimistas para Simulação Distribuída.* 247 p. Tese (Doutorado) — Instituto de Física de São Carlos, São Carlos, 2001.

SPOLON, R.; SANTANA, M. J.; SANTANA, R. H. C. Distributed simulation, Time Warp and variants: Taxonomy and performance evaluation issues. *Proceedings of the 13th European Simulation Multiconference*, p. 220–227, 1999.

SRINIVASAN, S. *NPSI Adaptive Synchronization Algorithms for Parallel Discrete Event Simulation.* Tese (Doutorado) — University of Virginia, 1995.

SRINIVASAN, S.; REYNOLDS, P. F. Adaptive algorithms vs. Time Warp: An analytical comparison. *Proceedings of the 27th Conference on Winter Simulation*, p. 666–673, 1995.

SRINIVASAN, S.; REYNOLDS, P. F. Elastic time. *ACM Transactions on Modeling and Computer Simulation*, v. 8, n. 2, p. 103–139, 1998.

STEINMAN, J. S. Speedes: A multiple synchronization environment for parallel discrete event simulation. *International Journal in Computer Simulation*, v. 2, p. 251–286, 1992.

STEINMAN, J. S. Breathing Time Warp. *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, p. 109–118, May 1993.

STEINMAN, J. S. Incremental state saving in speedes using c++. *Proceedings of the 1993 Winter Simulation Conference*, p. 687–696, December 1993.

STROM, R.; YEMINI, S. Optimistic recovery in distributed systems. *ACM Transactions on Computer Sytems*, v. 3, n. 3, p. 204–226, August 1985.

TANENBAUM, A. S. *Modern Operating Systems*. Upper Saddle River: Prentice Hall, 1992. 728 p.

TANENBAUM, A. S. *Distributed Operating Systems*. Upper Saddle River: Prentice Hall, 1995. 648 p.

TAY, S. C.; TEO, Y. M.; KONG, S. T. Speculative parallel simulation with an adaptive throttle scheme. *Proceedings of 11th Workshop on Parallel and Distributed Simulation*, p. 116–123, 1997.

TAYLOR, S. J. E. et al. Distributed simulation and industry: Potentials and pitfalls. *Proceedings of the 2002 Winter Simulation Conference*, p. 688–694, 2002.

TEO, Y. M.; NG, Y. K. Spades/java: Object-oriented parallel discrete-event simulation. *Proceedings of the 35th Annual Simulation Symposium*, p. 245–252, 2002.

TEO, Y. M.; NG, Y. K.; ONGGO, B. S. S. Conservative simulation using distributed-shared memory. *Proceedings of the Sixteenth Workshop on Parallel and Distributed Simulation*, p. 3–10, 2002.

TRIOLA, M. F. *Introdução à Estatística*. Rio de Janeiro: LTC - Livros Técnicos e Científicos Editora S.A., 1999.

VEE, V.-Y.; HSU, W.-J. Pal: A new fossil collector for Time Warp. *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*, p. 35–42, 2002.

VIEIRA, G. M. D. *Estudo Comparativo de Algoritmos Para Checkpointing*. Dissertação (Mestrado) — Universidade de Campinas, Campinas, 2001.

WANG, Y. M. Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Transactions on Computers*, v. 46, n. 4, p. 456–468, April 1997.

WANG, Y. M.; FUCHS, W. K. Lazy checkpoint coordination for bounding rollback propagation. IEEE Symp. on Reliable Distributed Systems, p. 78–85, October 1993.

ZENG, Y.; CAI, W.; TURNER, S. J. Batch based cancellation: A rollback optimal cancellation scheme in Time Warp simulations. *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, p. 78–86, 2004.

ZHANG, J. L.; TROPPER, C. The dependence list in Time Warp. *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*, p. 35–45, 2001.

Apêndice A

Listagem dos Algoritmos Descritos na Tese

A.1 Algoritmo *Sync-and-Stop*

Listagem A.1: Algoritmo *Sync-and-Stop*.

```
{Procedimento utilizado pelo processo responsável pela coordenação
do chekpointing}

procedure Coordenador;
const
  {Tipos de Mensagens para sincronização da operação de Checkpoint}
  IniciaSNS, ConfirmaSNS, RealizaCheckpoint, FinalSNS;
var
  P : array [1..n] of PID; {n = número de processos}
  k : integer;
begin
  {Aguarda intervalo de tempo para realizar o Checkpoint}
  Send (IniciaSNS, Broadcast);
  for k := 1 to n do
    Receive (ConfirmaSNS, P[k]);
  Send (RealizaCheckpoint, Broadcast);
  {Realiza Checkpoint}
  for k := 1 to n do
     Receive (ConfirmaSNS, P[k]);
  Send (FinalSNS, Broadcast);
end;

{Procedimento utilizado pelos demais processos}

procedure Processo (Id : integer);
const
  {Tipos de Mensagens para sincronização da operação de Checkpoint}
  IniciaSNS, ConfirmaSNS, RealizaCheckpoint, FinalSNS
var
  Msg : Message;
```

```
begin
  Receive (Msg);
  if Msg.Kind = IniciaSNS then
  begin
    {Libera os canais de comunicação}
    Send (ConfirmaSNS, Coordenador);
    Receive (RealizaCheckpoint, Coordenador);
    {Realiza Checkpoint}
    Send (ConfirmaSNS, Coordenador);
    Receive (FinalSNS, Coordenador);
  end;
end;
```

A.2 Algoritmo *Chandy-Lamport*

Listagem A.2: Algoritmo *Chandy-Lamport*.

```
{Procedimento utilizado pelo processo responsável pela coordenação
do chekpointing}

procedure Coordenador;
const
  {Mensagens de sincronização da operação de Checkpoint}
  IniciaCL, FinalProcessoCL, FinalCL;
var
  P : PID;
  k : integer;
  M : Message;
begin
  {Aguarda intervalo de tempo para realizar o Checkpoint}
  Send (IniciaCL, {Todos os canais de Saída});
  {Realiza Checkpoint}
  while {não receber IniciaCL de todos os canais de entrada} do
  begin
    Receive (M, P);
    if M <> IniciaCL then
      Save(M);
  end;
  {Receber FinalProcessoCL de todos os canais de entrada}
  Send (FinalCL, {Todos os canais de Saída});
end;

{Procedimento utilizado pelos demais processos}

procedure Processo (Id : integer);
const
  {Mensagens de sincronização da operação de Checkpoint}
  IniciaCL, FinalProcessoCL, FinalCL;
var
  P : PID;
  k : integer;
  M : Message;
begin
  {Após receber a primeira mensagem IniciaCL}
  Send (IniciaCL, {Todos os canais de Saída});
  {Realiza Checkpoint}
  while não receber IniciaCL de todos os canais de entrada do
  begin
    Receive (M, P);
    if M <> IniciaCL then
      Save(M);
  end;
  Send (FinalProcessoCL, Coordenador);
  Receive (FinalCL, Coordenador);
end;
```

A.3 Algoritmos do Padrão ZPF

Listagem A.3: Algoritmo *Fixed Dependency Interval.*

```
procedure FDI (Id : integer);
var
    RV : array [1..n] of integer;
    k : integer;

procedure Send_Message(M : Message; p : PID);
begin
  Send (M + RV, p);
end;

procedure Receive_Message(var M : Message; p : PID);
begin
  if m.RV[p] > RV[p] then
  begin
    RV[Id] := RV[Id] + 1;
    {Realiza Checkpoint forçado}
  end;
  for k := 1 to n do
    RV[k] := max (RV[k], M.RV[k]);
  {Libera mensagem M }
end;

begin
  for k := 1 to n do
     RV[k] := 0;
  RV[Id] := 1;
  {Armazena Checkpoint Inicial}
end;
```

Listagem A.4: Algoritmo *Fixed Dependency After Send*

```
procedure FDAS (Id : integer);
var
    RV : array [1..n] of integer;
    Sent : Boolean;
    k : integer;

procedure Send_Message(M : Message; P : PID);
begin
  Sent := True;
  Send (M + RV, P);
end;

procedure Receive_Message(M : Message; P : PID);
begin
  if Sent and M.RV[p] > RV[p] then
  begin
    Sent := False;
    RV[Id] := RV[Id] + 1;
    {Realiza Checkpoint forçado}
  end;
  for k := 1 to n do
    RV[k] := max (RV[k], M.RV[k]);
  {Libera mensagem M }
end;

begin
  for k := 1 to n do
    RV[k] := 0;
  RV[Id] := 1;
  Sent := False;
  {Realiza Checkpoint Inicial}
end;
```

A.4 Algoritmo *Sync-and-Stop* estendido

Listagem A.5: Algoritmo *Sync-and-Stop* Estendido

```
{Procedimento utilizado pelo processo responsável pela coordenação
 do chekpointing}

procedure Coordenador;
const
  {Tipos de mensagens para sincronização da operação de Checkpoint}
  IniciaSNS, ConfirmaSNS, RealizaCheckpoint, FinalSNS;
var
  P : array [1..n] of PID;
  LVT : array [1..n] of integer;
  GVT : integer;
  RecoveryLine : array [1..n] of integer;
  RL : array [1..n] of integer;
  aux : array [1..n] of integer;
  RecoveryLineSet : set of RecoveryLine;
  TrocouMensagens : array [1..n] of boolean;
  k : integer;
begin
  {Aguarda intervalo de eventos para realizar Checkpoint}
  Send (IniciaSNS, {broadcast});
  RecoveryLine := {último Recovery Line inserido em RecoveryLineSet}
  aux := RecoveryLine;
  for k := 1 to n do
    Receive (ConfirmaSNS + LVT[k] + TrocouMensagens[k], P[k]);
  for k := 1 to n do
    if not TrocouMensagens[k] then
      RecoveryLine[k] := LVT[k];
  /* Substitui o respectivo RecoveryLine em RecoveryLineSet */
  for {Todo RecoveryLine RL pertencente ao RecoveryLineSet} do
    for k := 1 to n do
      if RL[k] = aux[k] then
        RL[k] := RecoveryLine[k];
  RecoveryLineSet := RecoveryLineSet + LVT;
  Send_Message (RealizaCheckpoint, Broadcast);
  for k := 1 to n do
     Receive (ConfirmaSNS, P[k]);
  GVT := LVT[1];
  for k := 2 to n do
     GVT := min (GVT, LVT[k]);
  Send (FinalSNS + GVT, Broadcast);
end;

{Procedimento utilizado pelos demais processos}

procedure ProcessoDaSimulação (Id : integer);
const
  {Tipo das mensagens para sincronização da operação de Checkpoint}
  IniciaSNS, ConfirmaSNS, RealizaCheckpoint, FinalSNS;
```

```
var
  LVT : integer;
  GVT : integer;
  k : integer;
  TrocouMensagens : boolean;

procedure Send_Message (M : message; Id : PID);
begin
  Send(M, Id);
  TrocouMensagens := True;
end;

procedure Receive_Message (M : message; Id : PID);
begin
  Receive(M, Id);
  TrocouMensagens := True;
end;

begin
  Receive (IniciaSNS, Coordenador);
  /* Libera os canais de comunicação */
  LVT := {tempo lógico local da simulação no processo Id}
  Send (ConfirmaSNS + LVT + TrocouMensagens, Coordenador);
  Receive (RealizaCheckpoint, Coordenador);
  {Realiza Checkpoint}
  Send (ConfirmaSNS, Coordenador);
  Receive (FinalSNS + GVT, Coordenador);
  {Realiza Coleta de Fóssil para todo Checkpoint armazenado antes
   do GVT}
  TrocouMensagens := False;
end;
```

A.5 Algoritmo *Chandy-Lamport* estendido

Listagem A.6: Algoritmo *Chandy-Lamport* Estendido

```
{Procedimento utilizado pelo processo responsável pela
 coordenação do chekpointing}

procedure Coordenador;
const
  {Mensagens de sincronização da operação de checkpointing}
  IniciaCL , FinalProcessoCL , FinalCL;
var
  P   : PID;
  k   : integer;
  M   : Message;
  VT  : array [1..Número_de_canais] of integer;
  GVT : integer;
  TrocouMensagens : array [1..n, 1..n] of boolean;
begin
  {Aguarda intervalo de eventos para realizar o Checkpoint}
  Send (IniciaCL + LVT, {Todos os canais de Saída});
  /* Realiza Checkpoint */
  while {não receber IniciaCL de todos os canais} do
  begin
    Receive (M, P);
    if M <> IniciaCL then
      Save(M)
    else
    begin
      VT[M.PID] := M.LVT;
      {Atualiza Linhas de Recuperação}
    end;
  end;
  GVT := {mínimo entre elementos de VT
          e o LVT das mensagens Salvas};
  while {não receber FinalProcessoCL de todos os canais} do
  begin
    Receive (M, p);
    GVT := min (GVT, M.GVT_Local);
  end;
  Send (FinalCL + GVT, {Todos os processos});
end;

{Procedimento utilizado pelos demais processos}

procedure Processo (Id : integer);
const
  {Mensagens de sincronização da operação de checkpointing}
  IniciaCL , FinalProcessoCL , FinalCL;
var
  P : PID;
  k : integer;
```

```
  M : Message;
  VT  : array [1..Número_de_canais] of integer;
  GVT_Local : integer;
  GVT : integer;
begin
  {Após receber a primeira mensagem IniciaCL}
  Send (IniciaCL + LVT, {Todos os canais de Saída});
  /* Realiza Checkpoint */
  while {não receber IniciaCL de todos os canais} do
  begin
    Receive (M, P);
    if M <> IniciaCL then
      Save(M)
    else
      VT[M.PID] := M.LVT;
  end;
  GVT_Local := {mínimo entre elementos de VT
                e o LVT das mensagens Salvas};
  Send (FinalProcessoCL + GVT_Local, Coordenador);
  Receive (FinalCL + GVT, Coordenador);
end;
```

A.6 Tratamento dos *Rollbacks*

Listagem A.7: *Rollback* Solidário Síncrono ou Coordenado

```
procedure RollbackSolidarioSincrono;
const
  {Tipos de mensagens para sincronização da operação de checkpointing}
  RealizaRollback, PontoDeRetorno,
  ConfirmaRollback, FinalRollback;
var
  P : PID;
  Pv : array [1..n] of PID;
  RecoveryLine : array [1..n] of integer;
  RecoveryLineSet : set of RecoveryLine;
  LVT_Retorno : integer;
  M : Message;
  k : integer;
begin
  Receive (M, P);
  if M.Kind = RealizaRollback then
  begin
    RecoveryLine := {identifica RecoveryLine em RecoveryLineSet
                     cujo processo P realizou checkpoint com
                     Relógio Lógico <= M.LVT_Retorno};
    for k := 1 to n do
      Send (PontoDeRetorno + RecoveryLine[k], Pv[k]);
    for k := 1 to n do
      Receive (ConfirmaRollback, Pv[k]);
    for k := 1 to n do
      Send (FinalRollback, Pv[k]);
  end;
end;
```

A.7 Listagem do Código Observador

Listagem A.8: Algoritmo Observador do protocolo *Rollback* Solidário

```
procedure Observador;
var
  P : PID;
  RecoveryLine : array [1..n] of integer;
  RV : array [1..n] of integer;
  GVT : integer;
  RecoveryLineSet : set of RecoveryLine;
  M : array [1..n, 1..n] of integer;
  i,j : integer;

function ExtraiRecoveryLineDeM (var RL : array [1..n] of integer;
        M : array [1..n, 1..n] of integer) : boolean;
var
  i, j : integer;
  Result : boolean;
  Aux : array [1..n] of integer;
begin
  for i := 1 to n do
    Aux[i] := M[i, i];
  Result := True;
  j := 1;
  while (j <= n) and Result do
  begin
    i := 1;
    while (i <= n) and Result do
    begin
      if (i <> j) and (M[i, j] > M[j, j]) then
        Result := False;
      i := i + 1;
    end;
    j := j + 1;
  end;
  if Result then RL := Aux;
  ExtraiRecoveryLineDeM := Result;
end;

begin
  for i := 1 to n do
    for j := 1 to n do
      if i = j then
        M[i, j] := 1
      else
        M[i, j] := 0;
  while {não acontece rollback} do
  begin
    Receive (RV, P);
    for j := 1 to n do
```

```
      M[P, j] := RV[j];
    if ExtraiRecoveryLineDeM (RecoveryLine, M) then
      RecoveryLineSet := RecoveryLineSet + RecoveryLine;
  end;
end;
```

A.8 Método para Obtenção das Linhas de Recuperação

Listagem A.9: Método para a localização de uma linha de recuperação utilizando listas encadeadas

```
type PtrNo = ^No;
     No = record
       VD : array [1..n] of integer;
       Proximo : PtrNo;
     end;

     Listas : array [1..n] of record
                  Inicio : PtrNo;
                  Atual  : PtrNo;
     end;

procedure BuscaLinhaDeRecuperacao
          (var L : Listas; Pid, Checkpoint : integer);
var
  i, j : integer;
  Ok : boolean;
begin
  for i := 1 to n do
    L[i].Atual := L[i].Inicio;
  while ( L[Pid].Atual^.VD[Pid] <> Checkpoint ) do
    L[Pid].Atual = L[Pid].Atual^.Proximo;
  Ok := true;
  i := 1;
  while true do
  begin
    if i <> Pid then
    begin
      for j := 1 to n do
        if (j <> i) and (L[i].Atual^.VD[j] >= L[j].Atual^.VD[j])
        then begin
          L[i].Atual = L[i].Atual^.Proximo;
          Ok := false;
        end;
    end;
    i := i + 1;
    if (i > n) then
      if Ok then break
      else begin
        i := 1;
        Ok := true;
      end;
  end;
end;
```

A.9 *Rollback* Solidário Semi-Síncrono

Listagem A.10: *Rollback* Solidário Semi-Síncrono

```
procedure SolidaryRollbackSemiSincrono;
const
  {Tipos de mensagens para sincronização da operação de checkpointing}
  RealizaRollback , CheckpointDeRetorno;
var
  P : PID;
  Pv : array [1..n] of PID;
  RecoveryLine : array [1..n] of integer;
  RecoveryLineSet : Set of RecoveryLine;
  CL_Retorno : integer;
  M : Message;
  k : integer;
begin
  Receive (M, P);
  if M.Kind = RealizaRollback then
  begin
    RecoveryLine := {identifica RecoveryLine em RecoveryLineSet
                     cujo processo P realizou checkpoint com
                     Relógio Lógico <= CL_Retorno};
    for k := 1 to n do
      Send (CheckpointDeRetorno + RecoveryLine[k], Pv[k]);
  end;
  else
    { Processar Mensagem }
end;
```

A.10 *Rollback* Solidário Sem Observador

Listagem A.11: *Rollback* Solidário Sem Observador

```
procedure SolidaryRollback;
const
  {mensagens para operação do Rollback Solidário sem Observador}
  Normal, RealizaRollback, ConfirmaRollback,
  ConfirmaContinuação, FinalizaRollback;
var
  P   : array [1..n] of PID;
  VD  : array [1..n] of integer;
  LVT : integer;
  Msg : Message;
  Pk  : PID;
  CL  : integer;
  k   : integer;
begin
  Receive (Msg, Pk);
  case Msg.Kind of
    Normal : begin
      if Msg.LVT < LVT then
      begin
        { Restaurar estado tal que timestamp(CL) <= Msg.LVT }
        for k := 1 to n do
          Send (RealizaRollback + VD, P[k]);
        { Aguardar confirmação de todos os outros processos }
        { Montar Matriz M }
        { Identificar Linha de Recuperação }
        for k := 1 to n do
          Send (FinalizaRollback + VD, P[k]);
      end
      else
        { Armazena Evento na Lista de Eventos Futuros }
    end;
    RealizaRollback : begin
      if VD[Pk] >= Msg.VD[Pk] then
      begin
        { Restaurar estado tal que VD[Pk] seja o maior inteiro
          menor do que Msg.VD[Pk] }
        Send (ConfirmaRollback + VD, Pk);
      end
      else begin
        { Realiza Checkpoint }
        Send (ConfirmaContinuação + VD, Pk);
      end;
  end;
end;
```