Arquitetura de *Software* para Recuperação de Falhas Utilizando *Checkpointing* Quase-Síncrono

Ulisses Furquim Freire da Silva

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Arquitetura de *Software* para Recuperação de Falhas Utilizando *Checkpointing* Quase-Síncrono

Ulisses Furquim Freire da Silva[1]

05 de Maio de 2005

Banca Examinadora:

- Profª. Drª. Islene Calciolari Garcia (Orientadora)
- Prof. Dr. Edson Norberto Cáceres
  Departamento de Computação e Estatística — UFMS
- Prof. Dr. Edmundo Roberto Mauro Madeira
  Instituto de Computação — UNICAMP
- Prof. Dr. Luiz Eduardo Buzato (Suplente)
  Instituto de Computação — UNICAMP

[1]**Apoio financeiro do CNPq e da FAPESP (processo número 03/01525-8).**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Bibliotecário: Maria Júlia Milani Rodrigues / CRB8a - 2116

Silva, Ulisses Furquim Freire da

Si38a Arquitetura de software para recuperação de falhas utilizando checkpointing quase-síncrono / Ulisses Furquim Freire da Silva -- Campinas, [S.P. :s.n.], 2005.

Orientadora : Islene Calciolari Garcia

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

1. Tolerância a falha (Computação). 2. Processamento distribuído. 3. Algoritmos. I. Garcia, Islene Calciolari. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Software architecture for fault-recovery using quasi-synchronous checkpointing

Palavras-chave em inglês (Keywords): 1. Fault-tolerant computing. 2. Distributed processing. 3. Algorithms.

Área de concentração: Sistemas distribuídos

Titulação: Mestre em Ciência da Computação

Banca examinadora: Profa. Dra. Islene Calciolari Garcia (IC-UNICAMP)
Prof. Dr. Edson Cáceres (DCT-UFMS)
Prof. Dr. Edmundo R. Madeira (IC-UNICAMP)

Data da defesa: 05/05/2005

Arquitetura de *Software* para Recuperação de Falhas Utilizando *Checkpointing* Quase-Síncrono

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Ulisses Furquim Freire da Silva e aprovada pela Banca Examinadora.

Campinas, 01 de Fevereiro de 2006.

Profª Drª Islene Calciolari Garcia
(Orientadora)

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

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 05 de maio de 2005, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Edson Norberto Cáceres
UFMS

Prof. Dr. Edmundo Roberto Mauro Madeira
IC - UNICAMP

Profa. Dra. Islene Calciolari Garcia
IC - UNICAMP

Agradecimentos

Aos meus pais, Paulo e Regina, pela educação, pelo incentivo ao estudo e pelas lições de vida que foram papel fundamental na minha formação como indivíduo e me permitiram chegar até aqui.

Aos meus irmãos, Raquel e Eduardo, pelos muitos momentos alegres e felizes que compartilhamos e pelo apoio constante.

À minha querida Dorothy, por aparecer na minha vida e torná-la muito mais feliz e especial. O seu amor e apoio foram especialmente importantes na conclusão do trabalho e escrita da dissertação.

À Islene, minha orientadora, pela motivação, confiança e orientação durante o desenvolvimento do trabalho.

Aos grandes amigos, Vinicius Fortuna, Vinícius Pinheiro, Romulo, Luís e Rodrigo Tomita, pela amizade, pelo apoio e companheirismo.

Aos amigos da turma de Ciência da Computação de 1999, em especial, Vinicius, Romulo, Rodrigo Tomita, Sheila, Renata, Flávia e Gilberto, pelas inúmeras experiências compartilhadas.

À minha amiga Cândida, pela amizade, carinho e pelas longas e interessantes conversas sobre os mais variados assuntos.

Aos professores do Instituto de Computação da UNICAMP, em especial, Tomasz, Cid, Arnaldo e Guido, por terem colaborado muito para a minha formação acadêmica.

Aos colegas de competições de programação, em especial, Vinicius, Guilherme, Rodrigo Schmidt, Maurício, Pedro Demasi e Rodolfo, que me ajudaram a despertar um interesse especial pela área de projeto e análise de algoritmos.

Aos membros da banca examinadora, pelas sugestões e pelo incentivo.

Ao CNPq e à FAPESP, pelo apoio financeiro.

Resumo

Um sistema distribuído tolerante a falhas que utilize recuperação por retrocesso de estado deve selecionar os *checkpoints* dos seus processos que serão gravados. Além dessa seleção, definida por um protocolo de *checkpointing*, o sistema precisa realizar uma coleta de lixo, para eliminar os *checkpoints* que se tornam obsoletos à medida que a aplicação executa. Assim, na ocorrência de uma falha, a computação pode ser retrocedida para um estado consistente salvo anteriormente. Esta dissertação discute os aspectos teóricos e práticos de um sistema distribuído tolerante a falhas que utiliza protocolos de *checkpointing* quase-síncronos e algoritmos para a coleta de lixo e recuperação por retrocesso.

Existem vários protocolos de *checkpointing* na literatura, e nesta dissertação foram estudados os protocolos de *checkpointing* quase-síncronos. Esses protocols enviam informações de controle juntamente com as mensagens da aplicação, e podem exigir a gravação de *checkpoints* forçados, mas não necessitam de sincronização ou troca de mensagens de controle entre os processos. Com base nesse estudo, um *framework* para protocolos de *checkpointing* quase-síncronos foi implementado numa biblioteca de troca de mensagens chamada LAM/MPI. Além disso, uma arquitetura de *software* para recuperação de falhas por retrocesso de estado chamada Curupira também foi estudada e implementada naquela biblioteca. O Curupira é a primeira arquitetura de *software* que não precisa de troca de mensagens de controle ou qualquer sincronização entre os processos na execução dos protocolos de *checkpointing* e de coleta de lixo.

Abstract

A fault-tolerant distributed system based on rollback-recovery has to select which checkpoints of its processes are stored. Besides this selection, that is controlled by a checkpointing protocol, the system has to do garbage collection, in order to eliminate the checkpoints that become obsolete while the application executes. The garbage collection is important because checkpoints require the use of storage resources and the storage media always has limited capacity. So, when some fault occurs, the whole distributed computation can be restored to a consistent global state previously stored. This dissertation discusses the practical and theoretical aspects of a fault-tolerant distributed system based on quasi-synchronous checkpointing protocols and also garbage collection and rollback-recovery algorithms.

There are several checkpointing protocols proposed in the literature, and the quasi-synchronous ones were studied in this dissertation. These protocols piggyback control information in the application's messages and can induce forced checkpoints, but don't need any synchronization or exchanging of control messages among the processes. Based on that study, a framework for quasi-synchronous checkpointing protocols was implemented in a message passing library called LAM/MPI. Moreover, a software architecture based on rollback-recovery from faults named Curupira was also studied and implemented in that library. Curupira is the first software architecture that doesn't need the exchanging of control messages or any synchronization among the processes during the execution of the checkpointing and garbage collection protocols.

Conteúdo

Lista de Tabelas

Lista de Figuras

Lista de Algoritmos

Capítulo 1

Introdução

Atualmente há uma tendência de se utilizar grandes sistemas distribuídos como *clusters* e *grids* para resolver problemas cada vez mais complexos. Várias áreas como otimização combinatória, genômica, processamento de imagens, astronomia, mecânica de fluidos e tantas outras têm se beneficiado desse tipo de processamento. Porém, como são utilizados muitos recursos computacionais e as computações geralmente duram um longo tempo, a possibilidade de um falha obrigar o reinício da computação é grande. Assim, mecanismos de tolerância a falhas para aplicações distribuídas estão se tornando cada vez mais importantes.

Aplicações distribuídas são constituídas de vários processos espalhados pelos sistemas computacionais. Uma maneira de oferecer tolerância a falhas é utilizar algum protocolo de *checkpointing* para selecionar os estados dos processos que devem ser gravados, que são chamados de *checkpoints*. Com base nos *checkpoints* gravados, um *checkpoint* global pode ser formado contendo um *checkpoint* de cada processo. No entanto, nem todos os *checkpoints* globais representam estados consistentes da aplicação, pois a troca de mensagens entre os processos cria dependências entre os *checkpoints*. Assim, é preciso que na ocorrência de uma falha a computação seja retrocedida para um *checkpoint* global consistente onde não haja dependências entre quaisquer par de *checkpoints*. E como se deseja minimizar o retrocesso da aplicação, esta deve retomar a sua execução do *checkpoint* global consistente mais recente, também chamado de linha de recuperação.

Na literatura existem vários protocolos de *checkpointing* [11, 13, 14, 16, 17, 26, 31, 49]. Estes permitem que os processos da aplicação selecionem a gravação dos seus *checkpoints*, mas podem forçar a gravação de outros, de modo a garantir que a linha de recuperação progrida juntamente com a computação. Existem basicamente três abordagens adotadas pelos protocolos de *checkpointing*: assíncrona, síncrona e quase-síncrona. Na abordagem assíncrona, a gravação dos *checkpoints* é controlada apenas pela aplicação, e não há garantias de que a linha de recuperação irá progredir. Os protocolos síncronos utili-

zam mensagens para sincronizar os processos no momento que a aplicação seleciona um *checkpoint*, de maneira a formar um *checkpoint* global consistente depois da execução do protocolo. Finalmente, a abordagem quase-síncrona envia informações de controle juntamente com as mensagens da aplicação. Dessa maneira, quando um processo recebe uma mensagem, o protocolo deve avaliar a informação de controle recebida e decidir se deve forçar ou não a gravação de um *checkpoint*.

À medida que a computação progride, vários *checkpoints* já armazenados tornam-se obsoletos, pois não serão utilizados em nenhuma linha de recuperação possível. Assim, como o meio de armazenamento possui capacidade limitada, é necessária uma coleta de lixo que identifique e elimine os *checkpoints* obsoletos durante a execução da aplicação. Dependendo da abordagem adotada pelo protocolo de *checkpointing*, a coleta de lixo pode ser mais simples ou não de ser executada. Os protocolos síncronos garantem a formação de um *checkpoint* global consistente após cada execução do protocolo, que será usado no retrocesso da aplicação no caso de uma falha futura. Assim, os *checkpoints* anteriores a essa linha de recuperação podem ser eliminados, pois não serão usados num eventual retrocesso. Em contrapartida, os protocolos de *checkpointing* que utilizam as abordagens assíncrona ou quase-síncrona podem necessitar de sincronização para realizar a coleta de lixo, pois não há como saber apenas com informação local a cada processo qual é a atual linha de recuperação. As dependências entre os *checkpoints* geradas pela troca de mensagens provocam mudanças constantes na linha de recuperação. Além disso, ainda por causa daquelas dependências, podem existir linhas de recuperação diferentes para grupos de processos falhos diferentes.

As aplicações distribuídas geralmente utilizam alguma biblioteca para realizar a troca de mensagens entre os processos e, atualmente, as principais bibliotecas usadas são a do PVM [3] e alguma implementação do padrão MPI [2]. A biblioteca PVM era muito usada há alguns anos, quando o padrão MPI ainda estava sendo projetado. À medida que o padrão MPI foi evoluindo e sendo implementado, o PVM passou a ser cada vez menos utilizado. Hoje em dia, existem muitas implementações do padrão MPI, que são utilizadas nos mais variados tipos de aplicações distribuídas. Porém, como o padrão MPI não define mecanismos para tolerância a falhas, o PVM ainda é utilizado em aplicações que necessitam de mais garantias na ocorrência de uma falha durante o processamento.

O padrão MPI não define meios para que a aplicação consiga implementar alguma forma de tolerância a falhas. Porém, existem várias implementações do padrão MPI que possuem alguns mecanismos implementados para prover tolerância a falhas [7, 15, 41, 45, 50]. Alguns deles fogem da especificação padrão do MPI e permitem que a aplicação saiba quando uma falha ocorreu. Outros são implementados dentro da biblioteca e de forma transparente para a aplicação. Algumas implementações do MPI possuem protocolos síncronos de *checkpointing* disponíveis, sendo que a maioria deles é alguma variação do

algoritmo proposto por Chandy e Lamport [17]. Há protocolos assíncronos implementados também, mas até onde se conhece, não existe nenhuma implementação de protocolos quase-síncronos em um biblioteca de passagem de mensagens do padrão MPI.

Assim, a primeira contribuição desse projeto de mestrado consiste na implementação de uma infra-estrutura para *checkpointing* quase-síncrono dentro da biblioteca de passagem de mensagens do padrão MPI conhecida como LAM [1]. Esta infra-estrutura foi pensada de maneira que qualquer protocolo quase-síncrono pode ser facilmente implementado e usado durante a execução das aplicações.

Além disso, com base na infra-estrutura para *checkpointing* quase-síncrono, foi projetada e implementada uma arquitetura de *software* para recuperação de falhas dentro da biblioteca LAM. Esta arquitetura de *software*, chamada de CURUPIRA, possui algoritmos para coleta de lixo e recuperação aliados ao protocolo de *checkpointing* quase-síncrono. Os protocolos quase-síncronos que podem ser usados no CURUPIRA se restringem aos que possuem a propriedade RDT [49] (*Rollback-Dependency Trackability*), permitindo que as dependências de retrocesso entre os *checkpoints* sejam capturadas em tempo de execução. O CURUPIRA permite que numa execução sem falhas da aplicação, não sejam necessárias nenhuma mensagem de controle ou nenhuma sincronização para que o protocolo de *checkpointing* quase-síncrono e o algoritmo de coleta de lixo funcionem. Outra vantagem é que o algoritmo de coleta de lixo usado limita a quantidade de armazenamento necessário para guardar os *checkpoints*. Essas vantagens, até onde se conhece, não são encontradas em nenhum outro sistema de recuperação de falhas por retrocesso de estado.

Esta dissertação está organizada da seguinte maneira. No Capítulo 2 são apresentados os conceitos e aspectos teóricos sobre protocolos de *checkpointing*. O Capítulo 3 aborda a teoria sobre os algoritmos de recuperação e de coleta de lixo. No Capítulo 4 uma análise é feita sobre alguns trabalhos relacionados existentes na literatura. Por fim, a infra-estrutura para *checkpointing* quase-síncrono e a arquitetura de *software* CURUPIRA (ambas implementadas no LAM) são apresentadas e analisadas no Capítulo 5.

Capítulo 2

Checkpointing

Em sistemas distribuídos, a atividade conhecida como *checkpointing*, que consiste em selecionar os estados dos processos que devem ser gravados na forma de *checkpoints*, pode ser utilizada tanto para a recuperação de falhas como para depuração distribuída ou mesmo monitorização. Para realizar todas essas tarefas, é necessário que se consiga construir estados globais consistentes da aplicação distribuída, que podem ser usados para avaliar alguma propriedade da computação ou servir para que a aplicação seja reiniciada após uma falha.

Este capítulo apresenta a fundamentação teórica e as abordagens existentes de *checkpointing* aplicado à tolerância a falhas. Primeiramente, será definido o modelo computacional adotado e os conceitos de causalidade entre eventos e consistência de estados serão apresentados. Em seguida, a definição de *checkpoint* global consistente será mostrada, assim como propriedades que podem existir na computação como conseqüência das dependências causadas pela troca de mensagens entre os processos da aplicação. Finalizando o capítulo, serão mostradas as abordagens de *checkpointing* em tolerância a falhas, e aprofundando mais nos protocolos quase-síncronos, que são um dos pontos importantes nessa pesquisa.

2.1 Modelo Computacional

Uma aplicação distribuída é composta por n processos: $P = \{p_0, p_1, \ldots, p_{n-1}\}$. Supõe-se que os processos executam de forma seqüencial e autônoma, não tendo acesso a nenhum tipo de memória compartilhada entre eles. Os processos também não possuem acesso a um relógio global e a sua comunicação é feita exclusivamente pela troca de mensagens.

A execução de uma aplicação distribuída pode ser modelada como uma seqüência de eventos $(e_i^0, e_i^1, \ldots)$, onde e_i^γ representa o γ-ésimo evento executado pelo processo p_i. Os eventos podem ser classificados como internos ou eventos de comunicação. Os eventos

internos têm relação com a execução local do processo, geralmente mudando o seu estado, enquanto os eventos de comunicação compreendem os envios e recepções de mensagens.

A maneira gráfica normalmente utilizada para representar a execução de uma aplicação distribuída é chamada de diagrama espaço-tempo [32]. As linhas horizontais representam a execução dos processos ao longo do tempo, que progride da esquerda para a direita. Os eventos são representados por pontos nessas linhas e as mensagens são representadas por setas com as extremidades conectadas aos eventos de envio e recepção. A Figura 2.1 mostra um diagrama espaço-tempo de um aplicação com três processos.

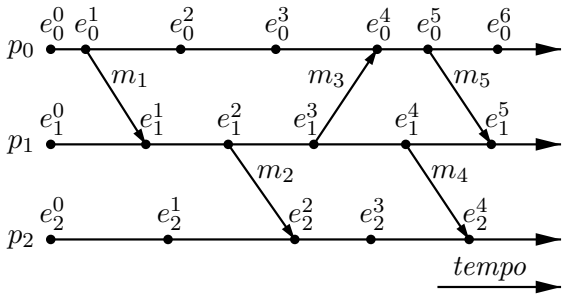


Figura 2.1: Diagrama espaço-tempo.

2.2 Causalidade e Consistência

Ordenar eventos em um sistema distribuído, que não possui um relógio global, é uma tarefa um pouco mais complicada. No entanto, a ordem relativa dos eventos pode ser capturada localmente por meio da relação de precedência causal [32]. Os eventos internos de um processo estão naturalmente ordenados, e as mensagens permitem que eventos de processos diferentes possam ser ordenados, pois o evento de recebimento da mensagem em um processo é sempre precedido pelo evento de envio em outro processo. Assim, capturando as precedências causais entre os eventos é possível tentar dizer se um evento ocorreu antes ou depois de outro na computação distribuída. Com a relação de precedência causal é possível definir *checkpoints* globais consistentes, que são os estados da aplicação distribuída que interessam para fins de recuperação por retrocesso.

2.2.1 Precedência Causal

A relação de precedência causal define uma ordem relativa entre dois eventos de uma computação distribuída [32].

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

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

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

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

A Figura 2.2 ilustra os casos em que ocorre relação de precedência causal. Percebe-se que há uma relação de precedência causal entre quaisquer dois eventos consecutivos de um processo. Desse modo, $e_0^0 \rightarrow e_0^1$, pois o evento e_0^0 ocorre imediatamente antes de e_0^1. Ainda naquela figura, pode-se observar que $e_0^1 \rightarrow e_1^1$, pois a mensagem m é enviada em e_0^1 e recebida em e_1^1. Além disso, como a relação de precedência causal é transitiva, é possível afirmar que $e_0^0 \rightarrow e_1^1$, pois $e_0^0 \rightarrow e_0^1$ e $e_0^1 \rightarrow e_1^1$.

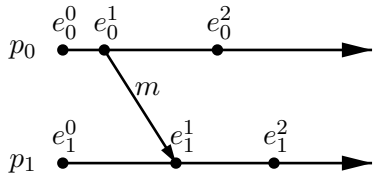


Figura 2.2: Precedência causal entre eventos.

A relação de precedência causal não é reflexiva, pois não permite que um evento preceda ele próprio. Para que isso ocorresse, uma mensagem deveria ser entregue antes de ser enviada. Por outro lado, dois eventos podem ser ditos concorrentes ($e \mid\mid e'$), quando não existe precedência causal entre eles, isto é, $e \not\rightarrow e' \ \wedge \ e' \not\rightarrow e$. Pela existência de eventos concorrentes, a precedência causal não é uma relação que define uma ordenação total dos eventos. Assim, para realizar uma ordenação total dos eventos, alguma ordenação arbitrária para os eventos concorrentes deve ser empregada.

2.2.2 Captura das Precedências Causais

As precedências causais entre eventos de uma computação distribuída podem ser capturadas utilizando-se os relógios lógicos[1] (LC) definidos por Lamport [32]. Esses relógios lógicos são contadores inteiros que são mantidos pelos processos e que são incrementados a cada evento. Na transmissão de uma mensagem, o relógio lógico do processo também é

[1]Em inglês: *Logical Clock*.

enviado, de modo que a precedência causal seja capturada no processo receptor da mensagem. Quando um processo recebe uma mensagem, este deve atualizar o seu relógio lógico para o máximo entre o valor atual e o valor recebido na mensagem mais um. Utiliza-se $LC(e)$ para representar o valor do relógio lógico de um processo após a ocorrência do evento e e $m.LC$ para representar o valor do relógio lógico enviado na mensagem m. A Figura 2.3 ilustra a propagação de relógios lógicos no mesmo diagrama espaço-tempo mostrado anteriormente na Figura 2.1.

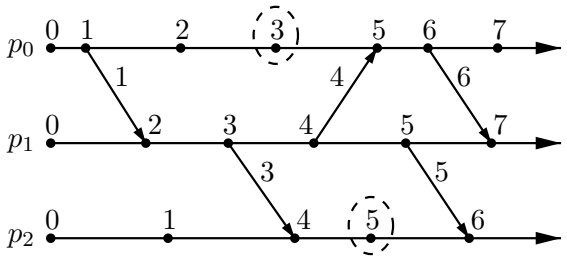


Figura 2.3: Propagação de relógios lógicos.

Pode-se observar da propagação de relógios lógicos e sua relação com a precedência causal que se um evento e precede causalmente um outro evento e', o valor do relógio lógico do evento e é menor que o do evento e' [19]:

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

No entanto, os relógios lógicos não caracterizam a relação de precedência causal, pois dados dois eventos e e e' tais que $LC(e) < LC(e')$, não é possível determinar se $e \rightarrow e'$ ou se $e \mid\mid e'$. Isto pode ser observado na Figura 2.3, onde os eventos e_0^3 e e_2^3 possuem relógios lógicos $LC(e_0^3) = 3$ e $LC(e_2^3) = 5$, e apesar de $LC(e_0^3)$ ser menor que $LC(e_2^3)$, sabe-se que $e_0^3 \mid\mid e_2^3$.

Para que a relação de precedência causal possa ser completamente capturada, é necessário que cada evento tenha mais informação sobre os eventos que o precedem na computação distribuída. Uma maneira de fazer isto consiste em cada processo da computação manter um relógio vetorial[2] [19] (VC). Este relógio é um vetor com n posições, contedo todo o histórico causal de um evento. A entrada $VC[i]$ do processo p_i deve ser incrementada a cada evento e o relógio vetorial do processo deve ser enviado juntamente com todas as mensagens. Assim, quando um processo recebe uma mensagem, este atualiza as entradas do seu relógio vetorial com o maior valor entre o recebido na mensagem e o já armazenado. Utiliza-se $VC(e)$ para representar o relógio vetorial do evento e e $m.VC$

[2]Em inglês: *Vector Clock*.

para o relógio vetorial enviado na mensagem m. A Figura 2.4 apresenta a propagação de relógios vetoriais para o mesmo diagrama espaço-tempo da Figura 2.3.

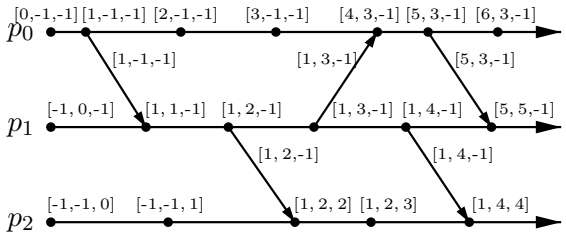


Figura 2.4: Propagação de relógios vetoriais.

Usando relógios vetoriais é possível distingüir se dois eventos em uma computação distribuída são concorrentes ou se um deles precede o outro. Esta propriedade está descrita nas seguintes relações entre precedência causal e relógios vetoriais:

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

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

Considerando os mesmos eventos e_0^3 e e_2^3 já mencionados anteriormente, mas observando os seus relógios vetoriais na Figura 2.4 dessa vez, conclui-se que os eventos são concorrentes. Essa conclusão pode ser feita usando-se as relações entre a precedência causal e os relógios vetoriais já vistas. Percebe-se que $VC(e_0^3)[0] = 3$ é maior que $VC(e_2^3)[0] = 1$, portanto $e_0^3 \not\rightarrow e_2^3$. Além disso, como $VC(e_2^3)[2] = 3$ é maior que $VC(e_0^3)[2] = 0$, pode-se dizer que $e_2^3 \not\rightarrow e_0^3$. Assim, como $e_0^3 \not\rightarrow e_2^3$ e $e_2^3 \not\rightarrow e_0^3$, é possível afirmar que $e_0^3 \ || \ e_2^3$.

2.2.3 Cortes Consistentes

Um corte de uma computação distribuída é um conjunto formado por prefixos das seqüências de eventos executados por cada processo [19]. O conjunto dos últimos eventos de cada processo em um corte é chamado de fronteira do corte. Um estado global é um conjunto dos estados locais de cada um dos processos e é determinado pelos estados dos processos na fronteira de um corte. Quando um corte $\mathcal{C}$ está contido em um corte $\mathcal{C}'$ ($\mathcal{C} \subset \mathcal{C}'$), pode-se dizer que $\mathcal{C}$ está no passado de $\mathcal{C}'$ e, dualmente, que $\mathcal{C}'$ está no futuro de $\mathcal{C}$.

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

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

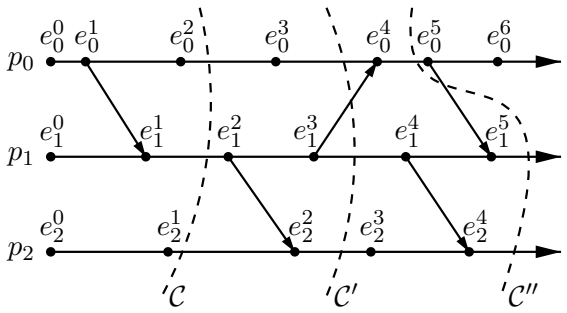


Figura 2.5: Cortes consistentes e inconsistente.

Nem todo corte define um estado consistente da aplicação distribuída. Um corte consistente não pode conter a recepção de uma mensagem, mas não incluir o seu envio. Desse modo, um estado consistente precisa conter todas as precedências causais dos eventos nele contidos. Um estado global é consistente se for gerado pela fronteira de um corte consistente.

No diagrama espaço-tempo da Figura 2.5 existem três cortes representados pelas linhas tracejadas. O critério visual para que um corte seja consistente está no fato de todas as mensagens que atravessam a linha do corte terem o evento de envio do lado esquerdo da linha e o evento de recepção do lado direito. Caso contrário, o corte é considerado inconsistente. Assim, pode-se concluir que os cortes $\mathcal{C}$ e $\mathcal{C}'$ são consistentes e o corte $\mathcal{C}''$ é inconsistente.

2.3 Padrão de *Checkpoints* e Mensagens

Em uma execução de uma aplicação distribuída podem acontecer muitos eventos. Dessa forma, é natural que os estudos relizados em sistemas distribuídos se atenham apenas a alguns eventos durante a computação distribuída. Na área de *checkpointing*, os eventos de interesse em uma computação distribuída são a gravação dos estados dos processos, os *checkpoints*, e os eventos de envio e recepção de mensagens.

Normalmente, os *checkpoints* são salvos em memória estável, de modo que o seu conteúdo possa ser recuperado após uma falha no processamento. Estes *checkpoints* são chamados de estáveis. O estado corrente de cada um dos processos durante a computação também pode ser considerado um *checkpoint*, que é chamado de volátil. Considera-se que os processos gravam um *checkpoint* inicial imediatamente após o início da computação, de modo que o estado global formado por esses *checkpoints* represente o estado consistente inicial da aplicação distribuída.

O conjunto formado pelos *checkpoints* selecionados e as dependências causadas entre eles pela troca de mensagens durante a execução da aplicação gera um padrão de checkpoints e mensagens[3] (CCP). Os CCPs são muito usados para estudar protocolos de *checkpointing*, sendo utilizados em provas e demonstrações, pois propriedades do conjunto de *checkpoints* e mensagens representadas ficam melhor evidenciados. Os CCPs são representados graficamente por diagramas espaço-tempo que destaquem os eventos de envio e recepção de mensagens, bem como os eventos de gravação de *checkpoints*.

A Figura 2.6 representa um padrão de *checkpoints* e mensagens de uma computação distribuída por meio de um diagrama espaço-tempo. Os *checkpoints* estáveis são os quadrados pretos, enquanto os *checkpoints* voláteis são os triângulos pretos.

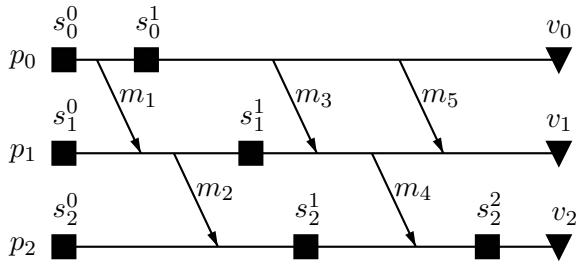


Figura 2.6: Padrão de *checkpoints* e mensagens.

De modo geral, o γ-ésimo *checkpoint* de um processo p_i é representado por c_i^γ, e deve corresponder a algum evento interno $e_i^{\gamma'}$ com $\gamma \leq \gamma'$. Caso o *checkpoint* seja estável, este pode ser representado por s_i^γ, e no caso de ser o *checkpoint* volátil do processo p_i, ele é representado por v_i. O índice do último *checkpoint* estável de p_i é dado por $last(i)$, e o *checkpoint* $s_i^{last(i)}$ também é representado como s_i^{last}. Para simplificar o uso de *checkpoints* estáveis e voláteis, pode-se definir o *checkpoint* c_i^γ de acordo com a seguinte regra:

$$c_i^\gamma = \begin{cases} s_i^\gamma, & \gamma \leq last(i); \\ v_i, & \gamma = last(i) + 1. \end{cases} \tag{2.4}$$

Dois *checkpoints* consecutivos em um mesmo processo determinam um intervalo entre *checkpoints*, que abstrai os eventos executados entre esses dois *checkpoints*. Dessa forma, o intervalo entre *checkpoints* I_i^γ representa o conjunto de eventos entre os *checkpoints* $c_i^{\gamma-1}$ e c_i^γ, incluindo $c_i^{\gamma-1}$ mas excluindo c_i^γ. As mensagens enviadas e recebidas nesses intervalos geram dependências entre os *checkpoints* de processos diferentes. A Figura 2.7 mostra o intervalo entre *checkpoints* I_0^2 como exemplo.

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

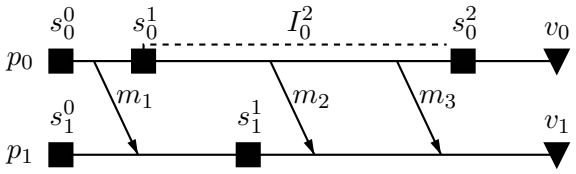


Figura 2.7: Intervalo entre *checkpoints*.

2.4 *Checkpoints* Globais Consistentes

Um *checkpoint* global é formado por um *checkpoint* de cada processo da aplicação distribuída e representa um estado global consistente da aplicação caso ele esteja associado a um corte consistente. Assim, os *checkpoints* de um *checkpoint* global consistente não podem ter dependências causais entre eles.

Definição 2.3 ***Checkpoint Global Consistente*** — *Um checkpoint global C representa um estado consistente da computação distribuída se, e somente se,*

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

A Figura 2.8 ilustra o fato de que *checkpoints* globais consistentes só possuem *checkpoints* concorrentes entre si. O *checkpoint* global C é consistente, enquanto o C' é inconsistente, pois s_0^0 e s_1^1 pertencem a C', mas $s_0^0 \rightarrow s_1^1$. Além disso, o estado global de C' contém a recepção da mensagem m_1, mas não o seu envio.

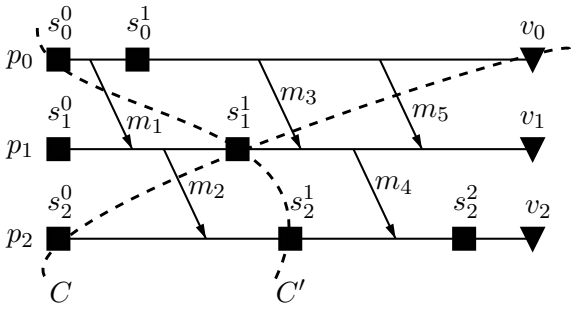


Figura 2.8: *Checkpoint* global consistente e inconsistente.

É possível realizar algumas operações sobre *checkpoints* globais. As mais importantes são a união e a intersecção de *checkpoints* globais. A união de *checkpoints* globais consiste no *checkpoint* global formado pelos *checkpoints* mais recentes de cada processo. A intersecção de *checkpoints* globais também é um *checkpoint* global, mas é formado pelos

checkpoints mais antigos de cada processo. Uma propriedade interessante da união e da intersecção de *checkpoints* globais é que a união ou a intersecção de dois *checkpoints* globais consistentes também resultam em um *checkpoint* global consistente [42]. A Figura 2.9 ilustra essas operações, mostrando o resultado da união e da intersecção dos *checkpoints* globais C e C'.

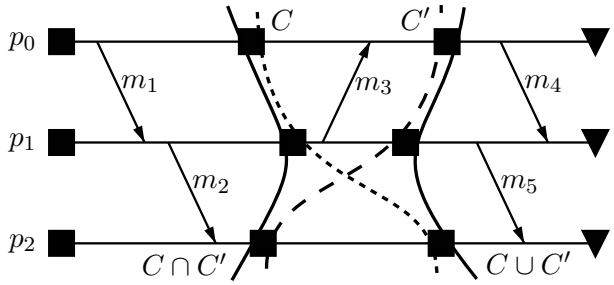


Figura 2.9: União e intersecção de *checkpoints* globais.

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

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

Definição 2.5 ***Intersecção de Checkpoints Globais*** — *A intersecção de dois checkpoints globais C e C' é dada por:*

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

2.5 Caminhos em Ziguezague

A relação de precedência causal não é suficiente para determinar se um conjunto de *checkpoints* pode fazer parte de um mesmo *checkpoint* global consistente. Por exemplo, na Figura 2.10, os *checkpoints* s_0^1 e s_2^0 não possuem dependência causal entre si, mas não podem fazer parte de um mesmo *checkpoint* global consistente.

Netzer e Xu [37] determinaram as condições necessárias e suficientes para que um conjunto de *checkpoints* possa ser usado na formação de um *checkpoint* global consistente. As condições estão relacionadas a caminhos formados entre os *checkpoints* pelas mensagens

da aplicação. Esses caminhos são chamados de caminhos em ziguezague e um conjunto de *checkpoints* poderá fazer parte do mesmo *checkpoint* global consistente caso não haja caminhos em ziguezague entre quaisquer dois *checkpoints* do conjunto.

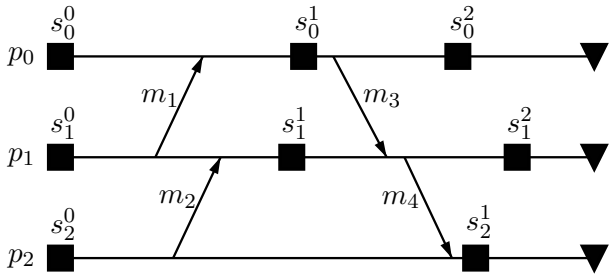


Figura 2.10: Exemplos de caminhos em ziguezague.

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

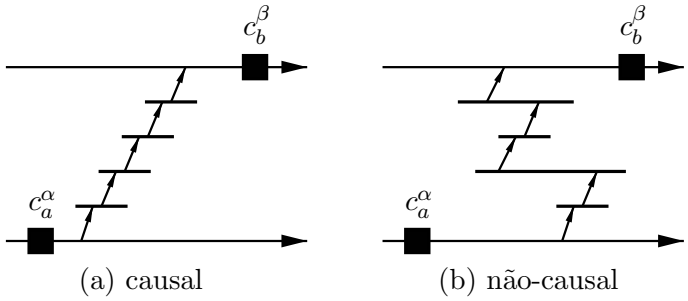
(i) p_a *envia* m_1 *depois de* c_a^α*;*

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

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

Existem dois tipos de caminhos em ziguezague: os causais, chamados de caminhos-C e os não-causais, chamados de caminhos-Z [13]. Um caminho em ziguezague é causal se a recepção de toda mensagem $m_i, 1 \leq i < k$, ocorre sempre antes do envio de m_{i+1}, como ilustrado Figura 2.11 (a). Um caminho em ziguezague é não-causal se a recepção de alguma mensagem $m_i, 1 \leq i < k$, ocorre após o envio de m_{i+1}, como é mostrado na Figura 2.11 (b). Com base nessas definições é possível observar na Figura 2.10 um caminho em ziguezague causal composto pela seqüência de mensagens $[m_3, m_4]$ e um outro não-causal formado por $[m_2, m_1]$. Caminhos em ziguezague definem dependências entre *checkpoints* que são chamadas de causais e não-causais, que são geradas por caminhos-C e caminhos-Z, respectivamente.

Dada a definição de caminho em ziguezague, é possível que haja um caminho em ziguezague não-causal de um determinado *checkpoint* para ele mesmo. Tal caminho é chamado de ciclo-Z, e o *checkpoint* em questão é chamado de inútil, pois não poderá ser usado em nenhum *checkpoint* global consistente [37]. Na Figura 2.10 pode-se observar que o *checkpoint* s_1^1 é inútil, pois a seqüência de mensagens $[m_2, m_4]$ corresponde a um ciclo-Z, que liga s_1^1 a ele mesmo.



(a) causal (b) não-causal

Figura 2.11: Tipos de caminhos em ziguezague.

2.6 Rastreando as Dependências entre os *Checkpoints*

Um mecanismo transitivo pode ser usado para capturar as dependências entre os *checkpoints* durante a execução de uma aplicação. Cada processo p_i deve manter um vetor de dependências[4] (DV) [49]. Esses vetores de dependência são muito semelhantes aos relógios vetoriais vistos anteriormente e possuem n entradas, tal que a entrada $DV[i]$ representa o intervalo entre *checkpoints* corrente do processo p_i, enquanto as demais entradas $DV[j]$ representam o índice do último intervalo entre *checkpoints* de p_j que p_i teve conhecimento.

Os processos devem manter o seu vetor de dependências e também colocá-lo nas suas mensagens enviadas. Desse modo, ao receber uma mensagem, o processo deve atualizar o seu vetor de dependências colocando em cada entrada o maior valor entre o recebido na mensagem e o mantido pelo processo. Cada vez que o processo p_i grava um *checkpoint*, a entrada $DV[i]$ deve ser incrementada, representando o início de um novo intervalo entre *checkpoints*. Além disso, o vetor de dependências deve ser gravado juntamente com cada *checkpoint*, pois possui todas as dependências daquele *checkpoint* capturadas durante a execução. Utiliza-se $DV(c)$ para representar o vetor de dependências salvo juntamente com o *checkpoint* c e $m.DV$ para representar o vetor de dependências enviado com a mensagem m. A Figura 2.12 ilustra a propagação de vetores de dependência, destacando aqueles que são salvos com os *checkpoints* e os enviados com as mensagens.

Os vetores de dependências possuem toda a informação causal de um *checkpoint*. Assim, pode-se estabelecer uma relação entre a precedência causal e os vetores de dependência dos *checkpoints* gravados numa execução de uma aplicação distribuída:

$$c_a^\alpha \rightarrow c_b^\beta \iff \alpha < DV(c_b^\beta)[a] \tag{2.5}$$

$$c_a^\alpha \rightarrow c_b^\beta \iff DV(c_a^\alpha)[a] < DV(c_b^\beta)[a] \tag{2.6}$$

[4]Em inglês: *Dependency Vector*.

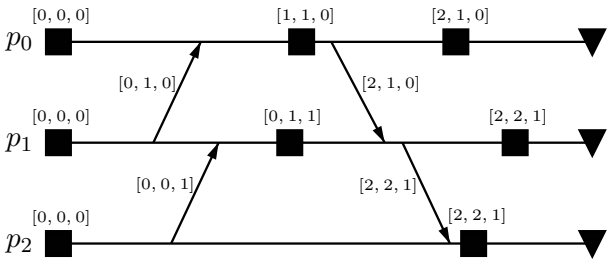


Figura 2.12: Propagação de vetores de dependência.

2.7 Protocolos de *Checkpointing*

Os protocolos de *checkpointing* podem ser classificados segundo suas abordagens em: assíncronos, síncronos e quase-síncronos. Os protocolos assíncronos permitem que os processos selecionem *checkpoints* de maneira completamente independente. No entanto, essa liberdade na seleção dos *checkpoints* apresenta uma limitação no contexto de recuperação de falhas por retrocesso de estado detectado por Randell [40]. Em um cenário denominado efeito dominó, uma aplicação distribuída pode ter que retroceder ao seu estado inicial após a ocorrência de uma falha, apesar de ter gravado *checkpoints* ao longo de sua execução, como está exemplificado na Figura 2.13. Nesta figura, a falha do processo p_0 implica que ele deve ser retrocedido para o *checkpoint* s_0^2. Porém, os *checkpoints* s_0^2 e v_1 não formam um *checkpoint* global consistente, pela existência da mensagem m_4. Dessa forma, p_1 deve ser retrocedido para o *checkpoint* s_1^1. No entanto, mais uma vez, o *checkpoint* global formado por *checkpoints* dos dois processos, s_0^2 e s_1^1, não é consistente, pela existência da mensagem m_3. Assim, p_0 deve ser retrocedido novamente e essa propagação de retrocessos continua até que os dois processos são retrocedidos para os seus *checkpoints* iniciais.

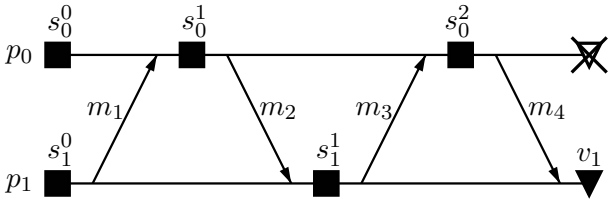


Figura 2.13: Efeito dominó.

Os protocolos síncronos utilizam mensagens de controle de forma a sincronizar a gravação dos *checkpoints* de cada processo e a formar um *checkpoint* global consistente a cada execução do protocolo. Além de utilizar mensagens de controle, muitas vezes

é preciso bloquear a comunicação até o término do protocolo, o que pode prejudicar o desempenho de toda a computação. Por outro lado, os protocolos quase-síncronos são um compromisso entre as outras duas abordagens, pois os processos podem selecionar *checkpoints* livremente e não há o envio de mensagens de controle. Porém, os protocolos quase-síncronos podem forçar os processos a gravarem alguns *checkpoints*, de modo a evitar o efeito dominó e a garantir a existência de *checkpoints* globais consistentes além daquele representado pelo estado inicial da aplicação.

As seções a seguir irão apresentar as principais características e alguns exemplos de protocolos síncronos e quase-síncronos. Os protocolos da abordagem assíncrona não serão apresentados, pois não oferecem garantias de formação de *checkpoints* globais consistentes, que é um requisito importante na recuperação de falhas por retrocesso de estado.

2.7.1 Protocolos Síncronos

Em protocolos de *checkpointing* síncronos são usadas mensagens de controle para sincronizar os processos na criação de um *checkpoint* global consistente. A execução do protocolo é ativada sempre que um processo deseja gravar um *checkpoint* local. Desse modo, a recuperação fica simplificada, pois na ocorrência de uma falha a aplicação voltará a executar a partir do último *checkpoint* global consistente gravado.

Um algoritmo simples que bloqueia a comunicação da aplicação durante a sua execução está ilustrado na Figura 2.14. Este protocolo consiste no processo que quer gravar um *checkpoint* local coordenar o processo de criação do *checkpoint* global consistente, gravando um *checkpoint* provisório e enviando uma mensagem de requisição para os demais processos. Ao receberem a mensagem de requisição, os outros processos devem gravar *checkpoints* locais provisórios e enviar uma mensagem de confirmação para o processo coordenador. De posse de todas as confirmações, o coordenador pode então mandar uma mensagem de finalização do protocolo para os demais processos e tornar permanente o seu *checkpoint* local. Os outros processos, por sua vez, ao receberem a mensagem de finalização devem tornar permanentes os *checkpoints* provisórios que foram salvos anteriormente. Depois disso, a comunicação pode ser reestabelecida e a aplicação volta a executar normalmente. Nesse mesmo algoritmo que foi descrito, o papel de coordenador também pode ser exercido por um processo externo à aplicação distribuída.

A sincronização de todos os processos da aplicação por meio da troca de mensagens de controle e a suspensão da comunicação durante a execução do protocolo de *checkpointing* pode atrasar muito o andamento da computação. Assim, alguns protocolos, como o de Koo e Toueg [31], minimizam o grupo de processos que necessitam de coordenação para gravarem os seus *checkpoints*. O protocolo de Koo e Toueg funciona de forma recursiva, onde cada processo coordena a gravação dos *checkpoints* dos processos que se comunicaram com ele no último intervalo entre *checkpoints* [31].

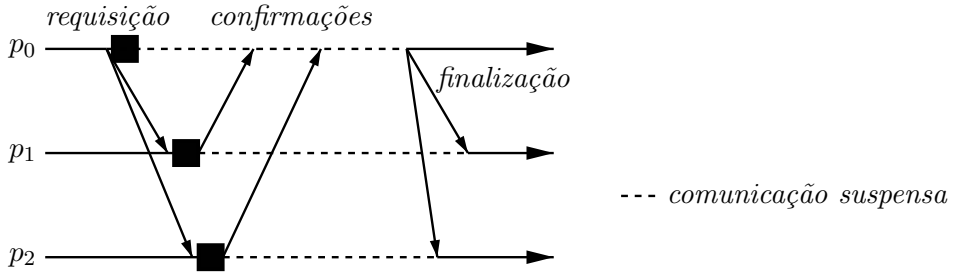


Figura 2.14: Exemplo de protocolo síncrono.

Os algoritmos de *checkpointing* síncronos bloqueantes suspendem a comunicação entre os processos para que as mensagens da aplicação não criem dependências entre os *checkpoints* que estão sendo gravados. Assim, algoritmos de *checkpointing* síncronos não-bloqueantes devem impedir a formação de dependências entre os *checkpoints* que irão formar o novo *checkpoint* global consistente da aplicação. O Algoritmo 2.1 mostra uma variação do protocolo síncrono não-bloqueante de Chandy e Lamport [17]. Este algoritmo resolve o problema de criação de dependências entre os *checkpoints* utilizando canais de comunicação FIFO confiáveis e fazendo com que os processos enviem uma mensagem de controle especial pelos seus canais de comunicação quando gravam um *checkpoint*. Dessa forma, sempre que um processo recebe uma dessas mensagens, ele grava o seu *checkpoint* e as outras mensagens daquele tipo que forem recebidas poderão ser usadas para salvar o estado dos canais de comunicação.

O Algoritmo 2.1 grava um *snapshot* consistente da aplicação, que é composto pelo estado de cada um dos processos e também pelo estado dos canais de comunicação. Nesse protocolo, os processos após gravarem os seus *checkpoints* devem guardar a seqüência de mensagens recebidas de cada um dos outros processos até receber a mensagem de requisição de gravação de *checkpoint* de cada um deles. O algoritmo termina em cada processo quando este já salvou o seu *checkpoint* e recebeu a mensagem de requisição de gravação de *checkpoint* daquela instância de todos os demais processos, o que signifca que todos os canais de comunicação também já foram salvos.

2.7.2 Protocolos Quase-Síncronos

Os protocolos quase-síncronos apresentam as vantagens de autonomia na seleção de *checkpoints* dos protocolos assíncronos, e ao mesmo tempo garantem que o efeito dominó não vai ocorrer, forçando a gravação de outros *checkpoints* dos processos. Os *checkpoints* selecionados pelos processos da aplicação são chamados de básicos, ao passo que aqueles gravados de forma forçada pelo protocolo são chamados de forçados. Esses protocolos

Algoritmo 2.1 Variação do protocolo síncrono de Chandy e Lamport em um processo p_i.

Estruturas de Dados

```
1: Var
2:    instância: inteiro                                  {instância de execução do protocolo}
3:    numcanais: inteiro                                  {número de canais já salvos}
4:    canal: array[0 ... n − 1] of conjunto de mensagens  {para gravar o estado dos canais}
```

Inicialização {inicialização realizada em todos os processos}

```
1: instância ← 0
```

Ao gravar *checkpoint*

```
1: instância ← instância +1
2: envia (CKPT, instância) para todos                     {requisição para gravar checkpoint}
3: numcanais ← 1
4: for k ← 0 to n − 1 do
5:    canal[k] ← ∅
```

Ao receber (*CKPT*, *instância_req*) do processo p_j

```
1: if instância_req > instância then                      {testa se instância_req = instância+1}
2:    grava checkpoint                                    {evento anterior, o qual envia mensagens de controle}
3: grava canal[j] junto com o checkpoint
4: numcanais ← numcanais +1
5: continua execução normal
```

Ao receber mensagem m **do processo** p_j

```
1: if numcanais < n then                                  {se estamos gravando o snapshot global ainda}
2:    canal[j] ← canal[j] ∪ m
3: entrega(m)
```

enviam informações de controle junto com as mensagens da aplicação. Assim, quando uma mensagem é recebida por um processo, o protocolo quase-síncrono deve analisar a informação de controle e decidir se deve ou não gravar um *checkpoint* forçado, como pode-se observar na Figura 2.15. Nos diagramas espaço-tempo de protocolos quase-síncronos, os *checkpoints* básicos são representados por quadrados pretos e os forçados por quadrados hachurados.

Segundo os conceitos de caminhos em ziguezague e ciclos-Z [37], Manivannan e Singhal dividiram os padrões de *checkpoints* e mensagens e os seus protocolos em quatro classes distintas: *Partially Z-Cycle Free* (PZCF), *Z-Cycle Free* (ZCF), *Z-Path Free* (ZPF) e *Strictly Z-Path Free* (SZPF) [36]. Estas classes respeitam a relação de continência ilustrada na Figura 2.16.

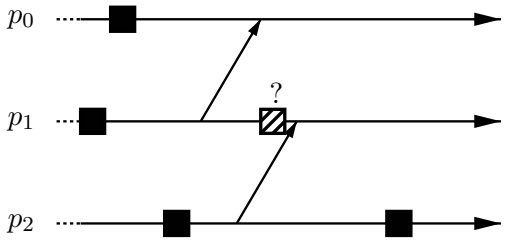


Figura 2.15: Possível indução de um *checkpoint* forçado em um protocolo quase-síncrono.

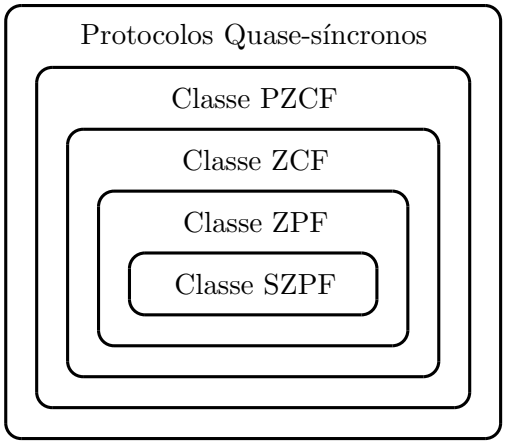


Figura 2.16: Classes de protocolos quase-síncronos.

Classe PZCF

Os protocolos dessa classe não garantem a inexistência de ciclos-Z, mas se esforçam para evitá-los. As principais diferenças entre os protocolos dessa classe são os tipos de dependência que eles quebram ao forçarem a gravação de *checkpoints* [36]. No entanto, a presença de ciclos-Z nos padrões de *checkpoints* e mensagens e, conseqüentemente, de *checkpoints* inúteis tornam os protocolos dessa classe pouco interessantes de serem utilizados na prática. Além disso, com a existência de *checkpoints* inúteis nos padrões de *checkpoints* e mensagens, há a possibilidade de ocorrência do efeito dominó. Assim, o estudo dos protocolos dessa classe não foi alvo dessa dissertação.

Classe ZCF

Nessa classe de protocolos, os padrões de *checkpoints* e mensagens estão isentos de ciclos-Z, ou seja, todos os *checkpoints* são úteis e podem ser usados em algum *checkpoint* global consistente.

A maioria dos protocolos ZCF utiliza índices semelhantes aos relógios lógicos de Lamport [32] e por esse motivo são comumente chamados de protocolos baseados em índice[5]. Um dos primeiros protocolos ZCF foi proposto por Briatico, Ciuffoletti e Simoncini (BCS) [16]. Cada processo mantém um relógio lógico que é enviado juntamente com todas as mensagens e é incrementado cada vez que o processo grava um *checkpoint*, seja este básico ou forçado. Além disso, toda vez que uma mensagem é recebida, o valor do relógio lógico da mensagem é verificado, e caso seja maior que o valor do relógio lógico local, um *checkpoint* forçado é gravado e o relógio lógico local é atualizado para o valor do relógio lógico da mensagem. Dessa maneira, todas as precedências causadas por mensagens anteriores que foram enviadas com o valor do relógio lógico menor são quebradas. A Figura 2.17 mostra um padrão de *checkpoints* e mensagens gerado pelo protocolo BCS, destacando os valores dos relógios lógicos levados pelas mensagens e aqueles gravados junto com os *checkpoints* estáveis.

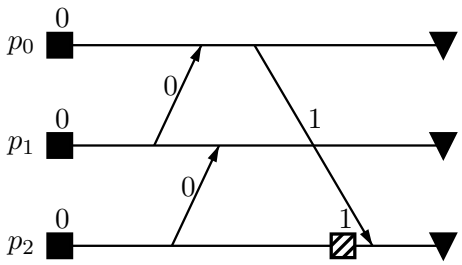


Figura 2.17: Padrão de *checkpoints* e mensagens do protocolo BCS.

Várias otimizações foram propostas para o BCS, e uma das mais simples consiste em evitar gravar um *checkpoint* forçado quando não foi enviada nenhuma mensagem naquele intervalo entre *checkpoints*. Desse modo, o relógio lógico local ainda é atualizado, mas o *checkpoint* forçado não precisa ser gravado pois nenhuma mensagem com o relógio lógico menor que o da mensagem recebida foi enviada no intervalo e, portanto, nenhuma dependência precisa ser quebrada. Essa e algumas outras otimizações foram apresentadas pela primeira vez num protocolo chamado BQF [14]. Porém, este protocolo é muito custoso, pois controla a indução de *checkpoints* forçados propagando matrizes de relógios.

Classe ZPF

Os protocolos dessa classe quebram todos os caminhos-Z que não estão duplicados causalmente. Um caminho-Z entre dois *checkpoints* está duplicado causalmente quando também

[5]Em inglês: *index-based*.

existe um caminho-C entre este mesmo par de *checkpoints* [11, 12, 13]. Na Figura 2.18 percebe-se que o caminho causal μ duplica o caminho não-causal ζ entre os *checkpoints* c_a^α e c_b^β.

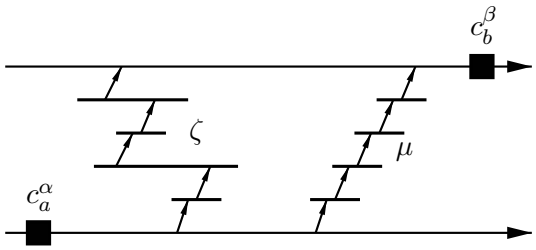


Figura 2.18: Duplicação causal.

Todos os *checkpoints* em um padrão de *checkpoints* e mensagens gerado por um protocolo da classe ZPF são úteis. Dessa forma, os protocolos dessa classe não sofrem do efeito dominó. Além disso, todas as dependências entre os *checkpoints* são causais, o que permite a construção de algoritmos eficientes para a formação de *checkpoints* globais consistentes [49].

Um exemplo de protocolo dessa classe é o FDI [49] (*Fixed-Dependency-Interval*). Nesse protocolo, os processos capturam as dependências entre os *checkpoints* propagando vetores de dependência, como visto anteriormente na Seção 2.6. Além disso, quando uma mensagem é recebida e esta traz a informação de uma nova dependência causal, um *checkpoint* forçado é gravado para que todos os caminhos em ziguezague não-causais sejam duplicados. A Figura 2.19 mostra um padrão de *checkpoints* e mensagens gerado pelo protocolo FDI. Pode-se observar que um *checkpoint* forçado é induzido em p_1 antes de m_1, mas não antes de m_3. Além disso, percebe-se que o caminho-Z $[m_3, m_2]$ está duplicado causalmente pelo caminho-C $[m_1, m_2]$.

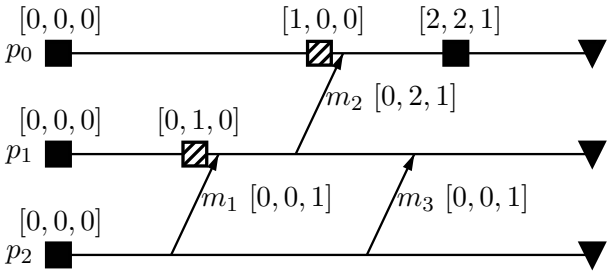


Figura 2.19: Padrão de *checkpoints* e mensagens gerado pelo protocolo FDI.

Classe SZPF

Esta classe de protocolos possui uma condição ainda mais restrita que a classe ZPF. Os padrões de *checkpoints* e mensagens dos protocolos dessa classe possuem apenas dependências causais entre os *checkpoints*, ou seja, todas as dependências não-causais são quebradas. Dessa forma, não existem *checkpoints* inúteis, pois um *checkpoint* não pode preceder causalmente a si próprio, e os protocolos dessa classe não estão sujeitos ao efeito dominó.

Os protocolos dessa classe costumam ser muito simples, pois a decisão de induzir a gravação de um *checkpoint* forçado está mais relacionada à ordem dos eventos de envio e entrega de mensagens dentro de um intervalo entre *checkpoints* do que com as dependências entre os *checkpoints*. Por este motivo, os protocolos dessa classe são chamados de baseados em modelo[6] [22].

Para que todas as dependências entre os *checkpoints* sejam causais, os protocolos SZPF induzem a gravação de *checkpoints* forçados de modo que todos os eventos de recepção de mensagem precedam os eventos de envio de mensagem em um mesmo intervalo entre *checkpoints*. Wang identificou quatro protocolos SZPF: *Checkpoint-After-Send* (CAS), *Checkpoint-Before-Receive* (CBR), *Checkpoint-After-Send-Before-Receive* (CASBR) e *No-Receive-After-Send* (NRAS) [49], ilustrados na Figura 2.20. Estes protocolos são muito simples, mas induzem um número muito grande de *checkpoints* forçados para quebrar todos os caminhos-Z existentes, o que pode prejudicar a execução da computação distribuída.

2.7.3 *Rollback-Dependency Trackability*

Wang identificou uma propriedade chamada de RDT (*Rollback-Dependency Trackability*) nos protocolos das classes ZPF e SZPF [49]. Esta propriedade consiste em conseguir capturar todas as dependências entre os *checkpoints* em tempo de execução, o que permite soluções simples para a determinação de *checkpoints* globais consistentes que incluem um grupo de *checkpoints*. Os protocolos RDT induzem um *checkpoint* forçado sempre que uma dependência não rastreável em tempo de execução puder ser formada pelo padrão de *checkpoints* e mensagens da aplicação [13, 49]. Os nomes ZPF e SZPF são usados mais em comparações entre as classes de protocolos quase-síncronos, enquanto o termo RDT é normalmente usado para a definição, para a propriedade, para o padrão de *checkpoints* e mensagens e para os protocolos dessas classes [11, 12, 13, 25, 26, 27, 46, 49].

Todas as dependências entre os *checkpoints* de um padrão de *checkpoints* e mensagens RDT são causais. Tendo em vista essa observação, um protocolo RDT ótimo seria aquele

[6] Em inglês: *model-based*.

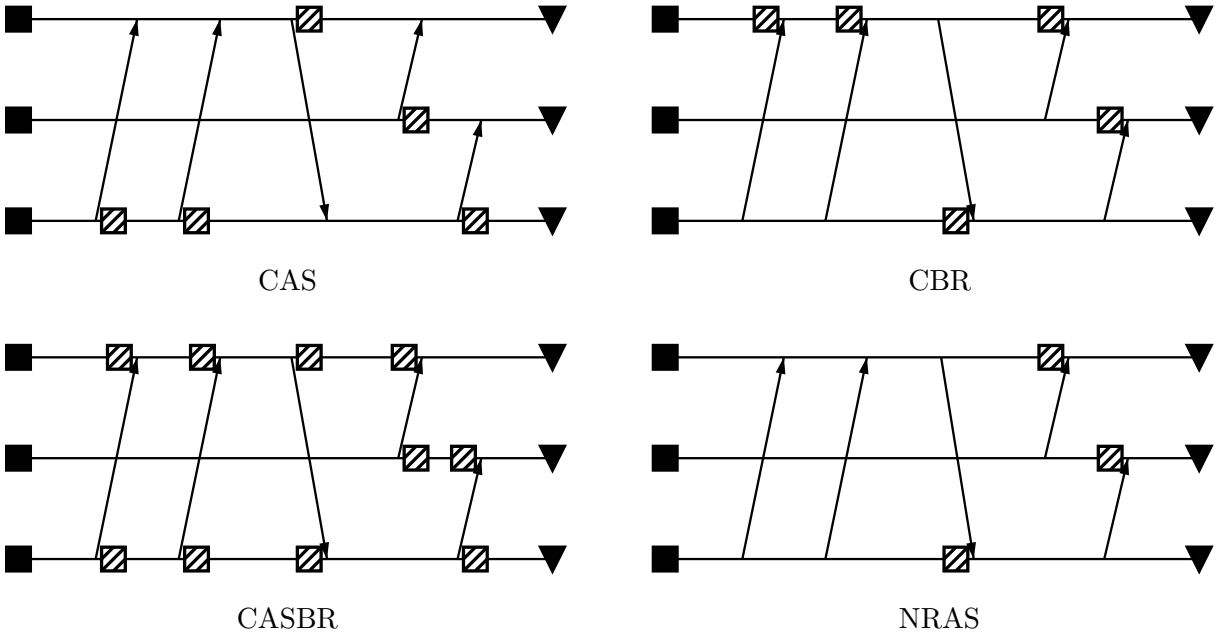


Figura 2.20: Protocolos SZPF.

que eliminando-se qualquer *checkpoint* forçado resultaria em alguma dependência não-causal não duplicada causalmente ou não quebrada. No entanto, tal protocolo parece ser impossível de ser projetado, pois teria que levar em conta informação futura sobre duplicação causal de caminhos-Z [36]. A Figura 2.21 ilustra este fato, onde a duplicação causal do caminho-Z $[m_2, m_1]$ só será feita com a mensagem m_3, mas o processo p_1 precisa decidir se grava um *checkpoint* forçado antes de entregar a mensagem m_2 para a aplicação muito antes da mensagem m_3 sequer ser enviada.

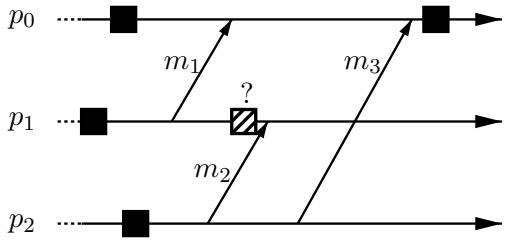


Figura 2.21: Impossibilidade de um protocolo RDT ótimo.

Apesar da impossibilidade de um protocolo RDT ótimo, um esforço pode ser visto na literatura em diminuir o número de *checkpoints* forçados dos protocolos RDT. Os pro-

tocolos da classe SZPF, por exemplo, induzem a gravação de um número de *checkpoints* forçados muito grande, pois estão baseados apenas nos eventos de gravação de *checkpoint* e envio e recepção de mensagens [36, 49]. O protocolo FDI grava menos *checkpoints* forçados que os protocolos da classe SZPF, pois propaga vetores de dependência, e não induz a gravação de um *checkpoint* forçado quando o processo recebe uma mensagem que não tenha dependências causais novas [49]. O protocolo *Fixed-Dependency-After-Send* (FDAS) também propaga vetores de dependência da mesma forma que o protocolo FDI, mas grava menos *checkpoints* forçados que este último, pois não induz um *checkpoint* forçado antes do primeiro evento de envio de mensagem em um intervalo entre *checkpoints* [49].

Da forma como foi proposto, o protocolo FDAS compara todas as entradas do vetor de dependências da mensagem recebida com as entradas do vetor mantido localmente pelo processo [49] para verificar se alguma dependência causal nova foi recebida. Porém, foi provado que é necessário apenas comparar a entrada correspondente ao emissor da mensagem, pois este valor sempre é atualizado quando o processo recebe informações causais novas [24]. O Algoritmo 2.2 mostra o FDAS com a otimização de verificar apenas uma entrada do vetor de dependências. O algoritmo envia um vetor de dependências nas mensagens da aplicação e possui uma variável booleana *enviou* que controla se o processo já enviou alguma mensagem no intervalo entre *checkpoints* corrente. Essa variável recebe o valor verdadeiro quando alguma mensagem é enviada e retorna ao valor falso quando algum *checkpoint* é salvo. Quando um processo recebe uma mensagem, ele verifica se recebeu alguma dependência causal nova, comparando a entrada referente ao processo emissor no seu vetor de dependências e no vetor da mensagem. Caso alguma dependência nova seja percebida, o processo deverá atualizar o seu vetor de dependências e também deverá gravar um *checkpoint* forçado se tiver enviado alguma mensagem no intervalo entre *checkpoints* atual. A Figura 2.22 mostra um padrão de *checkpoints* e mensagens gerado pelo protocolo FDAS.

Baldoni, Helary e Raynal fizeram um estudo aprofundado da propriedade RDT [12, 13], de modo a diminuir ainda mais o número de *checkpoints* forçados. Eles também conjeturaram que um determinado conjunto de dependências era o menor que deveria ser evitado em tempo de execução por um protocolo RDT [12]. Além disso, eles afirmaram que um algoritmo que evitasse esse conjunto de dependências deveria propagar e manter informações de controle com complexidade $O(n^2)$ [13]. No entanto, a caracterização minimal do conjunto de dependências a serem evitadas por um protocolo RDT foi provada por Garcia e Buzato e consiste em apenas um sub-conjunto do que foi conjeturado por Baldoni, Helary e Raynal [25]. Outra descoberta de Garcia e Buzato foi um protocolo RDT chamado RDT-Minimal, que implementa a caracterização minimal e que propaga e mantém informações de controle com complexidade linear no número de processos da aplicação [26].

Algoritmo 2.2 Protocolo FDAS em um processo p_i.

Estruturas de Dados

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

Inicialização

```
1: enviou ← falso
2: for j ← 0 to n-1 do
3:    DV[j] ← 0
4: grava checkpoint
```

Ao gravar *checkpoint*

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

Ao enviar mensagem m

```
1: enviou ← verdadeiro
2: for j ← 0 to n-1 do
3:    m.DV[j] ← DV[j]
4: envia(m)
```

Ao receber mensagem m **de** p_k

```
1: if m.DV[k] > DV[k] then                    {se nova dependência causal}
2:    if enviou then                          {se alguma mensagem foi enviada nesse intervalo}
3:       grava checkpoint                     {induz checkpoint forçado}
4:    for j ← 0 to n-1 do                     {atualiza o vetor de dependências}
5:       if m.DV[j] > DV[j] then
6:          DV[j] ← m.DV[j]
7: entrega(m)
```

O conjunto de dependências que devem ser evitadas em tempo de execução por um protocolo RDT são aquelas que podem ser causadas por caminhos-PMM que não possuem uma duplicação causal detectável durante a computação [25]. Um caminho-PMM é um caminho-Z constituído de uma mensagem-prima m_1 e uma outra mensagem m_2 [25]. Uma mensagem m enviada do intervalo entre *checkpoints* I_a^α para p_i é prima se m é a primeira mensagem recebida por p_i que traz informação sobre I_a^α [11, 12, 13], como a mensagem m_1 na Figura 2.23. Ainda nesta figura, o caminho-Z $[m_1, m_2]$ é um caminho-PMM.

O Algoritmo 2.3 mostra o RDT-Minimal, e como o conjunto de dependências mínimo pode ser evitado para que um protocolo garanta a propriedade RDT. O problema é dividido em dois casos menores, que utilizam informação de controle de complexidade $O(n)$. Um dos dois problemas menores são os chamados ciclos-CC, que são ciclos-Z formados por dois caminhos causais [11, 13]. Para detectar esses ciclos-CC, cada processo da aplicação

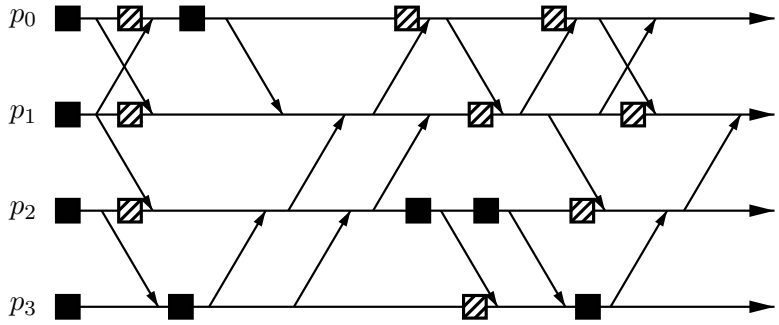


Figura 2.22: Padrão de *checkpoints* e mensagens gerado pelo protocolo FDAS.

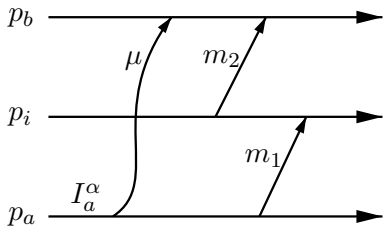


Figura 2.23: Caminho-PMM.

distribuída deve propagar e manter um vetor que identifique caminhos simples e não-simples, além do vetor de dependências. Assim, quando um processo p_i recebe uma mensagem m do processo p_k, e percebe que os vetores de dependência dos dois processos são iguais, indicando um ciclo formado por caminhos causais, e que este ciclo não é um caminho simples, ou seja, contém um *checkpoint*, o processo p_i deve induzir um *checkpoint* forçado, pois um ciclo-CC acaba de ser detectado. Esta detecção de um ciclo-CC pode ser vista na função **necessitaCheckpointForcado** do Algoritmo 2.3 (linhas 5-6).

A outra parte do conjunto de dependências que um protocolo RDT também precisa evitar são caminhos-PMM do processo p_k para o processo p_j quando os vetores de dependência dos dois processos são diferentes. Para detectar essas dependências, é preciso que cada processo da aplicação propague e mantenha um vetor que marque quais processos têm o vetor de dependências igual ao seu. Dessa forma, aquela dependência pode ser detectada no processo p_i, quando este recebe uma mensagem de p_k com uma nova dependência causal, e percebe que enviou uma mensagem para p_j naquele mesmo intervalo entre *checkpoints* e também que os vetores de dependência de p_j e p_k não são iguais. Quando isso ocorre, o processo p_i deve induzir um *checkpoint* forçado, como pode ser observado na função **necessitaCheckpointForcado** do Algoritmo 2.3 (linhas 7-9).

O conjunto mínimo de dependências a ser evitado por um protocolo RDT só pode ser detectado quando uma nova dependência causal é percebida na recepção de uma mensagem, como pode ser visto no procedimento que é executado quando uma mensagem é recebida no Algoritmo 2.3. Nesse procedimento também é realizada a atualização do vetor de dependências, do vetor de caminhos simples e do vetor que marca quais processos possuem vetores de dependências iguais ao do processo corrente.

Como o RDT-Minimal precisa propagar e manter vetores que indicam quais processos possuem vetores de dependências iguais, o intervalo entre *checkpoints* de um processo é dividido em três fases. Na fase zero, o processo não enviou nenhuma mensagem, portanto nenhum caminho-PMM pode ser formado, e então nenhum *checkpoint* forçado é induzido. Nesta fase, o processo pode atualizar sem restrições o seu vetor de dependências sempre que uma mensagem é recebida. Na fase um, o processo já enviou alguma mensagem, e *checkpoints* forçados são induzidos quando as dependências explicadas anteriormente são detectadas. Assim como na fase zero, nesta fase o processo também pode atualizar sem restrições o seu vetor de dependências. Por outro lado, na fase 2, o processo acaba de ter o conhecimento de que algum outro processo possui o vetor de dependências igual ao seu. Dessa maneira, o seu vetor de dependências não pode ser alterado quando uma nova dependência causal é percebida, e um *checkpoint* forçado é induzido, como pode ser visto na função **necessitaCheckpointForcado** do Algoritmo 2.3 (linhas 3-4).

A Figura 2.24 mostra o mesmo cenário de execução da Figura 2.22, mas com o RDT-Minimal como protocolo de *checkpointing* quase-síncrono. Percebe-se que o RDT-Minimal gravou menos *checkpoints* forçados que o FDAS nesse cenário. O RDT-Minimal tenta reduzir ao máximo o número de *checkpoints* forçados sem perder a propriedade RDT, mas é impossível que um protocolo induza menos *checkpoints* forçados que todos os outros em qualquer cenário de execução possível [46].

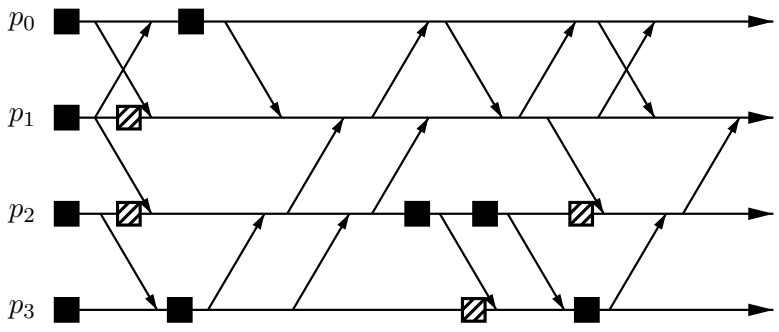


Figura 2.24: Padrão de *checkpoints* e mensagens gerado pelo protocolo RDT-Minimal.

Algoritmo 2.3 Protocolo RDT-Minimal em um processo p_i.

Estruturas de Dados

```
1: Var
2:    fase: inteiro
3:    DV: array[0...n-1] of inteiro
4:    igual, simples, enviou: array[0...n-1] of booleano
```

Inicialização

```
1: for j ← 0 to n-1 do
2:    DV[j] ← 0
3: grava checkpoint
```

Ao gravar *checkpoint*

```
1: for j ← 0 to n-1 do
2:    igual[j] ← falso
3:    simples[j] ← falso
4:    enviou[j] ← falso
5: igual[i] ← verdadeiro
6: simples[i] ← verdadeiro
7: fase ← 0
8: DV[i] ← DV[i] + 1
```

Ao enviar mensagem m **para o processo** p_k

```
1: for j ← 0 to n-1 do
2:    m.DV[j] ← DV[j]
3:    m.igual[j] ← igual[j]
4:    m.simples[j] ← simples[j]
5: enviou[k] ← verdadeiro
6: if fase = 0 then
7:    fase ← 1
8: envia(m)
```

Função necessitaCheckpointForcado(mensagem m)

```
1: if fase = 0 then
2:    retorna falso                                   {aceita novas dependências}
3: if fase = 2 then
4:    retorna verdadeiro                              {não aceita novas dependências}
5: if m.DV[i] = DV[i] ∧ ¬m.simples[i] then
6:    retorna verdadeiro                              {quebra um ciclo-CC}
7: for j ← 0 to n-1 do
8:    if enviou[j] ∧ ¬m.igual[j] then
9:       retorna verdadeiro          {quebra um caminho-PMM não visivelmente duplicado}
10: retorna falso
```

Algoritmo 2.3 Protocolo RDT-Minimal em um processo p_i. (Continuação)

Ao receber mensagem m **de** p_k

```
 1: if m.DV[k] > DV[k] then                          {se nova dependência causal}
 2:    if necessitaCheckpointForcado(m) then
 3:       grava checkpoint
 4:    for j ← 0 to n − 1 do                {atualiza o vetor de dependências e o vetor simples}
 5:       if m.DV[j] > DV[j] then
 6:          DV[j] ← m.DV[j]
 7:          simples[j] ← m.simples[j]
 8:       else if m.DV[j] = DV[j] then
 9:          simples[j] ← simples[j] ∧ m.simples[j]
10: if m.DV[i] = DV[i] then                  {se os vetores de dependências são iguais}
11:    for j ← 0 to n − 1 do                                  {atualiza o vetor igual}
12:       igual[j] = igual[j] ∨ m.igual[j]
13:    fase ← 2
14: entrega(m)
```

2.8 Sumário

Neste capítulo foram vistas as definições da relação de precedência causal e de corte consistente, de modo que o conceito de um *checkpoint* global consistente pudesse ser explicado. Trata-se de um estado consistente de toda a aplicação distribuída que é formado por um *checkpoint* de cada processo. Para que um grupo de *checkpoints* possa fazer parte de um *checkpoint* global consistente, é preciso que não exista caminhos em ziguezague entre nenhum par de *checkpoints* daquele conjunto [37]. Além disso, um caminho em ziguezague que inicia e termina no mesmo *checkpoint* é chamado de ciclo-Z, e o *checkpoint* em questão é dito inútil, pois não pode participar de nenhum *checkpoint* global consistente [37].

Os protocolos de *checkpointing* procuram garantir que *checkpoints* globais consistentes da aplicação possam ser criados ou calculados a partir dos *checkpoints* locais gravados pelos processos. Foram mostradas as três abordagens para *checkpointing*, que são: síncrona, assíncrona e quase-síncrona. Na abordagem sincrona, os processos utilizam mensagens de controle para que possam sincronizar a gravação dos seus estados e garantir que o conjunto formado pelos *checkpoints* gravados seja um *checkpoint* global consistente. Na abordagem assíncrona, os processos possuem autonomia para gravar *checkpoints* locais, mas a existência de um *checkpoint* global consistente não é garantida. Na abordagem quase-síncrona, além dos processos poderem selecionar *checkpoints*, que são chamados de básicos, informações de controle são enviadas juntamente com todas as mensagens da aplicação, de modo que o algoritmo possa avaliar se um *checkpoint* forçado deve ser induzido quando uma mensagem é recebida. Estes *checkpoints* são forçados para que de-

pendências no padrão de *checkpoints* e mensagens sejam quebradas e *checkpoints* globais consistentes possam ser formados com o conjunto de *checkpoints* gravados pelos processos.

Dentro do grupo de protocolos de *checkpointing* quase-síncronos existem algumas classes que se diferenciam pelos tipos de dependências que existem nos padrões de *checkpoints* e mensagens gerados. Entre as classes existentes, a classe RDT se destaca por possuir apenas protocolos que geram padrões de *checkpoints* e mensagens onde todas as dependências podem ser capturadas em tempo de execução utilizando-se vetores de dependência [49]. Embora os protocolos RDT induzam uma grande quantidade de *checkpoints* forçados, que está relacionada com o número de mensagens enviadas pela aplicação [47], vários estudos já foram realizados para tentar minimizar esse problema [13, 25]. Nessa linha de estudos, um algoritmo chamado RDT-Minimal que implementa a caracterização minimal do conjunto de dependências a serem evitadas por um protocolo RDT foi descoberto [26]. Este protocolo tenta minimizar o número de *checkpoints* forçados que são induzidos durante a execução da aplicação, embora não exista um protocolo que induza menos *checkpoints* forçados que todos os outros em qualquer cenário possível [46].

Capítulo 3

Recuperação e Coleta de Lixo

Durante a execução de uma aplicação distribuída que utiliza *checkpointing* para a recuperação de falhas por retrocesso de estado, vários *checkpoints* são salvos em meio de armazenamento estável. Desse modo, na ocorrência de uma falha, os *checkpoints* salvos podem ser usados para recuperar um estado anterior da aplicação. Porém, para que isso anconteça, o sistema de recuperação precisa perceber a falha, identificar os processos falhos e coordenar a aplicação para que esta retorne para um estado global consistente. Uma tarefa ortogonal à esta de recuperação, mas de grande importância, é a identificação e eliminação dos *checkpoints* gravados que se tornaram obsoletos à medida que a computação progrediu. Esta tarefa é chamada de coleta de lixo, e é desejável que ela estabeleça um limite superior para a utilização de espaço no meio de armazenamento estável, pois este último não é ilimitado.

Nesse capítulo serão abordados os aspectos teóricos da recuperação por retrocesso de estado e da coleta de lixo. Inicialmente, o modelo de execução da computação será explicado e algumas definições, como a de linha de recuperação, serão mostradas. Posteriormente, serão apresentados algoritmos de recuperação por retrocesso de estado e de coleta de lixo para serem usados em conjunto com protocolos de *checkpointing* em um sistema tolerante a falhas.

3.1 Modelo de Execução e Definições

Os processos de uma aplicação distribuída gravam seus *checkpoints* em um meio de armazenamento estável local ou global, de modo que na ocorrência de uma falha seus estados possam ser recuperados. Assume-se que os processos podem sofrer falhas do tipo *crash* nas quais param e perdem o seu estado corrente de execução. Assim, um sistema para prover tolerância a falhas precisa de um mecanismo de detecção de falhas e outro para realizar a recuperação das falhas.

O mecanismo de recuperação é acionado quando alguma falha é detectada. Ele deve identificar o grupo de processos falhos, e a partir desta informação, ou criar novos processos que possam responder por aqueles que falharam ou esperar que eles sejam reiniciados. Depois que todos os processos voltarem a executar, o sistema de recuperação precisa calcular a linha de recuperação e forçar os processos a retrocederem para o seu *checkpoint* local correspondente. Assim, o modelo de execução adotado é dividido em períodos de execução sem falha alternados com sessões de recuperação, como ilustrado na Figura 3.1.

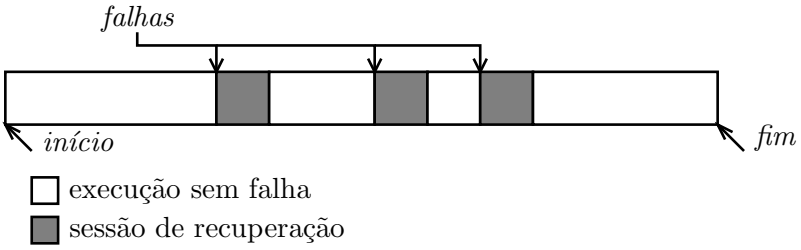


Figura 3.1: Modelo de execução da aplicação distribuída.

Durante uma execução sem falhas da aplicação distribuída, além do protocolo de *checkpointing* gravar os *checkpoints* no meio de armazenamento estável, a coleta de lixo deve identificar e eliminar os *checkpoints* que se tornam obsoletos e não serão mais usados numa recuperação do estado da aplicação. O conceito de *checkpoint* obsoleto é válido apenas para *checkpoints* estáveis, pois está diretamente associado à coleta de lixo.

Definição 3.1 ***Checkpoint Obsoleto*** — *Um checkpoint estável é obsoleto se, e somente se, ele não pode tomar parte em qualquer linha de recuperação presente ou futura, mesmo depois de um ou mais retrocessos.*

A linha de recuperação que é calculada na sessão de recuperação é o estado recuperável mais recente da computação distribuída. Em recuperação de falhas por retrocesso de estado utilizando *checkpointing*, corresponde ao *checkpoint* global consistente mais recente dentro do padrão de *checkpoints* e mensagens gerado até o momento da falha. É fato que a linha de recuperação não poderá conter o *checkpoint* volátil de um processo que falhou, pois este estado é perdido no momento da falha. A determinação da linha de recuperação é baseada na minimização da quantidade de trabalho que é perdido, e que pode ser medida pelo número de *checkpoints* que são retrocedidos, ou seja, que ficam à frente do *checkpoint* global consistente calculado.

Definição 3.2 ***Custo de Retrocesso*** — *Dado um checkpoint global consistente para o qual a computação pode ser retrocedida após uma falha, seu custo de retrocesso corresponde ao número de checkpoints que estão após o estado global representado.*

Definição 3.3 ***Linha de Recuperação*** — *Dados um padrão de checkpoints e mensagens e um conjunto $F \subseteq P$ de processos falhos, a linha de recuperação R_F é o checkpoint global consistente que não inclui um checkpoint volátil de um processo falho e que minimiza o custo de retrocesso.*

3.2 Recuperação por Retrocesso de Estado

Uma vez que uma falha foi detectada durante a execução de uma aplicação distribuída, o sistema de recuperação por retrocesso de estado deverá ser acionado. Depois que o conjunto de processos falhos foi identificado e foram reiniciados ou outros foram criados, a aplicação como um todo deve ser retrocedida para a sua linha de recuperação. O cálculo da linha de recuperação precisa ser feito pois podem haver dependências entre os *checkpoints* gravados ao longo da execução da aplicação distribuída pelo protocolo de *checkpointing*, e pode não ser possível saber apenas com informações locais para qual *checkpoint* cada processo precisa retroceder. Além disso, dependendo do protocolo de *checkpointing* utilizado, o padrão de *checkpoints* e mensagens gerado pode conter *checkpoints* inúteis. Nas próximas seções serão apresentados mecanismos de recuperação e meios para calcular a linha de recuperação que podem ser usados num sistema de recuperação de falhas por retrocesso de estado.

3.2.1 Mecanismos de Recuperação

Quando uma sessão de recuperação é iniciada, o mecanismo de recuperação deve retroceder a computação para um estado consistente anterior o mais recente possível. Dependendo da sua implementação, o mecanismo de recuperação pode ser classificado como síncrono ou assíncrono [22].

Quando um esquema síncrono de recuperação é utilizado, um processo deve exercer o papel de coordenador e este pode ser externo à computação ou ser um dos processos da própria aplicação distribuída. Ele deve avisar os demais processos que uma sessão de recuperação será iniciada e coordenar o retrocesso para a linha de recuperação. Esse coordenador pode ser implementado utilizando-se um algoritmo distribuído de validação atômica, como o *two-phase commit* [29, pp. 562-572], para garantir que ou todos os processos participam do retrocesso ou nenhum deles participa, não deixando a aplicação em um estado inconsistente. Numa primeira fase, o coordenador do retrocesso avisa os demais processos da sessão de recuperação, e esses concordam com o retrocesso e respondem com informações sobre os seus *checkpoints* gravados. Na segunda fase, o coordenador calcula a linha de recuperação e a divulga para todos os processos, que retrocedem para o seu *checkpoint* pertencente à linha calculada.

Em um esquema assíncrono, não há a presença de um coordenador do retrocesso e a recuperação é realizada por uma propagação de retrocessos [35]. Quando um processo que falhou é reiniciado, este comunica aos demais processos que ele falhou e que está retrocedendo para o seu último *checkpoint* estável. Os demais processos que recebem essa mensagem e percebem alguma dependência com relação ao retrocesso informado devem retroceder também para garantir a consistência do estado da aplicação distribuída. Ao retrocederem, eles também enviam um aviso para os outros processos informando o seu retrocesso. Essa propagação de retrocessos continua até que nenhum processo necessite mais retroceder e o estado da aplicação tenha atingido um *checkpoint* global consistente. Neste mecanismo de recuperação assíncrono, o cálculo da linha de recuperação não precisa ser feito de forma explícita, pois ele é realizado de forma implícita durante a propagação de retrocessos.

As duas abordagens para o mecanismo de recuperação podem ser utilizadas com qualquer tipo de protocolo de *checkpointing*. Porém, ao utilizar um mecanismo de recuperação assíncrono, falhas concorrentes ou falhas durante a execução dos retrocessos podem tornar a implementação muito complexa. Por outro lado, mecanismos síncronos de recuperação são mais simples de serem implementados, pois o coordenador tem o controle total do retrocesso de todos os processos da aplicação distribuída.

3.2.2 Cálculo da Linha de Recuperação

A linha de recuperação precisa minimizar o custo de retrocesso da aplicação distribuída. Desse modo, é possível que a linha de recuperação para uma falha em um conjunto de processos da aplicação distribuída contenha o *checkpoint* volátil de processos que não falharam. O padrão de *checkpoints* e mensagens apresentado na Figura 3.2 mostra uma falha no processo p_2 durante a computação. A aplicação poderia ser retrocedida para o *checkpoint* global consistente C que possui apenas *checkpoints* estáveis, mas a linha de recuperação para este CCP é o *checkpoint* global consistente C', que inclui o *checkpoint* volátil do processo p_0.

Quando um protocolo síncrono de *checkpointing* é utilizado, a tarefa de calcular a linha de recuperação é muito simples. Uma vez que a cada execução do protocolo é formado um *checkpoint* global consistente, o sistema de recuperação precisa apenas coordenar os processos da aplicação para retornarem ao último *checkpoint* global consistente gravado, que será a linha de recuperação. Assim, cada processo da aplicação distribuída precisa retroceder para o seu *checkpoint* local pertencente ao último *checkpoint* global consistente gravado durante a execução do protocolo síncrono de *checkpointing*.

Por outro lado, quando um protocolo assíncrono ou um protocolo quase-síncrono de *checkpointing* é utilizado, a troca de mensagens realizada durante a execução da aplicação

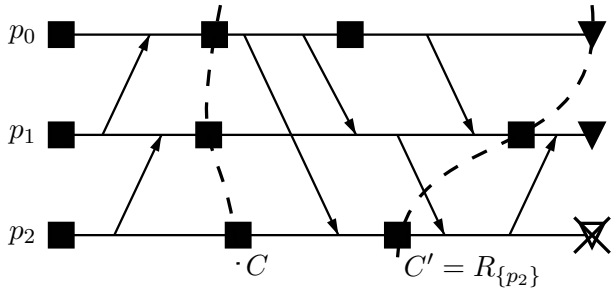


Figura 3.2: Exemplo de linha de recuperação.

cria dependências entre os *checkpoints* gravados. Assim, o *checkpoint* global consistente que será a linha de recuperação não é claramente conhecido pelos processos quando ocorrem falhas e deverá ser calculado. Além disso, a linha de recuperação neste caso não será a mesma para qualquer conjunto de processos falhos, como acontece quando se utiliza um protocolo de *checkpointing* síncrono.

Cálculo da Linha de Recuperação usando *R-graph*

Para calcular a linha de recuperação, um grafo de dependências de retrocesso[1] (também chamado de *R-graph*) pode ser utilizado [48, 49]. Um *R-graph* é um grafo em que os vértices representam *checkpoints* do CCP e arestas significam dependências de retrocesso. Desse modo, uma aresta do *checkpoint* c_a^α para o *checkpoint* c_b^γ significa que o retrocesso do *checkpoint* c_a^α deve forçar o retrocesso do *checkpoint* c_b^γ.

Um *R-graph* de um padrão de *checkpoints* e mensagens é construído representado-se cada *checkpoint* por um vértice e colocando-se uma aresta do *checkpoint* c_a^α para o *checkpoint* c_b^γ caso:

1. $a \neq b$ e uma mensagem m foi enviada em I_a^α e recebida em I_b^γ; ou

2. $a = b$ e $\gamma = \alpha + 1$.

Para que o *R-graph* do padrão de *checkpoints* e mensagens possa ser construído na ocorrência de uma falha, os processos precisam realizar a captura de dependências diretas entre os *checkpoints*. É possível capturar as dependências diretas anexando o índice do intervalo entre *checkpoints* corrente a cada mensagem enviada. Além disso, cada processo deve guardar todas as dependências recebidas num intervalo entre *checkpoints* junto com o próximo *checkpoint* armazenado. Dessa forma, quando uma falha ocorre e o sistema de

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

recuperação é acionado, as dependências diretas gravadas juntamente com cada *checkpoint* de todos os processos devem ser recolhidas para que o *R-graph* possa ser construído e a linha de recuperação determinada.

Para calcular a linha de recuperação usando um *R-graph*, basta fazer uma busca no grafo a partir dos vértices dos *checkpoints* voláteis dos processos que falharam e marcar todos os demais vértices que podem ser alcançados. Como o *R-graph* define dependências de retrocesso, os vértices marcados nessa busca deverão ser retrocedidos, e a linha de recuperação será formada pelo último *checkpoint* não marcado de cada um dos processos. Na Figura 3.3 (a) é mostrado um padrão de *checkpoints* e mensagens de uma aplicação até a ocorrência de falhas em p_1 e p_3. O *R-graph* correspondente a esse CCP está ilustrado na Figura 3.3 (b), onde também estão destacados os *checkpoints* que devem ser retrocedidos e a linha de recuperação para as falhas em p_1 e p_3, $R_{\{p_1,p_3\}}$.

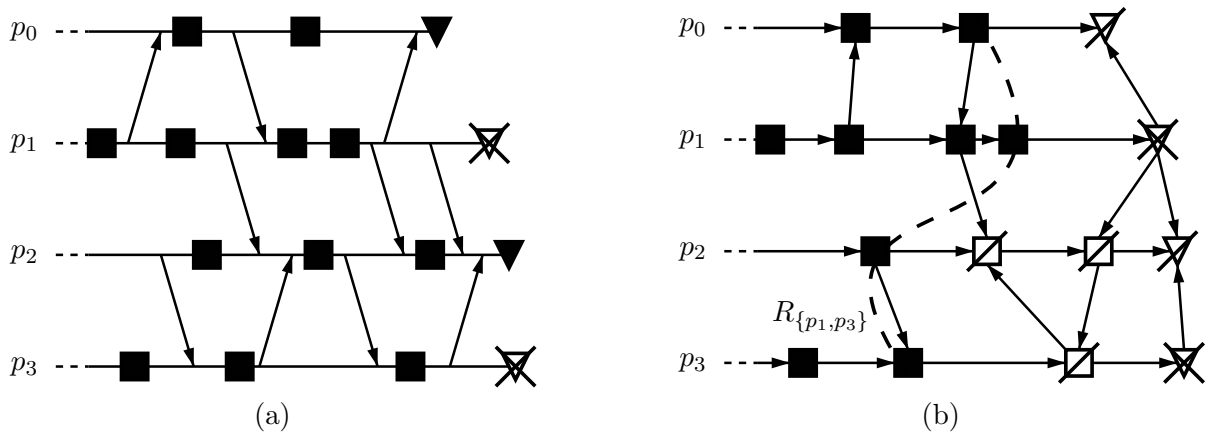


Figura 3.3: CCP com falhas e o *R-graph* correspondente.

Cálculo da Linha de Recuperação em Padrões RDT

Os padrões de *checkpoints* e mensagens gerados por protocolos RDT garantem que as dependências existentes entre os *checkpoints* são causais e que elas podem ser capturadas em tempo de execução utilizando-se vetores de dependências [49]. Aproveitando essas características de um padrão de *checkpoints* e mensagens RDT, um *R-graph* não precisa ser construído para calcular a linha de recuperação. Além disso, em padrões RDT, e em padrões gerados por protocolos ZCF também, o último *checkpoint* estável de um processo falho sempre faz parte da linha de recuperação considerando-se que no CCP da computação apenas esse processo falha [42]. Assim, a linha de recuperação para a falha de um processo p_f num padrão de *checkpoints* e mensagens RDT pode ser calculada

procurando-se para cada processo que não falhou, o *checkpoint* mais recente que não é precedido causalmente pelo último *checkpoint* estável do processo p_f [42]. O cálculo dessa linha pode ser definido mais formalmente como:

$$R_{\{p_f\}} = \bigcup_{i=0}^{n-1} \{c_i^{\max(k)} | s_f^{last} \not\rightarrow c_i^k\} \tag{3.1}$$

Para realizar o cálculo da linha de recuperação em padrões RDT de forma prática é possível usar o algoritmo proposto por Wang para calcular o máximo *checkpoint* global consistente que contém um conjunto de *checkpoints* [48, 49]. Este *checkpoint* global consistente é aquele mais recente no CCP que contém o grupo de *checkpoints* especificado. Desse modo, pode-se usar esse algoritmo para achar o *checkpoint* global consistente mais recente que contém o último *checkpoint* estável de um processo falho, ou seja, a linha de recuperação para a falha desse processo no padrão de *checkpoints* e mensagens RDT que foi gerado. Esse algoritmo usa as relações existentes entre a precedência causal e vetores de dependência para cada processo que não falhou descobrir qual *checkpoint* seu fará parte da linha de recuperação. Assim, se cada processo p_i que não falhou souber que o processo falho p_f será retrocedido para o último *checkpoint* estável $s_f^{last(f)}$, o *checkpoint* de p_i que fará parte da linha de recuperação será o *checkpoint* c_i^k de maior índice k tal que $DV(c_i^k)[f] \leq last(f)$ [48, 49].

A Figura 3.4 mostra o mesmo padrão de *checkpoints* e mensagens RDT gerado pelo protocolo FDI visto anteriormente na Figura 2.19, mas com uma falha no processo p_1. Os demais processos deverão ser informados que o processo p_1 falhou e será retrocedido para o seu último *checkpoint* estável s_1^1. Usando o algoritmo de Wang, percebe-se que o *checkpoint* s_0^1 de p_0 faz parte da linha de recuperação, pois $DV(s_0^1)[1] \leq last(1)$ $(0 \leq 1)$ e é o *checkpoint* de p_0 de maior índice que possui esta propriedade. Para minimizar o retrocesso da computação, um *checkpoint* volátil de um processo que não falhou pode ser usado na linha de recuperação. É o caso do *checkpoint* volátil de p_2, que faz parte da linha de recuperação para a falha em p_1, pois é o *checkpoint* mais recente de p_2 que obedece à propriedade do algoritmo de Wang, ou seja, $DV(v_2)[1] \leq last(1)$ $(0 \leq 1)$.

A aplicação distribuída está sujeita à ocorrência de falhas em mais de um processo durante a computação. Quando o conjunto de processos falhos detectado em uma sessão de recuperação é composto por mais de um processo, o cálculo da linha de recuperação deve levar em conta todas as falhas. Em padrões de *checkpoints* e mensagens RDT é possível calcular a linha de recuperação para todas as falhas a partir das linhas de recuperação para as falhas individuais de cada um dos processos. Depois de calcular as linhas de recuperação para as falhas individuais, a linha de recuperação final pode ser calculada usando-se a intersecção de *checkpoints* globais mostrada na Seção 2.4. Como a intersecção de dois *checkpoints* globais consistentes resulta em um *checkpoint* global

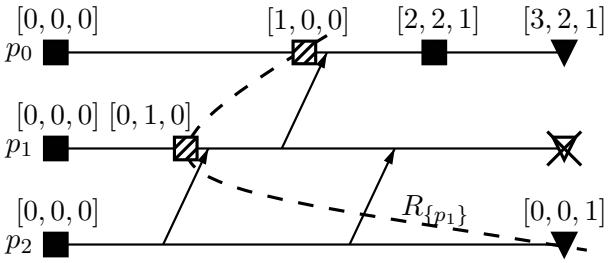


Figura 3.4: Linha de recuperação em um padrão de *checkpoints* e mensagens RDT.

também consistente, a linha de recuperação para um padrão de *checkpoints* e mensagens RDT com várias falhas será o resultado da intersecção entre as linhas de recuperação para as falhas individuais.

3.3 Coleta de Lixo

À medida que a aplicação distribuída executa e uns *checkpoints* são gravados pelo protocolo de *checkpointing*, outros se tornam obsoletos e podem ser eliminados do meio de armazenamento estável. Dependendo do protocolo de *checkpointing* utilizado, a identificação dos *checkpoints* obsoletos pode ser mais trabalhosa e nem sempre pode ser feita apenas com a informação local de cada processo. Nas próximas seções serão mostrados algoritmos de coleta de lixo para protocolos de *checkpointing*.

3.3.1 Coleta de Lixo para Protocolos de *Checkpointing* Síncronos

A coleta de lixo para protocolos síncronos de *checkpointing* se beneficia de que a linha de recuperação é sempre conhecida pelos processos mesmo que uma falha não ocorra. Como a aplicação será retrocedida para o último *checkpoint* global consistente na ocorrência de uma falha, os *checkpoints* atrás dessa barreira de *checkpoints* estáveis podem ser eliminados do meio de armazenamento, pois não serão mais utilizados. A Figura 3.5 mostra o *checkpoint* global consistente C e os *checkpoints* anteriores que se tornaram obsoletos (representados por quadrados brancos) e podem ser eliminados. Ainda nesta figura, o *checkpoint* global consistente C é a linha de recuperação para as falhas que ocorreram nos processos p_1 e p_2, ou seja, $R_{\{p_1,p_2\}} = C$.

Como os *checkpoints* anteriores ao último *checkpoint* global consistente gravado pelo protocolo síncrono se tornam obsoletos e podem ser eliminados, cada processo da aplicação distribuída precisa manter apenas um *checkpoint* armazenado entre duas execuções do

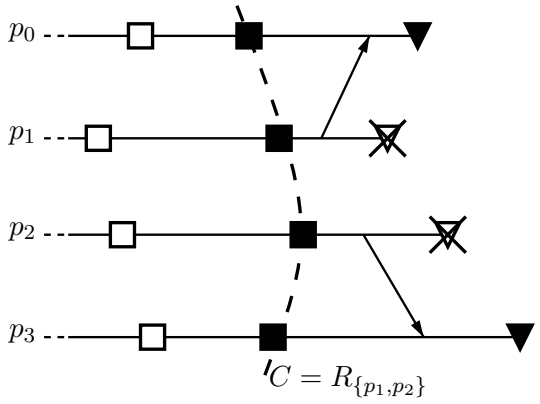


Figura 3.5: *Checkpoints* obsoletos e linha de recuperação.

protocolo de *checkpointing*. Durante a execução do protocolo, cada processo precisa manter temporariamente dois *checkpoints* armazenados. Enquanto o novo *checkpoint* global consistente não estiver completamente formado, e conseqüentemente a nova linha de recuperação não estiver completa, cada processo da aplicação não poderá eliminar os *checkpoints* pertencentes à linha de recuperação anterior. Pode-se justificar essa condição observando-se que uma falha pode ocorrer antes que o protocolo síncrono termine e um retrocesso para a linha de recuperação anterior pode ser necessário.

É necessário que os *checkpoints* anteriores sejam mantidos até o término da execução do protocolo de *checkpointing* síncrono, quando a nova linha de recuperação estará formada. Porém, existem alguns protocolos síncronos, como os não-bloqueantes [17], em que os processos não sabem do término do protocolo. Assim, é preciso que os processos avisem uns aos outros quando já gravaram os seus *checkpoints*, e quando um processo recebe essa notificação de todos os outros processos, ele pode eliminar o seu *checkpoint* pertecente ao *checkpoint* global consistente anterior.

Cada processo da aplicação distribuída poderá ter que manter mais de dois *checkpoints* armazenados caso haja mais de uma invocação do protocolo de *checkpointing*, pois dependências podem ser introduzidas entre os *checkpoints* das diferentes instâncias. Uma maneira de evitar esse problema é determinar que apenas um processo pode invocar o algoritmo para a gravação do *checkpoint* global consistente ou utilizar um protocolo que saiba lidar com execuções concorrentes [31].

3.3.2 Coleta de Lixo Ingênua

É possivel realizar uma coleta de lixo para protocolos de *checkpointing* assíncronos e quase-síncronos que utiliza a idéia de coleta de lixo para protocolos síncronos. Basta calcular a linha de recuperação mais antiga que a aplicação pode ser retrocedida no padrão de *checkpoints* e mensagens, e eliminar todos os *checkpoints* anteriores a ela. Esta linha de recuperação corresponde àquela calculada para uma falha em todos os processos da aplicação e a coleta de lixo que é feita utilizando essa linha é chamada de ingênua. Na Figura 3.6 é mostrada a linha de recuperação para uma falha em todos os processos da computação (R_P), e os *checkpoints* obsoletos atrás dessa barreira, representados por quadrados brancos, podem ser eliminados pela coleta de lixo ingênua.

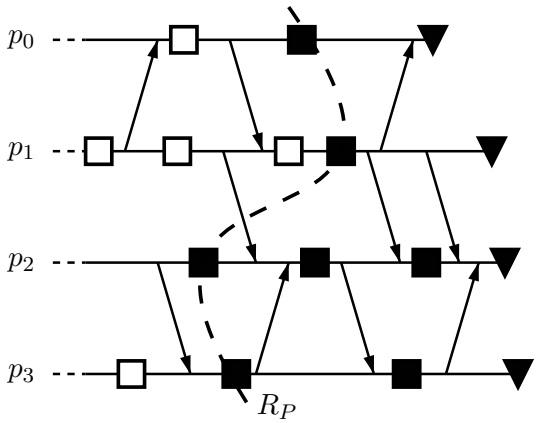


Figura 3.6: Exemplo de coleta de lixo ingênua.

O algoritmo de coleta de lixo ingênua poderia ser parecido com o mecanismo de recuperação síncrono descrito na Seção 3.2.1. Numa primeira fase, a execução da aplicação é bloqueada, e o processo coordenador recolhe informações sobre dependências entre *checkpoints* de todos os demais processos e constrói um *R-graph* do padrão de *checkpoints* e mensagens. Ainda na primeira fase, o coordenador calcula a linha de recuperação para uma falha em todos os processos da aplicação usando o *R-graph*. Na segunda fase, o coordenador propaga a linha de recuperação calculada para todos os processos, que irão eliminar todos os seus *checkpoints* anteriores a ela. As informações de dependência associadas a cada *checkpoint* eliminado também podem ser eliminadas, pois não serão mais úteis. Após completar a coleta de lixo, todos os processos podem voltar à sua execução normal.

Quando um protocolo de *checkpointing* quase-síncrono que satisfaz a propriedade RDT é utilizado, a coleta de lixo ingênua pode ser simplificada. Como as dependências causais

podem ser capturadas em tempo de execução com a utilização de relógios vetoriais ou vetores de dependências, cada processo pode realizar a coleta de lixo ingênua localmente, caso saiba o índice do último *checkpoint* estável de todos os outros processos. Supõe-se que cada processo tenha um vetor UC (Último *Checkpoint*) que contém o índice do último *checkpoint* estável de todos os processos da aplicação ($UC[k] = last(k)$, para $0 \leq k < n$). Além disso, os processos propagam e mantêm um vetor de dependências. Dessa forma, cada processo precisa procurar o seu *checkpoint* mais recente que não depende causalmente de nenhum *checkpoint* s_f^{last} para todos os valores $0 \leq f < n$. Traduzindo para um relação que utilize o vetor de dependências e o vetor UC, cada um dos processos da aplicação deve encontrar o seu *checkpoint* mais recente s_a^α tal que $UC[f] \geq DV(s_a^\alpha)[f]$ para todos os valores de f, e eliminar todos os *checkpoints* anteriores a ele [42]. A Figura 3.7 mostra um exemplo da coleta ingênua em um padrão de *checkpoints* e mensagens RDT, destacando a linha de recuperação R_P, o vetor UC utilizado e os *checkpoints* obsoletos eliminados, representados pelos quadrados brancos.

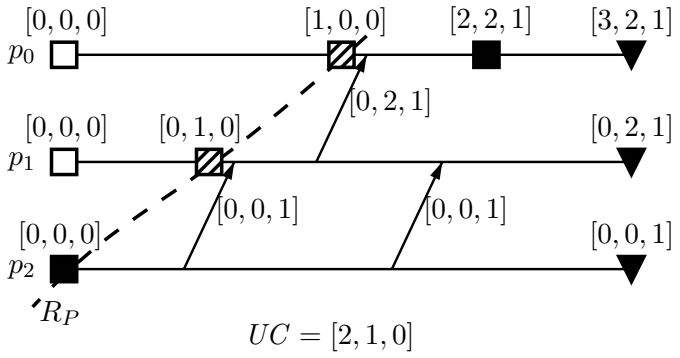


Figura 3.7: Coleta de lixo ingênua em um padrão de *checkpoints* e mensagens RDT.

Embora os algoritmos mostrados para a coleta ingênua sejam simples, eles apresentam alguns problemas. Um deles é que é necessário informação global sobre a computação para executar o algoritmo. Um processo deve recolher as informações de dependência entre todos os *checkpoints* para poder construir um *R-graph*, calcular a linha de recuperação para uma falha total da aplicação e propagá-la para todos os processos coletarem os seus *checkpoints* obsoletos. Ou então, no caso de um protocolo quase-síncrono RDT ser utilizado, cada processo deve mandar uma mensagem para os demais quando gravar um *checkpoint*, para que o vetor UC possa ser atualizado corretamente e a coleta de lixo possa ser realizada sem paralisar a computação. Ainda assim, é possível que alguns processos fiquem atrasados na coleta de lixo, mas um *checkpoint* que não é obsoleto nunca será eliminado, pois os valores das entradas do vetor UC nunca decrescem. Assim,

independente do protocolo de *checkpointing*, serão necessárias mensagens de controle para que a coleta de lixo ingênua seja realizada, o que praticamente elimina a principal vantagem dos protocolos assíncronos e quase-síncronos, que é a ausência desse tipo de mensagens.

Outro problema inerente à coleta de lixo ingênua é o fato de que ela depende do avanço da linha de recuperação para uma falha em todos os processos para que *checkpoints* se tornem obsoletos e sejam eliminados. Assim, é impossível estabelecer um limite para o número de *checkpoints* que precisam ser mantidos em meio de armazenamento estável, pois mesmo que uma aplicação grave *checkpoints* ao longo de sua execução, caso a linha de recuperação para uma falha em todos os processos não avance, nenhum *checkpoint* poderá ser eliminado. Além disso, a coleta de lixo ingênua não elimina todos os *checkpoints* obsoletos, pois estes podem existir depois da linha de recuperação para uma falha total da aplicação.

3.3.3 Coleta de Lixo Ótima

Para realizar uma coleta de lixo ótima é preciso que os *checkpoints* obsoletos possam ser identificados independente de onde eles ocorram no padrão de *checkpoints* e mensagens. Os *checkpoints* anteriores a uma barreria de *checkpoints* estáveis, como a linha de recuperação para uma falha em todos os processos da aplicação, são considerados obsoletos, pois não serão mais usados em nenhum retrocesso da computação. Seguindo com essa idéia, foi provado que um *checkpoint* é obsoleto se ele não faz parte de nenhuma das n linhas de recuperação para falhas individuais dos processos da aplicação [42]. Além disso, um *checkpoint* se tornar obsoleto é uma propriedade estável, ou seja, quando um *checkpoint* se torna obsoleto, ele continua assim até o fim da execução da aplicação, mesmo que ela seja retrocedida quantas vezes forem necessárias [42].

O algoritmo de coleta de lixo ótimo é parecido com o da coleta de lixo ingênua da Seção 3.3.2. A aplicação é bloqueada e um processo coordenador reúne as dependências entre *checkpoints* que cada processo guardou durante a computação e cria um *R-graph* do padrão de *checkpoints* e mensagens. Depois de construído o *R-graph*, o coordenador o utiliza para calcular as linhas de recuperação para falhas individuais dos processos da aplicação e marca todos os *checkpoints* que são utilizados em pelo menos uma linha de recuperação. Esses *checkpoints* que são utilizados em pelo menos uma linha de recuperação são os *checkpoints* não-obsoletos. Assim, o coordenador propaga esse conjunto de *checkpoints* para todos os processos, que eliminam os seus *checkpoints* que não estiverem no conjunto de *checkpoints* não-obsoletos recebido. Depois de eliminar todos os *checkpoints* obsoletos, os processos retomam a sua execução normal. Na execução desse algoritmo é preciso tomar cuidado, pois nem todos os *checkpoints* obsoletos po-

dem ter suas informações de dependência eliminadas. Quando os processos capturam dependências diretas entre os *checkpoints*, e o *checkpoint* obsoleto está depois da linha de recuperação para uma falha em todos os processos da aplicação, as dependências daquele *checkpoint* devem passar para o próximo *checkpoint* não-obsoleto naquele processo. Além disso, caso a dependência seja de um *checkpoint* obsoleto em outro processo, ela deverá ser ajustada para o primeiro *checkpoint* não-obsoleto daquele mesmo processo antes do *checkpoint* obsoleto que foi eliminado. A Figura 3.8 mostra um exemplo da coleta de lixo ótima no mesmo padrão de *checkpoints* e mensagens da Figura 3.6, destacando todos os *checkpoints* obsoletos (quadrados brancos).

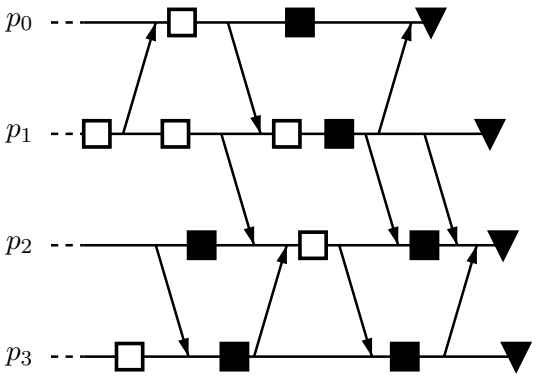


Figura 3.8: Exemplo de coleta de lixo ótima.

A identificação de todos os *checkpoints* obsoletos pode ser feita de forma diferente quando se usa um protocolo de *checkpointing* quase-síncrono RDT. Para identificar os *checkpoints* obsoletos é preciso procurar aqueles *checkpoints* s_a^α que não fazem parte de nenhuma linha de recuperação para falhas individuais dos processos da aplicação, ou seja, aqueles para os quais não existe um processo p_f tal que s_f^{last} precede causalmente $c_a^{\alpha+1}$ e não precede causalmente s_a^α [42]. Supondo que cada processo da aplicação tenha um vetor UC com o índice do último *checkpoint* estável de todos os processos, os *checkpoints* obsoletos são aqueles que não existe um processo p_f tal que $UC[f] < DV(c_a^{\alpha+1})[f] \wedge UC[f] \geq DV(s_a^\alpha)[f]$ [42].

Com base na idéia do parágrafo anterior é possível modificar o algoritmo de coleta de lixo ótima visto anteriormente para protocolos RDT. Nesse algoritmo, a aplicação também deve ser bloqueada durante a coleta de lixo e o coordenador deve recolher o índice do último *checkpoint* estável de cada processo da aplicação no vetor UC e propagá-lo para todos os processos. Assim, usando o vetor UC e o vetor de dependências DV, cada processo da aplicação pode identificar os seus *checkpoints* obsoletos e eliminá-los. Após realizar a coleta de lixo, a aplicação pode retornar a executar normalmente. A Figura 3.9

mostra um exemplo da coleta de lixo ótima em um padrão de *checkpoints* e mensagens RDT, destacando os *checkpoints* obsoletos identificados (quadrados brancos) e o vetor UC utilizado.

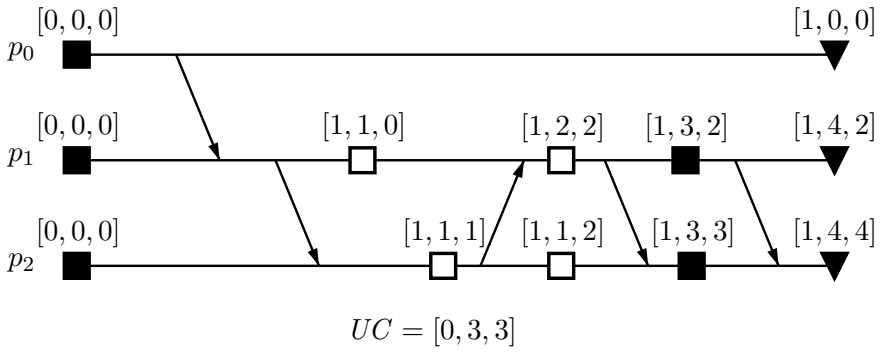


Figura 3.9: Coleta de lixo ótima em um padrão de *checkpoints* e mensagens RDT.

Os algoritmos de coleta de lixo ótima conseguem identificar e eliminar todos os *checkpoints* obsoletos de um padrão de *checkpoints* e mensagens. Além desta vantagem sobre a coleta de lixo ingênua, a coleta de lixo ótima também estabelece um limite para o número máximo de *checkpoints* não-obsoletos que devem permanecer no meio de armazenamento. Como um *checkpoint* não-obsoleto é aquele que pertence a pelo menos alguma linha de recuperação para falhas individuais dos processos da aplicação, cada processo pode ter no máximo n *checkpoints* não-obsoletos distintos, sendo um para cada uma das n linhas de recuperação para falhas individuais. Assim, na aplicação toda, é possível assumir que no máximo n^2 *checkpoints* não são obsoletos. Porém, o limite superior para o número de *checkpoints* não-obsoletos em um padrão de *checkpoints* e mensagens é menor e já foi provado como sendo $n(n+1)/2$ [42].

Apesar das vantagens da coleta de lixo ótima em relação à coleta de lixo ingênua, a primeira ainda possui o problema de precisar de informações globais sobre o padrão de *checkpoints* e mensagens, que são transmitidas sincronizando os processos e enviando mensagens de controle. Estas características não tornam a coleta de lixo ótima interessante para ser utilizada com protocolos assíncronos e quase-síncronos, que não utilizam nenhum tipo de sincronização ou mensagem de controle.

3.3.4 Coleta de Lixo Local em Padrões RDT

Recentemente, foi provado que é possível realizar a coleta de lixo em padrões de *checkpoints* e mensagens RDT apenas com informações locais a cada processo [42, 43, 44]. Essa coleta local em padrões RDT não identifica todos os *checkpoints* obsoletos, mas garante

que no máximo n *checkpoints* não-obsoletos devem ser mantidos por cada processo da aplicação [42, 43, 44]. Além disso, como a coleta de lixo é feita apenas com informação local, não é necessário nenhum tipo de sincronização ou mensagem de controle.

A coleta de lixo local para padrões RDT é baseada numa idéia semelhante à coleta ótima para padrões RDT mostrada na seção anterior. Ao invés de utilizar o vetor UC para saber o índice do último *checkpoint* estável de um processo, é possível utilizar a entrada para aquele processo no vetor de dependências do *checkpoint* volátil [42, 43, 44]. Assim, um *checkpoint* s_a^α será obsoleto se não existir um processo p_f tal que $s_f^{DV(v_a)[f]-1}$ preceda causalmente $c_a^{\alpha+1}$ e não preceda causalmente s_a^α [42, 43, 44]. Esta relação pode ser expressa apenas com o vetor de dependências. Dessa forma, um *checkpoint* será obsoleto caso não exista um processo p_f tal que:

$$DV(v_a)[f] = DV(c_a^{\alpha+1})[f] \wedge DV(v_a)[f] > DV(s_a^\alpha)[f] \text{ [42, 43, 44]}. \tag{3.2}$$

A Figura 3.10 mostra os *checkpoints* obsoletos identificados (quadrados brancos) pela coleta local no mesmo padrão de *checkpoints* e mensagens RDT da Figura 3.9.

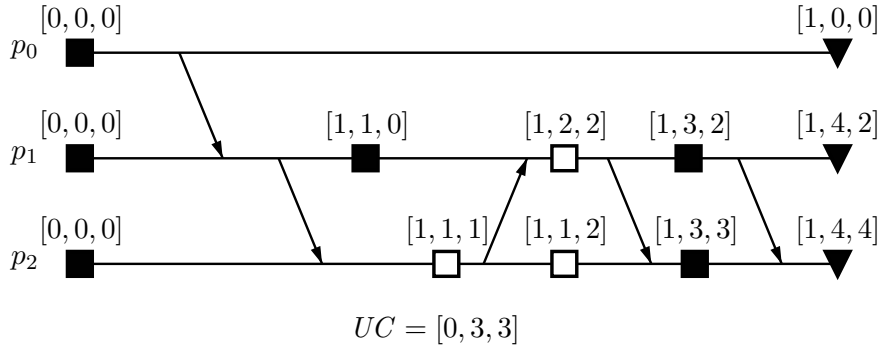


Figura 3.10: Coleta de lixo local em um padrão de *checkpoints* e mensagens RDT.

Algoritmo RDT-LGC

A coleta de lixo local em padrões de *checkpoints* e mensagens RDT pode ser feita em tempo de execução, de forma que os *checkpoints* são eliminados à medida que se tornam obsoletos. O algoritmo que implementa a coleta de lixo local em padrões RDT é chamado de RDT-LGC [42, 43, 44] e faz com que cada processo mantenha um *checkpoint* armazenado para o caso de falha individual em cada um dos n processos da aplicação. Dessa maneira, um processo precisa manter no máximo n *checkpoints* não-obsoletos distintos no meio de armazenamento estável. Como um mesmo *checkpoint* pode pertencer a mais de uma linha

de recuperação, é necessário manter um contador de referências para cada *checkpoint* não-obsoleto armazenado. Quando o contador de referências de um *checkpoint* chega a zero, isto significa que o *checkpoint* não vai ser usado em nenhuma linha de recuperação e, portanto, tornou-se obsoleto e pode ser eliminado.

O Algoritmo 3.1 mostra as estruturas de dados usadas pelo RDT-LGC e alguns procedimentos que as manipulam. Além de um vetor de dependências, o RDT-LGC utiliza duas outras estruturas de dados. Uma delas é o bloco de controle de *checkpoint* (BCC), que contém o índice e o contador de referências de um *checkpoint* armazenado. A outra estrutura de dados é o vetor CM com n ponteiros para blocos de controle de *checkpoints*, sendo que a entrada k aponta para o *checkpoint* não-obsoleto que é mantido por fazer parte da linha de recuperação para uma falha no processo p_k. O procedimento **novo_bcc** é utilizado quando o processo grava um *checkpoint*. Ele aloca um BCC para o novo *checkpoint* de índice ind e coloca a entrada $CM[i]$ apontando para o BCC criado. O procedimento **liga** faz com que a entrada $CM[i]$ aponte para o mesmo BCC que a entrada $CM[j]$, incrementando o contador de referências deste BCC. O procedimento **libera** apaga a entrada $CM[i]$, ou seja, apaga a referência dessa entrada a um BCC existente. Além disso, esse procedimento decrementa o contador de referências do BCC apontado, e caso o resultado do decremento seja zero, o *checkpoint* referente àquele BCC será eliminado do meio de armazenamento estável, pois se tornou obsoleto, e o BCC será desalocado.

Dado que a Equação 3.2 para verificar se um *checkpoint* é obsoleto depende totalmente dos vetores de dependências dos *checkpoints* estáveis e do vetor de dependências do *checkpoint* volátil, o conjunto de *checkpoints* a serem mantidos só pode mudar quando um novo *checkpoint* é gravado, ou quando uma nova dependência causal é recebida em uma mensagem.

Quando o processo p_i grava um *checkpoint*, a entrada $DV[i]$ é alterada, e após a gravação do *checkpoint* e atualização de $DV[i]$ obtém-se:

$$DV(v_i)[i] = DV(v_i)[i] \wedge DV(v_i)[i] > DV(s_i^{last})[i]. \tag{3.3}$$

Pode-se dizer que o processo p_i está impedindo que o *checkpoint* s_i^{last} seja considerado obsoleto, pois o *checkpoint* s_i^{last} será usado na linha de recuperação para uma falha no próprio processo p_i, já que é o último *checkpoint* estável desse processo. Nesse caso, a entrada $CM[i]$ deve ser sempre atualizada para o BCC do último *checkpoint* estável do processo p_i, como pode ser visto no Algoritmo 3.2.

Além disso, quando o processo p_i recebe uma mensagem m e percebe uma nova dependência causal do processo p_j, depois de atualizar o vetor de dependências obtém-se:

$$DV(v_i)[j] = DV(v_i)[j] \wedge DV(v_i)[j] > DV(s_i^{last})[j]. \tag{3.4}$$

É possível dizer que o processo p_j está impedindo que o *checkpoint* s_i^{last} seja considerado obsoleto, pois o *checkpoint* s_i^{last} pode ser usado na linha de recuperação para uma falha

Algoritmo 3.1 Estruturas de dados e procedimentos do RDT-LGC.

Estruturas de Dados

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

Procedimento inicializa() {inicializa vetores CM e DV}

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

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

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

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

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

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

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

em p_j, já que os *checkpoints* estáveis posteriores de p_i serão precedidos causalmente pelo último *checkpoint* estável de p_j informado pela mensagem m. Nesse caso, a entrada $CM[j]$ deve ser sempre atualizada para o BCC do último *checkpoint* estável do processo p_i, como pode ser observado no Algoritmo 3.2.

O RDT-LGC segue executando como está mostrado no Algoritmo 3.2 até que uma falha ocorra no processamento. Durante a sessão de recuperação, os processos que não precisarem retroceder, ou seja, tiverem o *checkpoint* volátil na linha de recuperação, podem aproveitar e realizar a coleta de todos os *checkpoints* obsoletos que não foram eliminados pelo RDT-LGC. Para tanto, pode-se usar as próprias estruturas do RDT-LGC e um vetor UI com o índide do último intervalo entre *checkpoints* de cada processo. Esse vetor UI deve ser propagado para todos os processos da aplicação juntamente com a

Algoritmo 3.2 RDT-LGC em um processo p_i.

Inicialização

1: inicializa() {inicializa CM e DV}

Ao enviar mensagem m

1: **for** $j \leftarrow 0$ **to** $n-1$ **do**
2: $m.DV[j] \leftarrow DV[j]$
3: *envia*(m)

Ao receber mensagem m

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

Ao gravar *checkpoint*

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

linha de recuperação. Assim, quando $DV[j] < UI[j]$ em um processo p_i, pode-se concluir que $s_j^{last} \not\rightarrow v_i$, e portanto nenhum *checkpoint* precisa ser mantido por causa do processo p_j e a entrada $CM[j]$ pode ser liberada. Esse algoritmo de coleta ótima usando as estruturas do RDT-LGC está melhor detalhado no Algoritmo 3.3.

Algoritmo 3.3 Coleta de lixo ótima usando as estruturas do RDT-LGC.

Procedimento coleta_ótima(UI**: array**$[0 \ldots n-1]$ **of** ***inteiro***)

1: **for** $j \leftarrow 0$ **to** $n-1$ **do**
2: **if** $DV[j] < UI[j]$ **then**
3: libera(j) {libera entradas obsoletas}

Por outro lado, os processos que precisam retroceder para um *checkpoint* estável necessitam executar o Algoritmo 3.4 para ajustar as estruturas do RDT-LGC. Além da linha de recuperação e do vetor UI, esse algoritmo também necessita saber o índice do *checkpoint* para o qual o processo deve retroceder, chamado de índice de retrocesso, ou simplesmente IR. Dessa forma, depois de começada a recuperação, o processo p_i elimina os *checkpoints* posteriores à linha de recuperação e reconstrói o seu vetor de dependências a partir do vetor de s_i^{IR} (linhas 4-6). Caso p_i tenha falhado, é preciso alocar BCCs para os *checkpoints* armazenados em meio estável (linha 7). Caso p_i não tenha falhado, esse

passo do algoritmo não é necessário, pois os *BCC*s já estão na memória. No entanto, nesse último caso é preciso zerar os contadores de referência de cada *BCC*. Depois disso, as entradas do vetor *CM* são atualizadas para o *BCC* dos *checkpoints* que devem ser mantidos (linas 8-14). Para cada processo p_f, a condição da Equação 3.2 usando o vetor *UI* no lugar do vetor de dependências do *checkpoint* volátil ($DV(v_i)$) é testada para achar o *checkpoint* s_i^α que precisa ser mantido. Ao final, todos os *BCC*s que tiverem o contador de referências zerado são desalocados e os *checkpoints* correspondentes eliminados, já que são obsoletos (linhas 15-17).

Algoritmo 3.4 RDT-LGC durante um retrocesso em um processo p_i.

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

O algoritmo de coleta ótima usando as estruturas do RDT-LGC e o do RDT-LGC durante um retrocesso são usados quando informações globais sobre o padrão de *checkpoints* e mensagens podem ser enviadas para todos os processos em uma sessão de recuperação. Em outras palavras, esses algoritmos são usados quando um mecanismo síncrono de recuperação é utilizado. Quando os processos não têm acesso a informações globais do CCP na sessão de recuperação, como em um mecanismo de recuperação assíncrono, os processos que não são retrocedidos continuam a execução normal do RDT-LGC e aqueles que necessitam retroceder executam o Algoritmo 3.4, mas trocando o vetor *UI* por $DV(v_i)$ na linha nove.

3.4 Sumário

Neste capítulo foram mostrados meios de se calcular a linha de recuperação para qualquer padrão de *checkpoints* e mensagens e também métodos mais fáceis que podem ser utilizados em padrões produzidos por protocolos quase-síncronos RDT. Além da linha de recuperação para padrões da classe RDT ser mais fácil de se calcular, a distância de retrocesso dos protocolos RDT da classe ZPF é $n-1$ vezes menor que a dos protocolos da classe ZCF [6], o que representa uma vantagem na utilização de protocolos RDT da classe ZPF na recuperação por retrocesso de estado.

Além dos algoritmos de recuperação, também foram vistos os algoritmos de coleta de lixo. Quando um protocolo de *checkpointing* síncrono é utlizado, a coleta de lixo é totalmente simplificada, pois cada execução do protocolo produz um *checkpoint* global consistente, tornando obsoletos todos os *checkpoints* anteriores, que podem ser eliminados do meio estável. Por outro lado, quando um protocolo assíncrono ou quase-síncrono é utilizado, a coleta de lixo geralmente precisa de informações globais do padrão de *checkpoints* e mensagens para que possa ser realizada. No entanto, a classe de protocolos quase-síncronos RDT possui um algoritmo de coleta de lixo que pode ser realizado localmente em cada processo [42, 43, 44]. Este algoritmo é chamado de RDT-LGC, e estabelece um limite máximo de n *checkpoints* mantidos por processo no decorrer da computação distribuída.

Capítulo 4

Trabalhos Relacionados

Com a crescente utilização de grandes sistemas distribuídos, como *clusters* e *grids*, a preocupação com a tolerância a falhas é cada vez maior. Dessa forma, é possível encontrar na literatura vários trabalhos sobre tolerância a falhas para aplicações distribuídas. Técnicas como *checkpointing* e *message logging* são utilizadas para que falhas em um processamento distribuído não obrigue o reinício da aplicação.

Aplicações distribuídas normalmente utilizam alguma biblioteca de passagem de mensagens para que os seus processos possam se comunicar. Atualmente, o padrão mais utilizado para uma biblioteca de passagem de mensagens é o MPI [2]. Assim, os trabalhos em tolerância a falhas em sistemas distribuídos estão intimamente relacionados com a biblioteca de passagem de mensagens utilizada e muitas vezes são implementados dentro dela. O RENEW [38], por exemplo, é um ambiente que possui uma interface MPI, mas que foi projetado para a fácil implementação e teste de novos protocolos de *checkpointing* e de recuperação por retrocesso de estado.

Embora o padrão MPI seja amplamente utilizado, existem outras bibliotecas de passagem de mensagens como o PVM [3] e implementações do modelo BSP que também são utilizadas. O InteGrade [28] possui uma implementação da biblioteca BSP, que utiliza *checkpointing* síncrono para prover tolerância a falhas. Além disso, esta implementação de *checkpointing* para aplicações BSP no InteGrade é feita no nível da aplicação, usando um pré-compilador que insere o código necessário para gravar e recuperar o estado dos processos [20].

Existem muitos trabalhos na área de tolerância a falhas para aplicações distribuídas que utilizam o MPI como biblioteca de passagem de mensagens. Desde aquelas que são tolerantes a falhas na comunicação como o LA-MPI [10] até outras que permitem que a própria aplicação tenha controle de como se recuperar das falhas como o FT-MPI [23]. Nas próximas seções, serão comentados alguns dos trabalhos mais interessantes para esta

dissertação, que empregam *checkpointing* para migrar processos e realizar a recuperação de falhas por retrocesso de estado de aplicações distribuídas.

4.1 Condor

O Condor [33, 34] é um ambiente distribuído de execução de processos desenvolvido para sistemas operacionais UNIX. Este sistema foi projetado para executar tarefas em computadores que estejam ociosos em uma rede, de forma a utilizar mais eficientemente os recursos computacionais. Além disso, os usuários dos computadores que fazem parte do conjunto utilizado pelo Condor não são prejudicados, pois quando estes voltam a utilizar os seus computadores, os processos do Condor são migrados para outras máquinas ociosas.

Para conseguir migrar um processo de uma máquina para outra, o Condor grava o estado do processo na forma de um *checkpoint* antes de migrá-lo. O processo de gravação do *checkpoint* está implementado na biblioteca de *checkpointing* do Condor, que inclusive permite que o próprio processo acione a gravação do seu estado. Para que o processo de *checkpointing* possa funcionar, os programas submetidos para serem executados no Condor devem ser ligados estaticamente com a biblioteca de *checkpointing*.

O processo de *checkpointing* no ambiente Condor é transparente para a aplicação, ou seja, o código do usuário não precisa ser escrito com a preocupação de que pode ser migrado. Aliás, um programa executando no Condor não tem nenhum conhecimento sobre a migração de uma máquina para outra. O mecanismo de gravação do *checkpoint* é realizado totalmente no modo usuário e nenhuma modificação é necessária no núcleo do sistema operacional.

A gravação do estado do processo é feita no tratador de um determinado sinal, e ao ser reiniciada, a aplicação aparenta estar retornando daquele tratador. O estado do processo é escrito em um arquivo ou em um *socket*, que é lido no local onde o processo vai ser reiniciado. Durante a gravação do estado do processo, várias partes são levadas em consideração como as áreas de código e dados da aplicação, a pilha de execução, as referências a bibliotecas compartilhadas, arquivos, os sinais pendentes e o estado do processador.

Além de ser usado para a migração de processos, o mecanismo de *checkpointing* do Condor pode gravar *checkpoints* periodicamente da aplicação, com a finalidade de usar o *checkpoint* mais recente na recuperação de uma falha por retrocesso de estado. Porém, este mecanismo de gravação de *checkpoint* dos processos do Condor possui algumas limitações. Como a biblioteca de *checkpointing* executa sem nenhuma ajuda do sistema operacional, existem determinadas informações que são impossíveis de serem capturadas em modo usuário. Além disso, o Condor não é capaz de gravar um *checkpoint* global consistente de uma aplicação que possua vários processos, que se comunicam entre si utilizando algum

mecanismo de passagem de mensagens. Uma outra limitação é a necessidade de ligar o programa submetido com a biblioteca de *checkpointing* do Condor.

4.2 CoCheck

O CoCheck [45] é um ambiente que executa tarefas paralelas utilizando o tempo ocioso de máquinas em uma rede. Assim como o Condor, os processos executados pelo CoCheck devem ser migrados da máquina quando o usuário volta a utilizá-la. A migração dos programas no CoCheck é feita de forma transparente para a aplicação distribuída utilizando *checkpointing*. Além disso, o estado da computação pode ser gravado periodicamente, de modo a poder ser recuperado no caso de uma falha interromper o processamento.

Um dos objetivos do CoCheck é ser um ambiente facilmente portável para qualquer biblioteca de passagem de mensagens, e já foi implementado para bibliotecas MPI e PVM. Para garantir a portabilidade, o CoCheck é implementado sobre a biblioteca de passagem de mensagens e não dentro dela. Além disso, o CoCheck pode utilizar qualquer mecanismo que permita gravar o estado de um processo localmente, como o próprio Condor. Para que um processo possa ter seu estado gravado ou possa ser migrado, o código do CoCheck precisa ser compilado e ligado juntamente com o código do programa do usuário.

O algoritmo de *checkpointing* utilizado pelo CoCheck é parecido com o de Chandy e Lamport que foi mostrado na Seção 2.7.1. Um processo externo controla a invocação do algoritmo. No momento que cada processo da aplicação recebe a notificação de gravação de um *snapshot* consistente, ele deve enviar uma mensagem de controle por todos os seus canais de comunicação e gravar o estado dos canais até receber novamente a mesma mensagem de controle por todos os eles. Após salvar o estado de todos os seus canais de comunicação, cada processo pode gravar o seu estado na forma de um *checkpoint*. Quando a aplicação for retrocedida para algum *snapshot* consistente gravado ou quando ela for reiniciada depois de uma migração, os processos recuperam o seu estado e começam a ler as mensagens gravadas dos canais antes de utilizarem o canal de comunicação propriamente dito.

Embora o CoCheck tenha sido projetado para ser portável, a sua implementação em uma biblioteca de passagem de mensagens do padrão MPI necessita ser feita dentro do próprio código da biblioteca. O problema que dificulta a implementação do CoCheck sobre a biblioteca MPI são algumas definições do próprio padrão MPI. Assim, a implementação do CoCheck para esse tipo de biblioteca de passagem de mensagens necessita ser feita para cada uma que se deseje utilizar.

4.3 Starfish

O Starfish [7] é um ambiente para executar programas MPI em *clusters*. Este ambiente foi projetado para obter o máximo desempenho na execução das aplicações distribuídas. Além disso, a arquitetura do ambiente é totalmente modular, permitindo que várias tecnologias de comunicação sejam usadas e que protocolos de *checkpointing* síncronos ou assíncronos sejam implementados. Assim, é possível comparar a execução de uma mesma aplicação utilizando protocolos de *checkpointing* diferentes.

As aplicações que rodam no Starfish podem utilizar algumas funcionalidades que não fazem parte do padrão MPI, embora aplicações que sigam estritamente o padrão MPI também possam ser executadas. No entanto, elas não poderão acionar o protocolo de *checkpointing* dentro do código da própria aplicação e não poderão ser avisadas caso o grupo de processos mude devido a alguma falha em uma das máquinas do *cluster*. Ainda assim, o Starfish possui um mecanismo de *checkpointing* que pode ser acionado externamente à aplicação, e que pode ser usado para prover tolerância a falhas para aplicações que não utilizem as funcionalidades extras do Starfish.

Em cada uma das máquinas do *cluster*, o Starfish possui um *daemon* que controla as ações locais, como executar os processos MPI e notificar a ocorrência de alguma falha. Os *daemons* do Starfish em todas as máquinas usam um mecanismo de comunicação de grupo para interagirem entre si. Dessa forma, o administrador do *cluster* pode controlar o ambiente conectando-se a qualquer um desses *daemons* para adicionar ou remover máquinas e também para controlar parâmetros do *cluster*. O mecanismo de comunicação de grupo garante a consistência entre as visões do sistema que os *daemons* possuem, fazendo com que uma alteração seja atomicamente propagada para todo o ambiente.

Mesmo que algumas máquinas do *cluster* parem de funcionar, este fato não compromete o sistema inteiro, e o Starfish continua a executar aplicações. Quando nenhum processo de uma aplicação estava na máquina que falhou, aquela aplicação continua a executar sem perceber a falha. No caso em que algum processo da aplicação estava naquela máquina falha, o Starfish permite que aquele processo seja reiniciado em outro lugar ou que a aplicação seja avisada, para que ela possa reestruturar o seu processamento. Neste último caso, um evento de mudança de visão é enviado para o grupo dos processos que compõem a aplicação e que ainda continuam executando. A aplicação deve registrar que deseja receber essas notificações de quando a visão do seu grupo de processos é alterada por algum motivo. Geralmente, as aplicações que são trivialmente paralelizadas, ou seja, aquelas que apenas repartem o processamento, podem utilizar essa funcionalidade do Starfish.

Além disso, quando um programa MPI é submetido para ser executado no Starfish, a política de tolerância a falhas pode ser escolhida. Entre as políticas existentes estão

o reinício automático da aplicação ou a notificação de mudanças na visão do grupo de processos. Também é possível determinar a regra que define qual máquina será escolhida para reiniciar um processo da aplicação que estava numa máquina que falhou. Por fim, para compatibilidade com o padrão MPI, também é possível escolher que a aplicação seja toda terminada quando algum de seus processos falha no meio da sua execução.

4.4 MPICH-V

O MPICH-V [15] é uma biblioteca de passagem de mensagens que foi projetada para ser utilizada em grandes *clusters* e *grids*, que estão sujeitos a falhas freqüentes. Trata-se de uma implementação do padrão MPI, e assim, qualquer aplicação MPI pode ser executada pelo MPICH-V, bastando que a aplicação seja ligada com a biblioteca do MPICH-V.

Dado que o MPICH-V foi feito para ser utilizado em grandes sistemas distribuídos, uma arquitetura totalmente distribuída foi pensada para prover tolerância a falhas. Assim, um protocolo assíncrono de *checkpointing* e um algoritmo de *message logging* são utilizados, pois qualquer tipo de sincronização global iria prejudicar o desempenho da execução da aplicação. Além disso, como as falhas são mais freqüentes, uma máquina poderia falhar durante a sincronização dos processos. O mecanismo de tolerância a falhas do MPICH-V é transparente para as aplicações, e é capaz de tolerar falhas em todos os processos da aplicação. Outra característica é que nenhuma modificação é necessária no sistema operacional, pois esse mecanismo utiliza apenas bibliotecas que executam em modo usuário.

A arquitetura do MPICH-V é composta por três tipos de processos, além daqueles processos MPI que constituem as aplicações que estão executando. Estes processos implementam as funcionalidades básicas de uma biblioteca de passagem de mensagens, dando ênfase na tolerância a falhas. Esses tipos de processo são o *dispatcher*, os *channel memories* (CM) e os *checkpoint servers* (CS).

O *dispatcher* é responsável por executar os processos MPI das aplicações nas máquinas disponíveis. Além disso, este processo também monitora os processos da aplicação e detecta possíveis falhas nas máquinas ou na rede de comunicação. O *dispatcher* espera receber periodicamente mensagens dos processos da aplicação indicando que estes não falharam. Assim, quando uma destas mensagens de algum dos processos da aplicação não chega no tempo esperado, um falha pode ter ocorrido. Quando uma falha é detectada, o mecanismo de recuperação é acionado, que cria um novo processo MPI para retomar a execução do ponto mais recente possível, utilizando um *checkpoint* daquele processo falho que foi gravado anteriormente.

A gravação do estado dos processos e das mensagens enviadas pela aplicação é feita de forma totalmente descentralizada no MPICH-V. Os *channel memories* implementam

a transmissão de mensagens entre os processos da aplicação, e também armazenam as mensagens enviadas para que estas possam ser usadas numa sessão de recuperação futura. Cada processo da aplicação possui um *channel memory* associado, e um processo envia uma mensagem para outro, entregando-a ao *channel memory* do processo que deve receber a mensagem. Além de gravar o estado dos canais de comunicação nos *channel memories*, os estados dos processos da aplicação são periodicamente gravados nos chamados *checkpoint servers*. Dessa forma, na ocorrência de uma falha, o *channel memory* e o *checkpoint server* do processo falho devem se coordenar, para que a aplicação seja retrocedida para um *snapshot* consistente. Além disso, como os *checkpoints* dos processos são gravados nos *checkpoint servers*, a migração de processos da aplicação é facilmente realizada quando necessário.

O MPICH-V utiliza a biblioteca de *checkpointing* do Condor para gravar o estado dos processos da aplicação. No entanto, como o Condor não grava o estado dos *sockets* dos processos, quando um processo MPI é reiniciado no MPICH-V, as conexões entre aquele processo e o seu *channel memory* devem ser refeitas. Além disso, a operação de gravar um *checkpoint* de um processo normalmente requer que sua execução seja interrompida. Assim, gerar o *checkpoint* de um processo e enviá-lo para o *checkpoint server* pode demorar muito. Para resolver esse problema, o MPICH-V cria um novo processo idêntico ao anterior utilizando a chamada de sistema `fork()` comum em sistemas operacionais UNIX, grava o *checkpoint* desse novo processo e o envia ao *checkpoint server*, enquanto o processo original pode continuar normalmente a sua execução.

4.5 MPICH-GF

O MPICH-GF [50] é um ambiente que oferece tolerância a falhas de forma transparente para aplicações MPI executadas em um *grid* controlado pelo Globus [4]. Um protocolo de *checkpointing* síncrono é utilizado, e o Globus foi alterado de modo a suportar a recuperação por retrocesso de estado de aplicações MPI.

A arquitetura do MPICH-GF para a execução tolerante a falhas de aplicações MPI é composta por gerentes hierárquicos. Existe um gerente central que é responsável por tratar falhas no *hardware* ou na rede de comunicação e gerentes locais executam nas máquinas do *grid*, tratando as falhas ocorridas nos processos da aplicação. Os gerentes locais e o gerente central também são responsáveis pelo protocolo de *checkpointing* e pela recuperação automática das aplicações, na ocorrência de uma falha.

O protocolo síncrono de *checkpointing* utilizado no MPICH-GF é muito similar ao do CoCheck. O gerente central inicia o protocolo de *checkpointing* periodicamente, avisando os gerentes locais da gravação de um novo *snapshot* consistente. Os gerentes locais enviam sinais para os processos MPI, para que estes estejam prontos para gravar os seus *check-*

points. Quando um processo da aplicação recebe o sinal indicando o início do protocolo de *checkpointing*, ele executa uma função que garante que todas as mensagens já enviadas foram recebidas pelos processos receptores. Depois disso, como o estado dos canais de comunicação já está gravado de forma distribuída em cada processo da aplicação, um *checkpoint* de cada processo é gravado. Após a gravação de cada *checkpoint*, o gerente local é avisado e este avisa o gerente central, que pode então saber quando a gravação do novo *snapshot* consistente realmente terminou.

Os processos da aplicação são gerados a partir do gerente local de cada máquina usando as chamadas de sistema `fork()` e `exec()`. Assim, os gerentes locais são avisados quando um dos processos MPI acaba, e pode avaliar se alguma falha ocorreu ou não. Caso alguma falha no processo tenha ocorrido, o gerente local notifica o gerente central, para que este último possa iniciar uma sessão de recuperação. Além disso, o gerente central monitora os gerentes locais com mensagens periódicas. Caso algum gerente local não responda dentro do limite de tempo a uma dessas mensagens, o gerente central também inicia uma sessão de recuperação dos processos MPI rodando naquela máquina a partir do último *snapshot* consistente gravado. O MPICH-GF não trata falhas no gerente central, embora uma falha nesse processo possa ser resolvida utilizando-se redundância.

Quando um protocolo de *checkpointing* síncrono é utilizado, a falha em um processo acarreta um retrocesso em toda a aplicação. Assim, quando o gerente central detecta uma falha, uma mensagem é enviada para todos os processos que ainda estão executando, avisando que um retrocesso para o último *snapshot* consistente é necessário. Quando a aplicação é reiniciada, os canais de comunicação entre cada par de processos são refeitos antes de cada processo continuar a sua execução normal.

4.6 LAM/MPI

O LAM/MPI [1] é uma implementação livre do padrão MPI que possui um mecanismo de tolerância a falhas que utiliza um protocolo de *checkpointing* síncrono semelhante ao algoritmo de Chandy e Lamport, mostrado na Seção 2.7.1, para gravar um *snapshot* consistente da aplicação [41]. A execução do algoritmo de *checkpointing* é totalmente transparente para a aplicação MPI e um sistema chamado BLCR [21] é utilizado para gravar o estado dos processos da aplicação.

O BLCR é composto por módulos para o *kernel* Linux [5], que possibilitam a gravação do estado de um processo em um arquivo e também o reinício daquele processo a partir do arquivo gravado. Para que as aplicações possam interagir com a gravação e recuperação dos seus estados, a biblioteca do BLCR permite que a aplicação registre funções que são chamadas quando um *checkpoint* do processo vai ser gravado. É possível registrar uma função que é executada no contexto de um sinal e outra que é executada em uma *thread*

separada. Estas funções permitem que a aplicação trate de recursos que não são gravados pelo BLCR, como *sockets*, e também permitem que na recuperação esses recursos possam ser reativados corretamente. A biblioteca do BLCR também permite que a aplicação execute trechos de código de forma atômica, sem ser interrompida pela gravação de um *checkpoint* do processo.

O componente do LAM/MPI que implementa o protocolo de *checkpointing* é chamado de CR (*Checkpoint/Restart*). Este componente é responsável por coordenar os processos durante a execução do protocolo de *checkpointing* e utilizar um sistema como o BLCR para gravar os estados dos processos. Além do componente CR, um outro componente importante é o RPI (*Request Progression Interface*), encarregado da comunicação entre os processos da aplicação. O componente CR possui uma implementação que utiliza o BLCR e uma outra que utiliza funções definidas pela aplicação para gravar e recuperar o estado dos processos. Já o RPI possui implementações que utilizam como sistema de comunicação o TCP, memória compartilhada, UDP e também existe uma versão que utiliza TCP para ser usada em conjunto com a implementação do CR que utiliza o BLCR. É possível compilar o LAM/MPI com suporte a todas as implementações existentes para os componentes CR e RPI, e escolher qual delas utilizar no momento da submissão da aplicação MPI a ser executada.

Antes que as tarefas possam ser submetidas para que o LAM/MPI as execute, é preciso que as máquinas que vão ser utilizadas estejam rodando o *daemon* do LAM/MPI chamado de `lamd`. O comando `lamboot` se encarrega de executar o `lamd` em todas as máquinas, e o comando `lamhalt` é usado para terminar a execução desses *daemons* quando o usuário não necessita mais submeter programas MPI para serem executados. Os programas são executados no LAM/MPI por meio do programa `mpirun`. Este último se comunica com os `lamd`s para que estes executem os processos MPI nas máquinas utilizando as chamadas de sistema `fork()` e `exec()`. Dessa forma, os `lamd`s são informados quando os processos da aplicação terminam e podem informar possíveis falhas ao `mpirun`. A Figura 4.1 mostra a execução de uma aplicação com dois processos no LAM/MPI. As mensagens trocadas entre os processos da aplicação trafegam pelas conexões diretas existentes entre eles, criadas pelos componentes RPI. A outra conexão mostrada na figura, que conecta todos os processos, inclusive o `mpirun` e os `lamd`s, é usada na sincronização dos processos durante o protocolo de *checkpointing*, que é realizada pelos componentes CR. Além disso, essa última conexão também é utilizada pelo `mpirun` para monitorar a execução dos processos MPI.

No LAM/MPI, o processo `mpirun` é usado para que o usuário possa requisitar que um *snapshot* consistente da aplicação seja gravado. Sendo assim, o estado do processo `mpirun` também é gravado. Na sua inicialização, o `mpirun` registra a sua função que vai executar numa *thread* separada no momento que o protocolo de *checkpointing* for acionado.

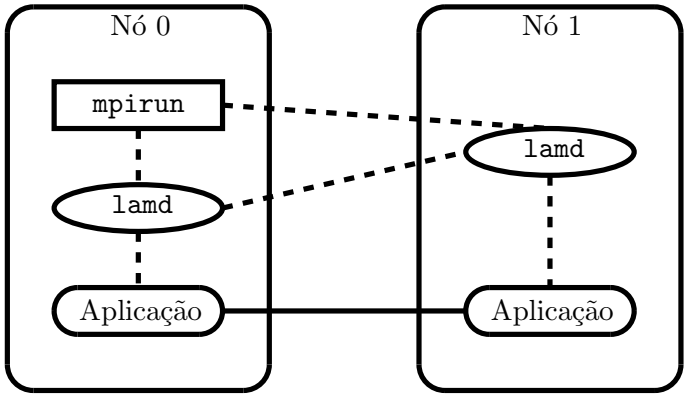


Figura 4.1: Aplicação no LAM/MPI.

Quando o `mpirun` recebe uma requisição para gravar um *snapshot* consistente da aplicação, a função registrada junto ao BLCR começa a executar numa *thread* separada. Esta função grava a topologia dos processos da aplicação, para que esta possa ser reiniciada corretamente na ocorrência de uma falha. Além disso, aquela função também notifica os processos MPI de que estes precisam se coordenar para gravar um *snapshot* consistente da aplicação.

Ao começarem a sua execução, os processos da aplicação devem chamar a função de inicialização da biblioteca MPI, e esta registra as funções que serão chamadas no momento de gravação de um *checkpoint* pelo BLCR. Assim, quando um processo é notificado pelo `mpirun` de que deve proceder com o protocolo de *checkpointing*, a função registrada junto ao BLCR para executar em uma *thread* separada é acionada. A *thread* que executa código da aplicação e a *thread* acionada pelo BLCR para executar o protocolo de *checkpointing* podem executar ao mesmo tempo, mas a *thread* da aplicação não pode executar nenhuma chamada à biblioteca MPI. Sendo assim, a *thread* do protocolo de *checkpointing* bloqueia a *thread* da aplicação de executar chamadas à biblioteca MPI e depois aciona o componente RPI, para que este se sincronize com os demais processos de forma a garantir que todas as mensagens enviadas por qualquer processo da aplicação até aquele momento já foram recebidas. Depois que todas as mensagens da aplicação foram recebidas, cada processo MPI grava o seu estado no meio de armazenamento estável utilizando o BLCR. A seguir, a *thread* da aplicação é desbloqueada e pode continuar executando normalmente, pois a gravação do *snapshot* consistente já foi realizada.

Na ocorrência de uma falha, o processo `mpirun` é avisado pelo `lamd` da máquina em que o processo MPI falhou, e então o `mpirun` termina a execução de toda a aplicação, conforme está especificado no padrão MPI. Posteriormente, quando a aplicação precisar

ser reiniciada a partir de algum *snapshot* consistente gravado, o processo `mpirun` é reiniciado primeiro utilizando-se o BLCR. Depois de ser reiniciado, o `mpirun` percebe que necessita recuperar a aplicação e procede para reiniciar os processos MPI com a mesma topologia que estes estavam executando antes da gravação do *snapshot* consistente. Ao serem reiniciados, os processos MPI estão executando a *thread* que implementa o protocolo de *checkpointing*. Assim, as conexões entre cada par de processos MPI são refeitas, a *thread* da aplicação é desbloqueada e a aplicação pode voltar a executar normalmente.

4.7 Sumário

Como foi visto neste capítulo, é possível encontrar na literatura, vários trabalhos sobre implementações de algoritmos de *checkpointing* síncronos ou assíncronos em bibliotecas de passagem de mensagens [7, 15, 20, 41, 45, 50]. A maioria das implementações de protocolos síncronos não permite que a própria aplicação requisite a gravação do *checkpoint* global consistente ou do *snapshot* consistente, e a estratégia adotada é periodicamente ativar o protocolo de *checkpointing* [7, 41, 45, 50]. Dessa forma, uma degradação exagerada no desempenho da aplicação pode ser provocada. Por outro lado, as implementações que utilizam protocolos assíncronos não podem garantir que a linha de recuperação progrida, e numa recuperação a aplicação pode ter que ser retrocedida até o seu início [7, 15].

Assim, percebeu-se que apesar de existirem muitos estudos teóricos na área de protocolos quase-síncronos de *checkpointing*, poucos são os trabalhos que envolvem implementações desses protocolos quase-síncronos, e aqueles que existem são baseados em simulações, e não na execução de aplicações reais em bibliotecas de passagem de mensagens [9, 47].

Capítulo 5

Curupira

Os algoritmos de *checkpointing* quase-síncronos apresentam algumas vantagens teóricas em relação às outras abordagens por não utilizarem mensagens de controle ou qualquer tipo de sincronização entre os processos durante a computação. Além disso, como foi visto nesta dissertação, os protocolos quase-síncronos da classe RDT oferecem vantanges em relação aos demais protocolos dessa abordagem, pois a coleta de lixo pode ser feita localmente em cada processo, e com um limite máximo de *checkpoints* que necessitam ser mantidos [42, 43, 44]. Outras vantagens já conhecidas da classe RDT sobre as outras classes de protocolos quase-síncronos são a maior facilidade no cálculo de uma linha de recuperação dado um conjunto de processos falhos [48, 49] e também que o retrocesso no caso de uma falha é menor quando um protocolo quase-síncrono RDT é utilizado [6].

Neste capítulo, primeiramente será mostrada a implementação de uma infra-estrutura para *checkpointing* quase-síncrono na biblioteca LAM/MPI [1] de passagem de mensagens. Esta infra-estrutura permite que protocolos quase-síncronos sejam implementados e facilmente testados com aplicações MPI reais. Dessa forma, é possível estimar melhor, por exemplo, o ônus causado pelo protocolo quase-síncrono no desempenho da computação distribuída.

Posteriormente, será descrita uma arquitetura de *software* para recuperação de falhas utilizando protocolos de *checkpointing* quase-síncronos da classe RDT. Esta arquitetura de *software* foi chamada de Curupira, e será comentada a sua implementação realizada na biblioteca de passagem de mensagens LAM/MPI, aproveitando-se a infra-estrutura para *checkpointing* quase-síncrono mencionada anteriormente. Além dos protocolos de *checkpointing* RDT, como o RDT-Minimal [26], o Curupira utiliza o RDT-LGC [42, 43, 44] para realizar a coleta de lixo em tempo de execução e um algoritmo de recuperação por retrocesso é usado em caso de falhas. Por fim, as limitações dessa implementação do Curupira no LAM/MPI são analisadas, e resultados utilizando algumas aplicações MPI também são mostrados.

5.1 *Checkpointing* Quase-Síncrono no LAM/MPI

Uma infra-estrutura para *checkpointing* quase-síncrono foi implementada no LAM/MPI [18]. Optou-se por essa biblioteca MPI, pois ela é amplamente utilizada e já possui implementado um mecanismo que permite requisitar a gravação de um *snapshot* consistente da aplicação, explicado anteriormente na Seção 4.6. Esse mecanismo utiliza o BLCR [21] para gravar os estados dos processos, e foi modificado para que fosse possível a sua utilização em *checkpointing* quase-síncrono.

O BLCR é composto por uma biblioteca que fornece os serviços para *checkpointing* de processos e módulos para serem inseridos no *kernel* Linux. Além da abordagem para *checkpointing* adotada pelo BLCR, que pressupõe a modificação do sistema operacional [21, 51], existem outras duas abordagens. Uma delas exige que a aplicação forneça a implementação das funções para gravar e recuperar o seu estado [20]. A outra abordagem é utilizar uma biblioteca que intercepta as interações entre a aplicação e o sistema operacional, e assim consegue gravar o estado daquele processo [34, 39]. A abordagem no nível da aplicação é a mais portável de todas, mas não é nada transparente para a aplicação. Por outro lado, a abordagem da biblioteca é transparente para a aplicação e bastante portável, mas não consegue gravar todo o estado de um processo. Na infra-estrutura para *checkpointing* quase-síncrono implementada no LAM/MPI decidiu-se por utilizar o próprio BLCR, pois embora ele tenha uma abordagem menos portável, ela permite o acesso a todos os dados de um processo e a operação de gravação e recuperação do estado é totalmente transparente para a aplicação.

Nas próximas seções será mostrado como os protocolos quase-síncronos de *checkpointing* foram modelados, de modo que uma infra-estrutura genérica pudesse ser desenvolvida, e serão detalhadas as alterações feitas no LAM/MPI para que a infra-estrutura fosse implementada. Por fim, uma seção irá comentar as limitações da implementação realizada.

5.1.1 Modelo da Infra-Estrutura

Os protocolos de *checkpointing* quase-síncronos possuem uma certa uniformidade na maneira como eles funcionam. Como regra geral, é preciso que a aplicação consiga gravar *checkpoints* dos estados dos seus processos e que informações de controle sejam enviadas juntamente com todas as mensagens. Estas informações de controle são utilizadas pelo protocolo de *checkpointing* quando uma mensagem é recebida, para decidir se há a necessidade de gravação de um *checkpoint* forçado. Além de conseguir gravar *checkpoints* dos seus processos, a maioria dos protocolos quase-síncronos necessita atualizar as suas estruturas de dados no momento da gravação de um *checkpoint*.

As informações de controle que são enviadas nas mensagens da aplicação são específicas de cada protocolo de *checkpointing* quase-síncrono. Há protocolos que não enviam nenhum

tipo de inforamção de controle, como os protocolos CAS, CASBR, CBR e NRAS [49], que estão mais relacionados com os eventos de envio e recepção de mensagens. Outros protocolos, como o BCS [16], enviam apenas um índice junto com as mensagens. Além desses, existem protocolos quase-síncronos que enviam uma quantidade considerável de informação de controle, como o RDT-Minimal [26] que precisa enviar três vetores. Assim, uma infra-estrutura para *checkpointing* quase-síncrono precisa ser genérica o suficiente para permitir que os protocolos enviem e processem qualquer tipo de informação de controle.

Tendo em vista as necessidades comuns dos protocolos, criou-se um modelo de infra-estrutura para *checkpointing* quase-síncrono em uma biblioteca de passagem de mensagens MPI, que está ilustrado na Figura 5.1. Esta figura mostra que o módulo CKPTQS (*ChecKPoinTing* Quase-Síncrono) deve fazer a interação entre a biblioteca MPI e o protocolo de *checkpointing* quase-síncrono que será utilizado. Ainda na figura, percebe-se que os protocolos de *checkpointing* quase-síncronos foram modelados pelas suas ações, que incluem a inicialização e a finalização do protocolo, a gravação de *checkpoints* básicos e forçados e a manipulação das informações de controle das mensagens. Assim, vários protocolos de *checkpointing* quase-síncronos podem ser utilizados, apenas implementando-se as ações de cada um deles. Também encontra-se modelado naquela figura, as circunstâncias do processamento da aplicação MPI em que cada ação do protocolo quase-síncrono é executada pelo módulo CKPTQS.

5.1.2 Implementação no LAM/MPI

A infra-estrutura para *checkpointing* quase-síncrono implementada no LAM/MPI permite escolher o protocolo a ser utilizado no momento da submissão da tarefa para o ambiente por meio do comando `mpirun`. Essa funcionalidade permite que uma aplicação seja rodada com protocolos quase-síncronos diferentes mais facilmente, pois não é preciso recompilar e instalar todo o LAM/MPI modificado a cada vez que se deseja mudar o protocolo de *checkpointing*. A única exigência é que além das ações definidas no modelo da Figura 5.1, a implementação de cada protocolo de *checkpointing* quase-síncrono necessita informar o seu nome ao módulo CKPTQS. O exemplo a seguir mostra a compilação e a execução de uma aplicação com dez processos no LAM/MPI modificado utilizando o FDAS [49] como o protocolo de *checkpointing* quase-síncrono.

```
$ mpicc programa.c -o programa
$ mpirun -c 10 -ckptqs proto FDAS programa
```

Além do protocolo de *checkpointing* quase-síncrono poder ser escolhido no momento da execução da aplicação, os protocolos foram implementados na infra-estrutura como

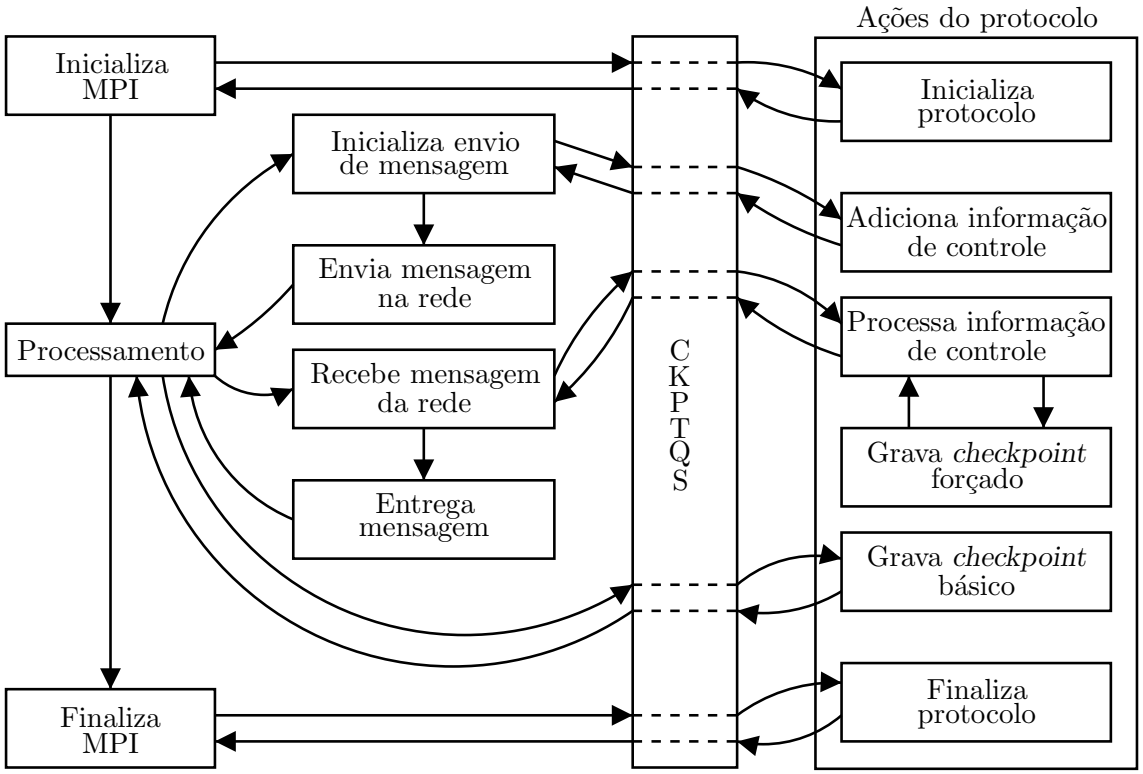


Figura 5.1: Modelo de *checkpointing* quase-síncrono em uma biblioteca MPI.

bibliotecas compartilhadas do GNU/Linux. Assim, os protocolos podem ser carregados dinamicamente no momento da execução da aplicação, e é possível passar para o `mpirun` um repositório de protocolos além dos já disponíveis com a instalação do LAM/MPI modificado. Dessa forma, é possível implementar um protocolo quase-síncrono conhecendo apenas a sua interface com o LAM/MPI modificado, e depois compilá-lo e usá-lo com alguma aplicação MPI, como o exemplo a seguir que utiliza o protocolo BCS [16].

```
$ cp bcs.so /protos/
$ mpirun -c 10 -ckptqs protospath /protos -ckptqs proto BCS programa
```

Também é possível controlar onde serão gravados os *checkpoints* de cada processo da aplicação MPI por meio de uma opção passada para o `mpirun`. Dessa maneira, consegue-se arrumar a gravação dos *checkpoints* de maneira centralizada, caso um sistema de arquivos distribuído seja utilizado, como o GFS ou mesmo o NFS. Caso o local informado ao `mpirun` para a gravação dos *checkpoints* não seja acessível por todos os processos, os *checkpoints* serão gravados localmente em cada uma das máquinas. O exemplo a seguir

mostra a utilização dessa opção juntamente com o uso do protocolo de *checkpointing* quase-síncrono CBR [49].

```
$ mpirun -c 10 -ckptqs ckptsdir /gfs -ckptqs proto CBR programa
```

Todas as opções do CKPTQS passadas ao `mpirun` são informadas aos processos da aplicação, para que a biblioteca MPI ao ser inicializada possa saber que protocolo utilizar, onde procurar por protocolos e também onde salvar os *checkpoints* daquele processo. Além das mudanças realizadas no `mpirun` para mandar opções para o módulo CKPTQS de cada processo da aplicação, outras mudanças foram feitas nos módulos CR e RPI que implementam o protocolo de *checkpointing* síncrono bloqueante do LAM/MPI. Essas mudanças serão detalhadas nas próximas seções e estão relacionadas com a inicialização e finalização do protocolo de *checkpointing* quase-síncrono, com a gravação de *checkpoints* e também com o envio e o processamento das informações de controle do protocolo utilizado.

Inicialização e Finalização do Protocolo de *Checkpointing* Quase-Síncrono

Para que o CKPTQS possa executar a ação de inicialização e posteriormente a de finalização do protocolo quase-síncrono, foram modificadas as implementações das funções `MPI_Init()` e `MPI_Finalize()`, respectivamente, da biblioteca do LAM/MPI.

A função `MPI_Init()` é executada por todos os processos da aplicação e inicializa todo o ambiente para que mensagens possam ser enviadas e recebidas. Assim, dentro do `MPI_Init()` foi colocada uma chamada para a função de inicialização do módulo CKPTQS. Esta função está situada após uma barreira distribuída de sincronização, que bloqueia o processo corrente até que todos os processos da aplicação cheguem naquele ponto da execução. Esse cuidado é necessário para que mensagens que contenham informações de controle não sejam enviadas a processos que ainda não inicializaram o protocolo de *checkpointing*. Ao ser executada, a função de inicialização do CKPTQS procura o protocolo quase-síncrono que foi passado pelo `mpirun` (ou utiliza o padrão) e executa a ação de inicialização dele.

A maioria dos protocolos de *checkpointing* quase-síncronos precisa identificar univocamente cada um dos processos da aplicação e também necessita saber o número total de processos. A implementação de grupos de processos no LAM/MPI possui essas informações, que são capturadas pelo CKPTQS e passadas para a ação de inicialização do protocolo quase-síncrono.

O padrão MPI [2] permite a criação de grupos de processos para que comunicações possam ser feitas apenas entre membros do grupo. Dentro de um grupo, todos os processos são identificados por índices e todos os grupos são identificados por um comunicador, que guarda o índice do processo corrente naquele grupo e quantos integrantes existem nele.

Os comunicadores são utilizados pelos processos para que eles se comuniquem entre si dentro do mesmo grupo. Além disso, todos os processos de uma aplicação MPI estão dentro de um grupo relacionado ao comunicador `MPI_COMM_WORLD`. Sendo assim, o índice do processo corrente e o número de processos no grupo do comunicador `MPI_COMM_WORLD` são os valores capturados pelo CKPTQS e repassados para a ação de inicialização do protocolo quase-síncrono.

Ao final da computação distribuída, todos os processos de uma aplicação MPI devem executar a função `MPI_Finalize()`, para que o ambiente seja terminado corretamente. Assim, uma chamada para a função de finalização do CKPTQS foi introduzida no `MPI_Finalize()`, depois de uma barreira distribuída de sincronização, para garantir que nenhuma mensagem contendo informação de controle do protocolo de *checkpointing* seja recebida por um processo após este ter finalizado localmente a execução do protocolo quase-síncrono. Ao ser executada, a função de finalização do CKPTQS executa a ação de finalização do protocolo de *checkpointing* quase-síncrono utilizado.

Gravação de *Checkpoints*

A implementação do protocolo de *checkpointing* síncrono bloqueante do LAM/MPI grava o estado de todos os canais de comunicação entre cada par de processos, e posteriormente utiliza o BLCR para gravar localmente o estado de cada processo. Assim, na infra-estrutura para *checkpointing* quase-síncrono, a coordenação entre os processos para que os canais de comunicação fossem gravados foi retirada. Dessa maneira, quando a gravação de um *checkpoint* básico é requisitada por um processo da aplicação ou quando um *checkpoint* forçado precisa ser gravado, apenas aquele processo grava o seu estado, sem nenhum tipo de mensagem de controle ou sincronização.

O padrão MPI não prevê nenhum tipo de tolerância a falhas ou mesmo alguma primitva para a gravação do estado de um processo, assim não existe nenhuma função na biblioteca do LAM/MPI que uma aplicação possa utilizar para gravar o seu estado. Sendo assim, uma função chamada `MPI_Checkpoint()` foi criada para que a aplicação pudesse requisitar a gravação de um *checkpoint* básico para a infra-estrutura de *checkpointing* quase-síncrono. Esta função recorre ao CKPTQS, que por sua vez executa a ação de gravação de um *checkpoint* básico do protocolo quase-síncrono que está sendo utilizado.

O LAM/MPI permite que um processo de uma aplicação distribuída tenha várias *threads* de execução. No entanto, a função `MPI_Checkpoint()` não pode ser chamada por duas *threads* de forma concorrente e também não pode executar ao mesmo tempo que outras funções da biblioteca do LAM/MPI, pois isto poderia gerar uma inconsistência no estado das estruturas de dados do protocolo de *checkpointing* que seria gravado no *checkpoint*. Embora permita a execução de várias *threads*, o LAM/MPI não permite que duas *threads* executem dentro da biblioteca MPI simultaneamente. Este controle é feito

utilizando-se um *mutex* que é pego na entrada de todas as funções da biblioteca MPI e que é liberado antes dessas funções retornarem para a aplicação. Assim, o problema com as várias *threads* de execução e a função `MPI_Checkpoint()` foi solucionada pegando e liberando aquele *mutex* global da mesma forma que as funções da biblioteca do LAM/MPI.

As ações de gravar um *checkpoint*, básico ou forçado, de um um protocolo quase-síncrono não gravam o *checkpoint* diretamente. Estas ações são executadas para que o protocolo quase-síncrono possa atualizar as suas estruturas de dados no momento de gravação de um *checkpoint* e então chamar uma função do módulo CKPTQS chamada `ckptqs_save_ckpt()` para realmente gravar o *checkpoint* do processo. A função `ckptqs_save_ckpt()` grava os *checkpoints* do processo em um diretório criado dentro do local indicado pelo `mpirun` (ou no local padrão). Este diretório é criado com o nome contendo o identificador daquele processo no sistema operacional mais o índice deste mesmo processo no grupo do comunicador `MPI_COMM_WORLD`. Um exemplo de nome de diretório é `ckptqs-1024-0`, que é o local onde os *checkpoints* do processo de índice zero da aplicação MPI e com identificador `1024` no sistema operacional serão gravados. Dessa maneira, os *checkpoints* de processos diferentes não serão confundidos, caso um esquema centralizado de gravação esteja sendo utilizado.

No LAM/MPI, todos os processos de uma aplicação registram uma *thread* com o BLCR, que é executada quando um *checkpoint* vai ser gravado. Esta *thread* sinaliza para o BLCR quando um *checkpoint* do processo corrente pode ser gravado executando uma função da biblioteca do BLCR chamada `rc_checkpoint()`. Após o retorno da função `rc_checkpoint()`, o *checkpoint* do processo já foi gravado. Como a *thread* do BLCR executa ao mesmo tempo que a *thread* da aplicação, esta última precisa ser bloqueada quando um *checkpoint* está sendo gravado, para que o estado do processo não seja alterado enquanto ele é salvo. Por este motivo, a função `ckptqs_save_ckpt()`, que executa na *thread* da aplicação, e a *thread* do BLCR precisam ter a sua execução sincronizada. Esta sincronização é feita utilizando-se o *mutex* global do LAM/MPI, para garantir que a *thread* da aplicação bloqueia a sua execução depois que a função `ckptqs_save_ckpt()` envia a requisição de gravação do *checkpoint* para o BLCR, permitindo que a *thread* do BLCR passe a executar.

Envio de Informação de Controle

Depois que a infra-estrutura para *checkpointing* quase-síncrono já foi completamente inicializada pelo módulo CKPTQS, todas as mensagens enviadas pela aplicação precisam levar as informações de controle do protocolo de *checkpointing* utilizado.

As mensagens enviadas pelos processos da aplicação no LAM/MPI possuem duas partes principais: o cabeçalho e o corpo da mensagem. O cabeçalho é chamado de envelope, possui um tamanho fixo conhecido previamente e carrega informações sobre a mensagem.

O corpo da mensagem tem o seu tamanho indicado no envelope e carrega os dados da aplicação. Na transmissão da mensagem, o envelope é enviado em primeiro lugar, e depois o corpo da mensagem é transmitido.

Para enviar a informação de controle do protocolo quase-síncrono junto com as mensagens da aplicação, uma outra parte foi criada nas mensagens. Esta parte se chama `cinfo` e ela é enviada logo após o envelope. Para que o processo que está recebendo a mensagem saiba o tamanho do `cinfo`, este valor foi colocado no envelope de todas as mensagens enviadas. Para colocar a informação de controle do protocolo quase-síncrono na mensagem, uma função do módulo CKPTQS é chamada assim que a mensagem vai ser enviada. Esta função se encarrega de executar a ação de adicionar informação de controle do protocolo utilizado, informando o local onde ela deve ser colocada.

Numa biblioteca de passagem de mensagens do padrão MPI, as aplicações podem utilizar funções que transmitem mensagens de forma bloqueante ou não-bloqueante. Sendo assim, as informações de controle do protocolo de *checkpointing* que serão enviadas não podem ser capturadas no momento em que a requisição de envio da mensagem é realizada, pois essa abordagem pode gerar inconsistências no caso de um envio não-bloqueante. A Figura 5.2 mostra que o processo p_i requisitou o envio de uma mensagem de forma não-bloqueante no evento e_i^a. Porém, a mensagem só foi realmente enviada no evento e_i^b mais adiante no tempo. Além disso, entre os dois eventos citados, a aplicação gravou o *checkpoint* estável s_i^α. No processo p_j, uma forma bloqueante de recepção foi utilizada pela aplicação, e a mensagem m enviada por p_i foi recebida no evento e_j^c. Neste exemplo da figura, caso as informações de controle do protocolo quase-síncrono enviadas na mensagem m fossem capturadas no evento e_i^a, o *checkpoint* s_j^β não iria depender causalmente do *checkpoint* s_i^α e os dois *checkpoints* poderiam ser usados num mesmo *checkpoint* global consistente. Porém, os *checkpoints* s_i^α e s_j^β não podem fazer parte de um mesmo *checkpoint* global consistente, pois a mensagem m já foi recebida por p_j, mas ainda não foi enviada por p_i. Assim, as informações de controle do protocolo de *checkpointing* quase-síncrono utilizado na execução de uma aplicação MPI só são capturadas e anexadas às mensagens quando estas últimas forem ser realmente transmitidas.

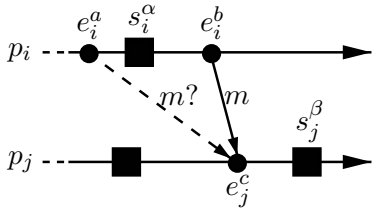


Figura 5.2: Exemplo de inconsistência com envio não-bloqueante de mensagem.

Processamento de Informação de Controle

Durante a execução de uma aplicação MPI utilizando a infra-estrutura para *checkpointing* quase-síncrono, as informações de controle vindas em todas as mensagens recebidas devem ser passadas para o protocolo de *checkpointing* utilizado processá-las, decidindo se um *checkpoint* forçado deve ou não ser gravado.

Assim que um processo recebe um envelope de uma mensagem, o campo do tamanho da informação de controle é verificado, para que o processo saiba se precisa ou não receber alguma informação de controle do protocolo de *checkpointing* quase-síncrono. Em caso positivo, a informação de controle é recebida logo após o envelope, e depois o corpo da mensagem também é recebido. Quando todas as partes da mensagem foram recebidas, uma função do módulo CKPTQS é chamada, para que a ação de processar informação de controle do protocolo quase-síncrono seja executada, podendo induzir a gravação de um *checkpoint* forçado.

Da mesma forma que no caso do envio de mensagens, numa biblioteca do padrão MPI, as aplicações podem requisitar o recebimento de mensagens de forma não-bloqueante, e posteriormente processar o seu conteúdo, quando este estiver disponível. Sendo assim, o processamento das informações de controle vindas na mensagem precisa ser feito pelo protocolo quase-síncrono quando a mensagem é totalmente recebida da rede de comunicação, e não quando a aplicação processa os dados vindos naquela mensagem. A Figura 5.3 mostra um cenário em que o processo p_j faz um envio bloqueante de mensagem para o processo p_i no evento e_j^d. O processo p_i havia requisitado a recepção não-bloqueante dessa mensagem no evento e_i^a, e embora a mensagem tenha sido completamente recebida no evento e_i^b, o processo p_i só consultou a biblioteca MPI para ver se a mensagem tinha sido recebida e utilizou o seu conteúdo no evento e_i^c. Além disso, entre os eventos e_i^b e e_i^c, o processo p_i gravou o *checkpoint* estável s_i^α. Nesta figura, caso o protocolo quase-síncrono processe a informação de controle da mensagem m quando o conteúdo desta é usado por p_i, o *checkpoint* s_i^α não iria depender causalmente do *checkpoint* s_j^β, e os dois poderiam fazer parte de um mesmo *checkpoint* global consistente. Porém, os *checkpoints* s_i^α e s_j^β não podem fazer parte de um mesmo *checkpoint* global consistente, pois a mensagem m já foi recebida por p_i, mas ainda não foi enviada por p_j. Assim, as informações de controle recebidas juntamente com as mensagens da aplicação MPI são sempre processadas pelo protocolo de *checkpointing* quase-síncrono utilizado assim que a mensagem inteira é recebida da rede de comunicação.

5.1.3 Limitações da Infra-Estrutura Implementada

A infra-estrutura para *checkpointing* quase-síncrono implementada no LAM/MPI possui algumas limitações relacionadas a decisões de projeto. Da forma como a infra-estrutura

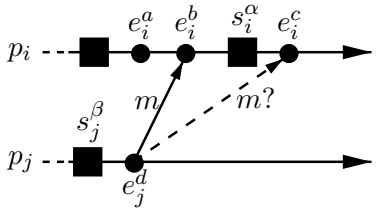


Figura 5.3: Exemplo de inconsistência com recepção não-bloqueante de mensagem.

foi implementada, esta precisa que o BLCR seja utilizado para a gravação dos estados dos processos. Além disso, enquanto o BLCR não é portado para outras plataformas diferentes, a infra-estrutura funciona apenas na plataforma Intel de 32 *bits*.

Embora o LAM/MPI implemente boa parte da versão 2 do padrão MPI [2], a infra-estrutura implementada funciona apenas com as funcionalidades presentes na versão 1 do padrão MPI. Os dois principais tipos de comunicação definidos na versão 1 do padrão MPI são as coletivas e as ponto-a-ponto. As comunicações coletivas possuem algumas implementações diferentes, que exploram características como máquinas com mais de um processador. Porém, a infra-estrutura funciona apenas com o conjunto de comunicações coletivas chamado de `lam_basic`, que é implementado utilizando as primitivas básicas de envio e recpção de mensagens do LAM/MPI.

5.2 Curupira

Como foi apresentado no Capítulo 4 sobre trabalhos relacionados, até onde se conhece, não existe uma biblioteca de passagem de mensagens que implemente recuperação de falhas por retrocesso de estado utilizando *checkpointing* quase-síncrono. Aliado a este fato, resultados recentes na área de algoritmos para coleta de lixo em padrões gerados por protocolos quase-síncronos RDT [42, 43, 44] motivaram o estudo e implementação de uma arquitetura de *software* para recuperação de falhas utilizando *checkpointing* quase-síncrono da classe RDT dentro do LAM/MPI. Esta arquitetura de *software* está restrita à classe RDT, pois esta última apresenta vantagens em relação às demais como: coleta de lixo local em cada processo, com um limite máximo para o número de *checkpoints* mantidos [42, 43, 44]; o retrocesso da computação é menor em caso de falha [6] e o cálculo da linha de recuperação é mais fácil em padrões RDT [48, 49]. Assim, para compor esta arquitetura de *software* chamada de CURUPIRA, foi utilizada a infra-estrutura para *checkpointing* quase-síncrono já implementada no LAM/MPI, juntamente com o algoritmo de coleta de lixo RDT-LGC [42, 43, 44] e um algoritmo síncrono de recuperação por

retrocesso de estado.

Nas próximas seções será mostrado como o CURUPIRA foi modelado, e também como as implementações foram feitas para integrar a infra-estrutura para *checkpointing* quase-síncrono com os algoritmos de coleta de lixo e recuperação no LAM/MPI. Após essa parte, um seção irá comentar as limitações do CURUPIRA e outra irá mostrar comparações utilizando o CURUPIRA com algumas aplicações MPI.

5.2.1 Arquitetura do Curupira

O CURUPIRA foi modelado como sendo a união de quatro módulos que possuem funções específicas, como pode ser visto na Figura 5.4. Estes módulos são: Coordenação, *Checkpointing*, Coleta de Lixo e Recuperação.

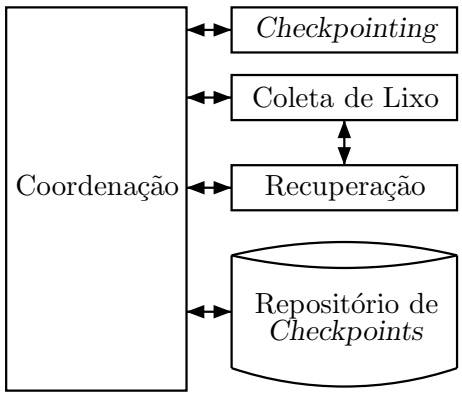


Figura 5.4: Arquitetura do CURUPIRA.

O módulo Coordenação faz a interface entre os demais módulos e o LAM/MPI. A sua implementação foi feita ampliando-se o módulo CKPTQS da infra-estrutura para *checkpointing* quase-síncrono já implementada. Os outros módulos, *Checkpointing*, Coleta de Lixo e Recuperação, implementam as partes principais do CURUPIRA que são, respectivamente, as ações do protocolo RDT que será utilizado na computação distribuída, o algoritmo de coleta de lixo RDT-LGC e o algoritmo síncrono de recuperação por retrocesso de estado.

O Repositório de *Checkpoints* é acessado apenas pelo módulo Coordenação, pois os módulos *Checkpointing* e Coleta de Lixo utilizam funções dele para gravar e remover um *checkpoint* do processo, respectivamente. O módulo Recuperação é apenas executado em uma sessão de recuperação, e utiliza funções do módulo Coleta de Lixo para obter informações sobre os *checkpoints* mantidos em meio estável.

Na implementação do CURUPIRA, todas as opções do CKPTQS que podiam ser passadas para o `mpirun` foram convertidas para utilizarem a chave `-curupira`. O exemplo a seguir mostra um programa com dez processos executado pelo `mpirun` com o CURUPIRA. O protocolo RDT utilizado nesse exemplo é o FDAS, que foi desenvolvido pelo usuário e se encontra no diretório `/protos`. Além disso, a gravação dos *checkpoints* estáveis será realizada no diretório `/gfs`. Nenhuma opção pode ser passada para o algoritmo de coleta de lixo, pois este sempre é executado juntamente com o protocolo RDT escolhido, ou o protocolo padrão que foi definido como sendo o RDT-Minimal [26]. O algoritmo de recuperação pode receber opções do `mpirun`, mas essas serão mostradas em uma seção mais à frente.

```
$ cp fdas.so /protos/
$ mpicc programa.c -o programa
$ mpirun -c 10 -curupira protospath /protos -curupira ckptsdir /gfs \
               -curupira proto FDAS programa
```

5.2.2 Coleta de Lixo com o RDT-LGC

Embora o RDT-LGC possa ser implementado juntamente com o protocolo quase-síncrono RDT [42, 43, 44], decidiu-se por implementá-lo separadamente no módulo Coleta de Lixo, para que qualquer protocolo pudesse ser utilizado. Dessa forma, a implementação do RDT-LGC precisa propagar e manter a sua própria informação de controle, que é o seu vetor de dependências, como está detalhado no Algoritmo 3.2 da Seção 3.3.4.

Para que o RDT-LGC possa enviar o seu vetor de dependências em todas as mensagens, logo após o protocolo de *checkpointing* RDT colocar a sua informação de controle na mensagem, uma função do módulo Coleta de Lixo é chamada para que o vetor de dependências do RDT-LGC também possa ser enviado. Assim, as informações de controle do protocolo RDT e do RDT-LGC são enviadas na mesma parte `cinfo` da mensagem, e o campo do tamanho dessa parte no envelope é atualizado para a soma dos tamanhos das duas informações de controle.

Quando uma mensagem é recebida, as informações de controle da parte `cinfo` são separadas, para que o protocolo RDT e o RDT-LGC possam processá-las. O protocolo RDT processa a sua informação de controle antes do RDT-LGC, pois um *checkpoint* forçado pode ser induzido, se tornando o último *checkpoint* estável daquele processo e, portanto, podendo ser usado nas operações do RDT-LGC. Ao processar a sua informação de controle, o RDT-LGC verifica se alguma dependência causal nova foi detectada, e em caso positivo, liga o seu último *checkpoint* estável ao processo de que foi percebida a nova dependência.

Além de processar a sua informação de controle recebida nas mensagens da aplicação, o módulo de Coleta de Lixo é acionado quando um novo *checkpoint* do processo corrente é gravado em meio estável. Quando o protocolo RDT induz um *checkpoint* forçado ou grava um *checkpoint* básico, a função do módulo Coordenação `cp_save_ckpt()` (antiga `ckptqs_save_ckpt()` do módulo CKPTQS) é chamada para usar o BLCR e realizar a gravação. Depois que esta função realmente grava o *checkpoint* em meio estável, ela chama a função do módulo Coleta de Lixo que implementa as operações do RDT-LGC que devem ser executadas assim que um novo *checkpoint* do processo é gravado. Eventualmente, por ocasião da gravação de um novo *checkpoint* básico ou pela indução de um forçado, quando o RDT-LGC é executado, algum *checkpoint* pode ser tornar obsoleto e precisar ser removido do meio estável. Para efetuar a operação de remoção, o módulo Coleta de Lixo executa a função `cp_delete_ckpt()` do módulo Coordenação, que se encarrega de remover o *checkpoint* com o índice fornecido do repositório de *checkpoints* no meio estável.

5.2.3 Recuperação por Retrocesso de Estado

No LAM/MPI sem nenhuma modificação, quando um falha ocorre com um dos processos da aplicação, o `lamd` daquela máquina reporta o ocorrido ao `mpirun`, que por sua vez encerra toda a computação.

No CURUPIRA, o `mpirun` foi modificado para em caso de falha notificar os demais processos de que houve uma falha e uma sessão de recuperação será iniciada. Cada processo da aplicação mantém no seu módulo Recuperação, uma variável booleana chamada `cp_failed`, para marcar se aquele processo falhou na sua execução ou não. Durante uma execução sem falhas, todas as variáveis `cp_failed` de todos os processos estão com o valor verdadeiro, indicando que eles falharam, de modo que quando um processo realmente falha e retorna após o retrocesso para algum *checkpoint* gravado, o conjunto de processos falhos pode ser identificado. Dessa forma, ao serem notificados pelo `mpirun` do início de uma sessão de recuperação, cada processo que não falhou muda o valor de `cp_failed` para falso, grava um *checkpoint* estável do seu estado corrente, que é o seu *checkpoint* volátil, e depois encerra a sua execução. Depois que todos os processos terminaram a sua execução, o `mpirun` também termina.

A justificativa para que os processos que não falharam tornem estáveis os seus *checkpoints* voláteis é que estes podem ser usados na linha de recuperação para o grupo de processos falhos, minimizando o custo de retrocesso. Caso algum processo não consiga mudar o valor de `cp_failed` para falso e gravar o *checkpoint* estável após a notificação do `mpirun`, esse processo irá contar como um dos processos falhos, e a recuperação irá funcionar normalmente. A única desvantagem é que a linha de recuperação poderá não ser tão recente quanto possível.

Depois que toda a computação termina, é preciso uma intervenção externa para que a sessão de recuperação realmente comece. Um novo `mpirun` deve ser executado com uma opção para o CURUPIRA requisitando o início de uma sessão de recuperação e passando o índice do último *checkpoint* estável mantido por cada um dos processos. Estes índices são passados para o `mpirun` separados por vírgula, e ordenados do processo 0 até o $n-1$. Além desses índices, o CURUPIRA precisa receber por meio do `mpirun` um arquivo com o esquema de como a aplicação foi executada nas máquinas. O `mpirun` foi modificado para gerar esse arquivo quando a aplicação é inicialmente executada com o CURUPIRA, podendo, inclusive, receber o nome desse arquivo por meio de uma opção passada na linha de comando. O exemplo a seguir mostra a execução de uma aplicação que foi interrompida por uma falha, e a posterior requisição de uma sessão de recuperação, informando os índices dos últimos *checkpoints* estáveis dos cinco processos da aplicação e o arquivo de esquema.

```
$ mpirun -c 5 -curupira schema esquema -curupira proto NRAS programa
$ #
$ # Falha!
$ #
$ mpirun -curupira recovery 5,3,7,2,0 esquema
```

Depois que uma requisiação de recuperação foi feita ao CURUPIRA por meio do `mpirun`, os `lamds` das máquinas são instruídos a reiniciarem os processos da aplicação a partir dos *checkpoints* de índices especificados. Assim que os processos são reiniciados pelos `lamds`, eles retomam a sua execução logo após a chamada da função `rc_checkpoint()` na *thread* do BLCR, enquanto a *thread* da aplicação está bloqueada na função `cp_save_ckpt()`. Quando a função `rc_checkpoint()` retorna, ela indica que uma recuperação foi iniciada, e então a função do módulo Recuperação que trata de uma sessão de recuperação é chamada. Esta função recebe do `mpirun` a etapa que deve ser executada da recuperação, que pode ser: calcular a linha de recuperação ou acertar as estruturas de dados do CURUPIRA e retornar à execução normal.

Cada uma das duas etapas de uma sessão de recuperação foram implementadas como um algoritmo distribuído de validação atômica semelhante ao *two-phase commit* [29, pp. 562-572]. A mensagem enviada pelo `mpirun` com a etapa a ser executada pelos processos já faz parte do algoritmo de validação atômica. Após receber essa mensagem, os processos irão realizar as tarefas correspondentes àquela etapa da recuperação e retornar informações para o `mpirun` ou uma confirmação de que as tarefas foram executadas. Ao receber as respostas dos processos, dependento da etapa da recuperação, o `mpirun` fará com que os processos encerrem a sua execução ou passem a executar a aplicação normalmente, pois

o retrocesso já foi concluído. Toda a recuperação foi implementada dessa maneira, para que o `mpirun` consiga garantir que todos os processos participam do cálculo da linha de recuperação, e que todos retrocedem para a linha de recuperação calculada e retomam a sua execução normal simultaneamente.

O Algoritmo 5.1 mostra os passos executados pelo `mpirun` e por um processo p_i para realizar a etapa do cálculo da linha de recuperação, após a mensagem inicial ser enviada pelo `mpirun` e recebida pelos processos. Cada processo p_i envia uma mensagem para o *mpirun* contendo o seu `cp_failed` e um conjunto de tuplas contendo o índice e o vetor de dependências de todos os seus *checkpoints* estáveis mantidos. Para que o vetor de dependências de cada *checkpoint* estável gravado pudesse ser facilmente recuperado, o *BCC* mostrado no Algoritmo 3.1, que é utilizado pelo RDT-LGC, foi acrescido de um campo para guardar o vetor de dependências do *checkpoint* correspondente. Dessa forma, o RDT-LGC garante que apenas os vetores de dependência dos *checkpoints* não-obsoletos são mantidos na memória e são facilmente acessíveis por meio dos *BCC*s. Após enviar a resposta para o `mpirun`, cada processo p_i espera a confirmação do `mpirun` e encerra a sua execução. Por outro lado, no Algoritmo 5.1, o processo `mpirun` acumula os índices e vetores de dependências de todos os *checkpoints* estáveis de todos os processos, e também cria o conjunto I_F com tuplas representando os últimos *checkpoints* estáveis dos processos falhos, de acordo com os valores das variáveis `cp_failed` recebidas (linhas 3-8). A seguir, as linhas de recuperação para cada um dos processos falhos são calculadas utilizando-se o algoritmo de Wang [48, 49] para padrões RDT descrito na Seção 3.2.2 (linhas 9-14). Por fim, a linha de recuperação para todos os processos falhos R_F é calculada por meio da intersecção entre as linhas de recuperação para falhas individuais, que pode ser feita da mesma forma que a intersecção de dois *checkpoints* globais, como foi descrito na Seção 2.4 (linhas 15-17).

Depois de calculada a linha de recuperação, o `mpirun` manda uma mensagem para os processos confirmando que tudo já foi calculado, e estes encerram a sua execução. Na seqüência, o `mpirun` começa a segunda etapa da recuperação, que consiste no retrocesso da aplicação para a linha de recuperação calculada, acerto das estruturas de dados do CURUPIRA e o prosseguimento normal da computação distribuída. Para tanto, o `mpirun` executa uma nova cópia dele mesmo com a chamada de sistema operacional `exec()`, passando parâmetros para uma opção do CURUPIRA de maneira semelhante à requisição de recuperação mostrada anteriormente. Os índices dos *checkpoints* de cada processo, que fazem parte da linha de recuperação são passados para a nova cópia do `mpirun` e o esquema de execução da aplicação também. Caso a linha de recuperação possa ser calculada externamente, o usuário pode requisitar o retrocesso para a linha de recuperação manualmente, como no exemplo a seguir, que é a continuação da sessão de recuperação do exemplo anterior. Este exemplo também ilustra como o `mpirun` chama a si mesmo com o `exec()`.

Algoritmo 5.1 Cálculo da linha de recuperação.

No processo `mpirun`

```
 1: Var
 2:    R_F: array[0...n-1] of inteiro
 3: I_F ← ∅                                   {recebe informação dos processos e gera I_F}
 4: for j ← 0 to n-1 do
 5:    recebe mensagem (cp_failed, CKPTS) do processo p_j
 6:    TODOS_CKPTS[j] ← CKPTS
 7:    if cp_failed then
 8:       I_F ← I_F ∪ {(j, max(α)) | ∃(k, DV) ∈ TODOS_CKPTS[j], k = α}
 9: LINHAS_RECUP ← ∅                          {calcula linha de recuperação para cada processo falho}
10: for all (f, α) ∈ I_F do
11:    for j ← 0 to n-1 do
12:       encontra (max(ind), DV) ∈ TODOS_CKPTS[j]  |  (DV[f] ≤ α)
13:       LINHA[j] ← ind
14:    LINHAS_RECUP ← LINHAS_RECUP ∪ { LINHA }
15: for j ← 0 to n-1 do                       {calcula linha de recuperação para todos os processos falhos}
16:    encontra min(ind)  |  (∃ LINHA ∈ LINHAS_RECUP , LINHA[j] = ind)
17:    R_F[j] ← ind
```

Em cada processo p_i

```
1: CKPTS ← ∅
2: for all BCC do
3:    CKPTS ← CKPTS ∪ {(BCC.IND, BCC.DV)}
4: envia mensagem (cp_failed, CKPTS) para mpirun
```

```
$ mpirun -curupira rollback 3,3,4,1,0 esquema
```

Na segunda etapa da recuperação, da mesma forma que na primeira, quando os processos são reiniciados, a função que trata da sessão de recuperação no módulo Recuperação vai ser chamada, e vai esperar que uma mensagem do `mpirun` seja recebida, para saber como proceder. O `mpirun` envia uma mensagem indicando que a segunda etapa da recuperação deve ser executada. Ao receber a mensagem do `mpirun`, cada processo p_i realiza os seguintes passos: muda o valor de `cp_failed` para verdadeiro; elimina todos os *checkpoints* do meio estável com índice maior que a entrada i do vetor de dependências do RDT-LGC; reinicia todas as requisições de envio e recepção de mensagens que estão em progresso dentro da biblioteca MPI e, por fim, envia uma mensagem de confirmação para o `mpirun`. Este último, ao receber as confirmações de todos os processos, envia uma última mensagem a todos eles, indicando que a sessão de recuperação terminou e a execução normal da aplicação pode ser retomada.

Para que a aplicação volte a executar normalmente após uma sessão de recuperação, ainda na *thread* do BLCR, o *mutex* global do LAM/MPI é liberado, desbloqueando a

thread da aplicação, que estava bloqueada na função `cp_save_ckpt()`. Como a *thread* da aplicação volta a executar a partir do ponto onde esta foi bloqueada na ocasião de gravação do *checkpoint*, todos os módulos do CURUPIRA e seus algoritmos continuam funcionando corretamente e como se nenhuma falha tivesse ocorrido.

Da forma como foi implementado, o CURUPIRA tolera falhas nos processos da aplicação MPI, que falham e param de executar. Porém, falhas podem ocorrer no `mpirun` e nos `lamds`. No caso de um falha ocorrer em um dos `lamds`, o LAM/MPI já possui suporte para que outro `lamd` seja executado sem interferir na execução da aplicação. Em compensação, uma falha no `mpirun` não é tratada nem pelo LAM/MPI e nem pelo CURUPIRA. Uma maneira de tratar as falhas no `mpirun` com o CURUPIRA seria encerrar toda a aplicação na ocorrência da falha, e iniciar uma sessão de recuperação logo em seguida. Como todos os processos foram encerrados com o valor de `cp_failed` verdadeiro, a linha de recuperação será calculada como se tivessem ocorrido falhas em todos eles. Sendo assim, tudo deve funcionar corretamente na recuperação por retrocesso de estado da aplicação quando uma falha ocorre no `mpirun`.

5.2.4 Limitações do Curupira

A implementação do CURUPIRA no LAM/MPI possui algumas limitações. Uma delas está relacionada ao uso do BLCR para gravar os estados dos processos, pois ele não é capaz de gravar o estado dos *sockets* de um processo. Porém, é factível imaginar que uma aplicação distribuída que utiliza o MPI para realizar a comunicação entre os processos não vai usar *sockets* diretamente. Além disso, no CURUPIRA, os *sockets* que são usados para a comunicação entres os processos são reabertos durante a sessão de recuperação. Assim, para a aplicação o fechamento e a reabertura das conexões fica totalmente transparente.

O CURUPIRA implementa o retrocesso de estado da aplicação em caso de falha, para o *checkpoint* global consistente o mais recente possível, chamado de linha de recuperação. Sendo assim, os canais são considerados não-confiáveis e as mensagens em trânsito na linha de recuperação não são recuperadas. Dessa forma, as aplicações podem perder mensagens e precisam ser projetadas para lidar com isso. Uma das maneiras de contornar a perda de mensagens é implementar um protocolo de janela deslizante semelhante ao que é usado no TCP. Outra maneira seria implementar algum tipo de *message logging* na própria aplicação, de modo que as mensagens pudessem ser reenviadas durante a sessão de recuperação. Esta última abordagem iria exigir uma pequena mudança no módulo Recuperação do CURUPIRA, para que a aplicação fosse avisada da sessão de recuperação, e pudesse proceder com o reenvio das mensagens necessárias.

O padrão MPI não define nenhum tipo de mecanismo para tolerância a falhas, e assim, a maioria das implementações não são feitas com essa preocupação. Dessa forma,

a integração de mecanismos para prover tolerância a falhas depois de criada a biblioteca MPI se torna realmente difícil, ainda mais um mecanismo baseado em *checkpointing* quase-síncrono. Deste fato, pode-se apontar outra limitação do CURUPIRA, que não suporta aplicações que utilizem comunicações coletivas, pois elas teriam que ser todas reimplementadas, e as modificações no código do LAM/MPI seriam realmente intrusivas.

5.2.5 Comparações e Resultados

Além de realizar a recuperação de falhas, uma arquitetura para recuperação de falhas que utilize *checkpointing* quase-síncrono deve tentar introduzir o mínimo de degradação no desempenho da aplicação durante uma execução sem falhas. Assim, algumas comparações utilizando aplicações reais foram feitas, de modo a tentar estimar a degradação que um protocolo RDT executando no CURUPIRA proporciona no desempenho de uma aplicação MPI. Além disso, a implementação do protocolo de *checkpointing* síncrono presente no LAM/MPI foi modificada para que os processos da aplicação pudessem requisitar a gravação do *snapshot* consistente. Isto foi feito para que a abordagem síncrona já existente no LAM/MPI pudesse ser devidamente comparada com a quase-síncrona do CURUPIRA.

Além das comparações de desempenho, como o protocolo RDT-Minimal [26] e o algoritmo de coleta de lixo RDT-LGC [42, 43, 44] são resultados muito recentes na área de *checkpointing* quase-síncrono, comparações foram feitas utilizando-se o CURUPIRA, de modo que algumas características desses novos algoritmos pudessem ser observadas, como a quantidade máxima de *checkpoints* mantidos pelo RDT-LGC em um processo durante a execução da aplicação e a quantidade de *checkpoints* forçados induzidos pelo RDT-Minimal.

Para realizar as comparações, três aplicações MPI foram escolhidas. Estas aplicações têm os nomes **Anel**, **Fractal** e **Fecho** e todos os tempos medidos nestas comparações são resultados de uma média de cinco execuções da aplicação em questão. Além disso, cada aplicação foi executada utilizando dois, quatro, oito, dezesses e trinta e dois processos, de modo que a escalabilidade das abordagens para *checkpointing* e seus respectivos algoritmos pudessem ser testados. A Tabela 5.1 contém o tempo de execução em segundos e o número de mensagens enviadas pelas aplicações ao executarem no LAM/MPI sem que nenhuma abordagem para *checkpointing* fosse utilizada. Esta tabela servirá de base para que as abordagens para *checkpointing* possam ser comparadas quanto à degradação de desempenho e outros aspectos.

Antes que as comparações e resultados sejam mostrados, é interessante que as aplicações utilizadas sejam descritas, para que os resultados possam ser melhor interpretados.

		Número de Processos				
		2	4	8	16	32
Anel	Tempo	0,62	0,72	0,78	1,98	2,87
	# Msgs	204	410	822	1646	3294
Fractal	Tempo	6,66	2,93	1,92	1,61	1,63
	# Msgs	2032	2040	2056	2088	2152
Fecho	Tempo	184,86	82,44	42,76	35,70	51,21
	# Msgs	24	96	336	1710	6510

Tabela 5.1: Tabela base para as comparações.

Anel Esta aplicação apenas circula cem vezes uma mensagem em um anel lógico composto por todos os processos da computação. Como percebe-se pelos valores na Tabela 5.1, aumentando-se o número de processos, os tempos de execução e o número de mensagens enviadas também aumentam. Além destas características, como cada processo da aplicação fica bloqueado esperando receber a mensagem, para então poder repassá-la, qualquer atraso na execução de um processo pode degradar o desempenho de toda a aplicação. Assim, a degradação no desempenho acarretada pelas abordagens para *checkpointing* implementadas podem ser melhor observadas durante a execução dessa aplicação.

Fractal Neste programa, um fractal é construído utilizando-se os processos disponíveis. Um processo mestre distribui pedaços do fractal para serem calculados pelos demais processos, denominados escravos. Depois que os escravos terminam os seus cálculos, eles enviam o resultado ao processo mestre. Este último continua a distribuir trabalho para os processos escravos até que o fractal inteiro esteja todo calculado. Como a quantidade de trabalho é sempre constante, um aumento no número de processos permite melhorar o tempo de execução de toda a aplicação, como observa-se na Tabela 5.1. A explicação está no fato de que mais pedaços do fractal podem ser calculados ao mesmo tempo. Porém, existe um limite no aumento do número de processos que é benéfico, como também pode-se perceber naquela tabela, onde o tempo de execução dessa aplicação com trinta e dois processos foi praticamente o mesmo que com dezesseis. Por outro lado, a quantidade de mensagens enviadas é praticamente a mesma independente do número de processos, já que é proporcional à quantidade de trabalho total, que é constante.

Fecho O fecho transitivo de um grafo denso, orientado e com mil vértices é calculado nesta aplicação [8]. A entrada é o grafo representado pela sua matriz de adjacências $A_{m,m}$. Duas submatrizes $B_{m/p,m}$ e $C_{m,m/p}$ daquela matriz são enviadas para todos os p processos da computação distribuída. Em cada rodada, todos os processos calculam o

fecho transitivo do seu subgrafo e mandam uma mensagem para todos os outros processos da aplicação com o resultado. Após $O(log\ p)$ rodadas, o programa obtém o fecho transitivo de todo o grafo. Como pode-se observar na Tabela 5.1, o aumento no número de processos diminui o tempo de execução da aplicação, pois mais trabalho é realizado em paralelo. Porém, como o número de mensagens aumenta muito também, existe um limite no número de processos que beneficia a aplicação, pois após esse limite o tempo gasto com a transmissão de mensagens passa a ser considerável. Pode-se perceber esse fato naquela tabela, onde o tempo de execução dessa aplicação com trinta e dois processos foi superior à com dezesseis.

Comparação entre Protocolos de *Checkpointing* Quase-Síncronos

A classe de protocolos quase-síncronos RDT é conhecida por induzir um número muito alto de *checkpoints* forçados [47]. Para tentar diminuir ao máximo essa quantidade, o RDT-Minimal apenas induz um *checkpoint* forçado quando uma dependência não rastreável em tempo de execução pode se formar [26]. Assim, uma comparação entre o RDT-Minimal e o FDAS foi feita com as aplicações já mencionadas.

A Tabela 5.2 mostra para cada uma das aplicações, o tempo de execução em segundos e o número de *checkpoints* forçados que foram induzidos pelo FDAS [49] e pelo RDT-Minimal. Para obter esses dados, nenhuma chamada à função `MPI_Checkpoint()` foi introduzida nas aplicações. Porém, antes de iniciar a aplicação propriamente dita, todos os protocolos de *checkpointing* quase-síncronos gravam um *checkpoint* de cada processo, e isto permite que a linha de recuperação progrida, mesmo em uma aplicação que não seleciona *checkpoints* básicos para serem gravados.

Percebe-se pelos dados da Tabela 5.2, que tanto o FDAS quanto o RDT-Minimal degradam muito a execução da aplicação **Anel**, o que é ainda mais perceptivo quando o número de processos aumenta. Este fato ocorre porque na aplicação **Anel**, o progresso da execução de um processo está muito fortemente ligado à execução de um outro, daquele que se espera receber uma mensagem. Assim, mesmo a operação de gravação de um *checkpoint* local interfere no tempo total de execução da aplicação. E este atraso no progresso da aplicação se torna ainda pior quando o número de processos aumenta, pois a própria aplicação **Anel** possui a sua complexidade de tempo proporcional ao número de processos. Embora o RDT-Minimal sempre induza um número de *checkpoints* forçados menor ou igual ao número induzido pelo FDAS, no caso da aplicação **Anel**, os dois protocolos RDT induziram praticamente a mesma quantidade de *checkpoints* forçados. Comparando-se o número de *checkpoints* forçados dos dois protocolos com o número de mensagens enviadas pela aplicação, percebe-se que praticamente todas as mensagens causaram a gravação de um *checkpoint* forçado. Isto é esperado, pois a aplicação **Anel** possui um padrão de comunicação muito uniforme, e caso uma mensagem induza um

checkpoint forçado, todas as demais também induzirão. Nem o RDT-Minimal consegue induzir menos *checkpoints* forçados nesse caso, pois sempre que uma mensagem chega em um processo, ela traz informações causais novais e um caminho-PMM cuja duplicação causal não é visível é detectado, provocando a gravação do *checkpoint* forçado.

Assim como a aplicação **Anel**, a aplicação **Fractal** também sofreu uma degradação muito grande no desempenho por ambos os protocolos RDT, como indica a Tabela 5.2. Porém, dessa vez o RDT-Minimal induziu menos *checkpoints* forçados, e a sua degradação foi menor que a do FDAS. É possível observar também, que o custo associado à utilização do FDAS fez com que a aplicação **Fractal** tivesse o menor tempo de execução com quatro processos. Quando nenhum protocolo de *checkpointing* era utilizado, o número ótimo de processos para essa aplicação era dezesseis. O RDT-Minimal possui melhores tempos de execução nessa aplicação, mas conforme o número de processos aumenta, o tempo de execução também aumenta, ao invés de diminuir, não sendo interessante executar a aplicação **Fractal** juntamente com o RDT-Minimal utilizando mais do que dois processos.

Comparando-se os resultados obtidos com o protocolo FDAS e com o protocolo RDT-Minimal na execução da aplicação **Fecho** mostrados na Tabela 5.2, percebe-se que ambos ficaram bem próximos do tempo de execução sem nenhum protocolo de *checkpointing*. Neste caso, o RDT-Minimal também degradou menos o desempenho da aplicação que o FDAS, induzindo menos *checkpoints* forçados. Além disso, ambos os protocolos RDT mostraram bons resultados quando o número de processos foi aumentado.

			Número de Processos				
			2	4	8	16	32
FDAS	**Anel**	Tempo	31,53	62,29	126,28	253,95	513,17
		# Ckpts Forçados	202	404	811	1623	3247
	Fractal	Tempo	214,15	105,64	106,33	110,35	119,41
		# Ckpts Forçados	1354	1354	1354	1354	1354
	Fecho	Tempo	196,35	94,43	58,11	52,38	58,07
		# Ckpts Forçados	14	34	80	213	433
RDT-Minimal	**Anel**	Tempo	0,90	62,36	124,79	252,26	512,71
		# Ckpts Forçados	0	403	807	1615	3231
	Fractal	Tempo	7,94	20,68	42,96	52,36	59,94
		# Ckpts Forçados	0	200	515	629	660
	Fecho	Tempo	184,49	83,84	57,44	52,15	54,03
		# Ckpts Forçados	0	1	61	171	354

Tabela 5.2: Execuções das aplicações sem seleção de *checkpoints* básicos.

Para que o FDAS e o RDT-Minimal também pudessem ser comparados quando a aplicação seleciona *checkpoints* básicos, as três aplicações foram estudadas e foram colo-

cadas chamadas para a função `MPI_Checkpoint()` em algumas posições do código. Essas localizações no código foram escolhidas na tentativa de não deixar que a gravação de um *checkpoint* básico em um processo atrasasse a execução de outro. Além disso, os locais escolhidos também levaram em conta algum progresso da computação naquele processo local que fosse interessante gravar em meio estável. Na Tabela 5.3 estão os números de *checkpoints* básicos que foram gravados pelas aplicações, quando estas foram executadas juntamente com o FDAS ou o RDT-Minimal no CURUPIRA.

	Número de Processos				
	2	4	8	16	32
Anel	201	403	807	1615	3231
Fractal	28	28	28	28	27
Fecho	8	16	32	80	160

Tabela 5.3: Quantidade de *checkpoints* básicos.

Na Tabela 5.4 estão o tempo de execução em segundos e o número de *checkpoints* forçados das aplicações que foram modificadas para selecionar *checkpoints* básicos. Uma característica que se percebe nessa tabela é que os tempos de execução da aplicação **Anel** foram drasticamente reduzidos. Isto foi possível selecionando-se um *checkpoint* básico logo após a mensagem ser enviada para outro processo. Desse modo, parte do tempo em que os demais processos estavam recebendo e repassando a mensagem, o processo local estava gravando o seu *checkpoint*. Como conseqüência direta da criação de um novo *checkpoint*, um novo intervalo entre *checkpoints* se inicia, e quando a mensagem retorna ao processo, ela já não induz mais um *checkpoint* forçado. Assim, tanto o FDAS quanto o RDT-Minimal induziram poucos *checkpoints* forçados na execução da aplicação **Anel**. Com a aplicação **Fractal**, os tempos de execução foram minimamente melhorados utilizando-se o FDAS, mas pioraram um pouco quando o RDT-Minimal foi utilizado. O mesmo aconteceu com o número de *checkpoints* forçados, que diminuiu um pouco no FDAS, mas aumentou no RDT-Minimal. Por fim, na aplicação **Fecho** houve pouca variação no tempo de execução, pois os *checkpoints* forçados que não foram mais induzidos acabaram sendo compensados pelos *checkpoints* básicos que foram gravados.

Embora no caso da aplicação **Anel**, a seleção de *checkpoints* básicos tenha melhorado os tempos de execução, percebe-se que para padrões de comunicação mais complicados e com muitas mensagens, a influência dos *checkpoints* básicos não foi consideravelmente benéfica ou prejudicial. Assim, percebe-se que mesmo que o *checkpoint* básico ou forçado seja gravado apenas localmente, a paralisação na execução daquele processo degrada o desempenho da computação como um todo [9]. Assim, uma maneira de resolver esse problema seria implementar um mecanismo que não paralisasse o processo quando o seu

			Número de Processos				
			2	4	8	16	32
FDAS	**Anel**	Tempo	16,41	16,40	16,83	17,26	21,21
		# Ckpts Forçados	2	4	8	16	32
	Fractal	Tempo	211,35	105,74	106,79	107,64	118,00
		# Ckpts Forçados	1327	1327	1327	1327	1328
	Fecho	Tempo	199,46	97,29	60,03	58,68	54,82
		# Ckpts Forçados	15	33	73	198	402
RDT-Minimal	**Anel**	Tempo	16,12	16,25	16,45	16,66	20,37
		# Ckpts Forçados	0	1	1	1	1
	Fractal	Tempo	21,68	34,45	56,07	57,49	63,89
		# Ckpts Forçados	26	343	646	661	662
	Fecho	Tempo	189,36	93,05	57,06	53,91	52,81
		# Ckpts Forçados	0	17	46	141	294

Tabela 5.4: Execuções das aplicações com seleção de *checkpoints* básicos.

estado fosse gravado em meio estável. Uma opção seria utilizar a chamada de sistema `fork()` para criar uma cópia do processo na memória e gravar o estado dessa cópia no meio estável, enquanto o processo real continua a sua execução [9].

Número de *Checkpoints* Mantidos pelo RDT-LGC

Embora o RDT-LGC estabeleça um limite máximo para o número de *checkpoints* que precisam ser mantidos armazenados no meio estável, que são n *checkpoints* por processo, como este algoritmo nunca foi implementado, algum resultado prático seria interessante. Assim, foram obtidos das execuções apresentadas na Tabela 5.2 e na Tabela 5.4, o número máximo de *checkpoints* mantidos armazenados por algum processo da aplicação. Estes dados estão relacionadas na Tabela 5.5 e na Tabela 5.6, que se referem, respectivamente, às outras tabelas citadas. Percebe-se que não há muita variação entre uma tabela e outra, mas uma característica interessante é que apenas na aplicação **Fractal** que o limite máximo definido pelo RDT-LGC é atingido. E explica-se isto, pelo fato de que essa aplicação é do tipo mestre-escravo, onde o processo mestre é o único que se comunica com todos os processos da aplicação, pois os escravos nunca se comunicam entre si. Assim, o processo mestre precisa manter armazenado no meio estável, um *checkpoint* distinto para a falha individual de cada um dos processos da aplicação, incluindo ele mesmo. Aparentemente, a linha de recuperação progride rapidamente em aplicações que enviam muitas mensagens e os processos não possuem restrição de se comunicarem, fazendo com que poucos *checkpoints* precisem ser mantidos em meio estável.

		Número de Processos				
		2	4	8	16	32
FDAS	**Anel**	2	2	3	3	3
	Fractal	2	4	8	16	32
	Fecho	2	3	4	4	4
RDT-Minimal	**Anel**	1	2	2	2	2
	Fractal	1	4	8	16	32
	Fecho	1	1	4	4	4

Tabela 5.5: Número de *checkpoints* mantidos quando não há seleção de *checkpoints* básicos.

		Número de Processos				
		2	4	8	16	32
FDAS	**Anel**	2	2	3	3	3
	Fractal	2	4	8	16	32
	Fecho	2	2	4	4	4
RDT-Minimal	**Anel**	2	2	2	2	2
	Fractal	2	4	8	16	32
	Fecho	2	2	3	3	3

Tabela 5.6: Número de *checkpoints* mantidos quando há seleção de *checkpoints* básicos.

Comparação entre *Checkpointing* Quase-Síncrono e Síncrono

Para que uma comparação pudesse ser realizada entre a implementação do protocolo de *checkpointing* síncrono existente no LAM/MPI com a abordagem quase-síncrona RDT do CURUPIRA, o LAM/MPI foi modificado para que as aplicações conseguissem requisitar a gravação do *snapshot* consistente. Assim, a função `MPI_Checkpoint()` foi introduzida no LAM/MPI e ao ser executada por qualquer processo da aplicação, ela faz com que o protocolo de *checkpointing* síncrono do LAM/MPI seja ativado.

Assim, as três aplicações que foram modificadas para incluir as chamadas para a função `MPI_Checkpoint()` foram compiladas e rodadas no LAM/MPI que permite a ativação do protocolo de *checkpointing* síncrono pelos processos. No CURUPIRA, a função `MPI_Checkpoint()` grava um *checkpoint* básico local daquele processo, ao passo que no LAM/MPI com a modificação comentada, quando essa função é executada, um *snapshot* consistente de toda a aplicação é gravado. A Tabela 5.7 possui os tempos de execução das aplicações no CURUPIRA com o FDAS e o RDT-Minimal, como já foi mostrado na Tabela 5.4, e também o tempo de execução das aplicações quando estas foram executadas na

versão do LAM/MPI modificada com a inclusão da função `MPI_Checkpoint()`. Pode-se perceber pelos dados da Tabela 5.7, que a sincronização global dos processos no protocolo síncrono degradou mais o desempenho das aplicações **Anel** e **Fecho** do que os protocolos RDT, tanto o FDAS quanto o RDT-Minimal. Com relação à aplicação **Fecho**, o protocolo síncrono do LAM/MPI não escalou bem para um número maior que oito processos, enquanto o RDT-Minimal conseguiu atingir o menor tempo de execução da aplicação com trinta e dois processos. No caso da aplicação **Fractal**, o protocolo síncrono do LAM/MPI foi melhor que o RDT-Minimal por uma pequena margem. De modo geral, percebe-se que a sincronização dos processos no protocolo síncrono bloqueante do LAM/MPI penaliza muito a execução da aplicação. Além disso, os tempos de execução utilizando os protocolos quase-síncronos RDT no CURUPIRA também podem ser melhorados com a gravação não-bloqueante dos estados dos processos, utilizando a chamada de sistema `fork()`, como já foi comentado.

		Número de Processos				
		2	4	8	16	32
FDAS	**Anel**	16,41	16,40	16,83	17,26	21,21
	Fractal	211,35	105,74	106,79	107,64	118,00
	Fecho	199,46	97,29	60,03	58,68	54,82
RDT-Minimal	**Anel**	16,12	16,25	16,45	16,66	20,37
	Fractal	21,68	34,45	56,07	57,49	63,89
	Fecho	189,36	93,05	57,06	53,91	52,81
Síncrono	**Anel**	38,99	80,16	114,02	146,88	210,79
	Fractal	22,28	17,32	18,13	27,41	45,50
	Fecho	195,16	100,28	64,82	84,87	127,16

Tabela 5.7: Execuções das aplicações com *checkpointing* síncrono e quase-síncrono RDT.

Usando-se os dados da Tabelas 5.1 e 5.7 foram feitos gráficos dos tempos de execução das aplicações **Anel**, **Fractal** e **Fecho** pelo número de processos utilizados. As Figuras 5.5, 5.6 e 5.7 mostram, respectivamente, estes gráficos. Percebe-se que tanto os protocolos RDT quanto o protocolo síncrono bloqueante do LAM/MPI penalizam muito o desempenho das aplicações. Além disso, observa-se tendências completamente diferentes nos gráficos das aplicações, sugerindo que o padrão de comunicação tem uma influência considerável no ônus que um protocolo de *checkpointing* pode causar na execução de uma aplicação distribuída.

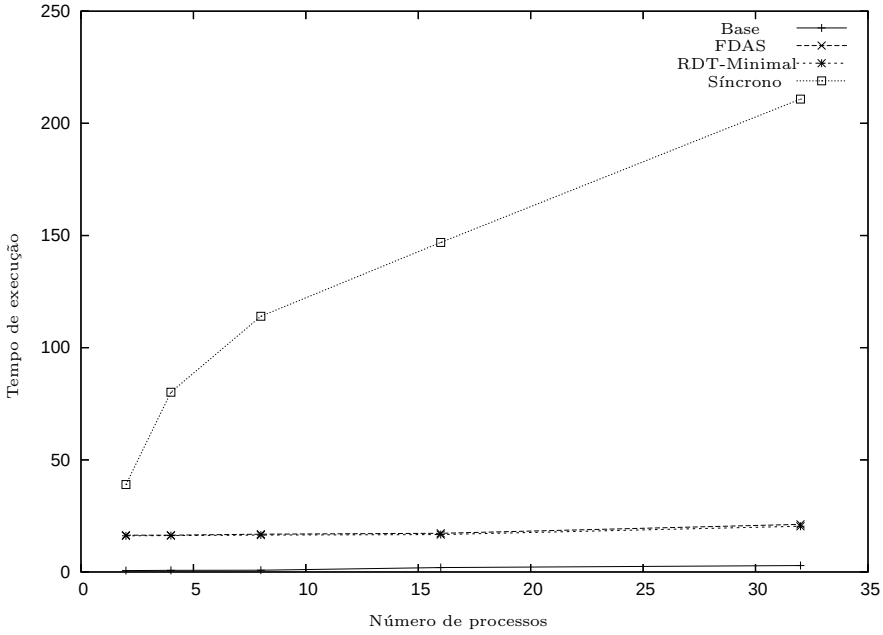


Figura 5.5: Gráfico da aplicação **Anel**.

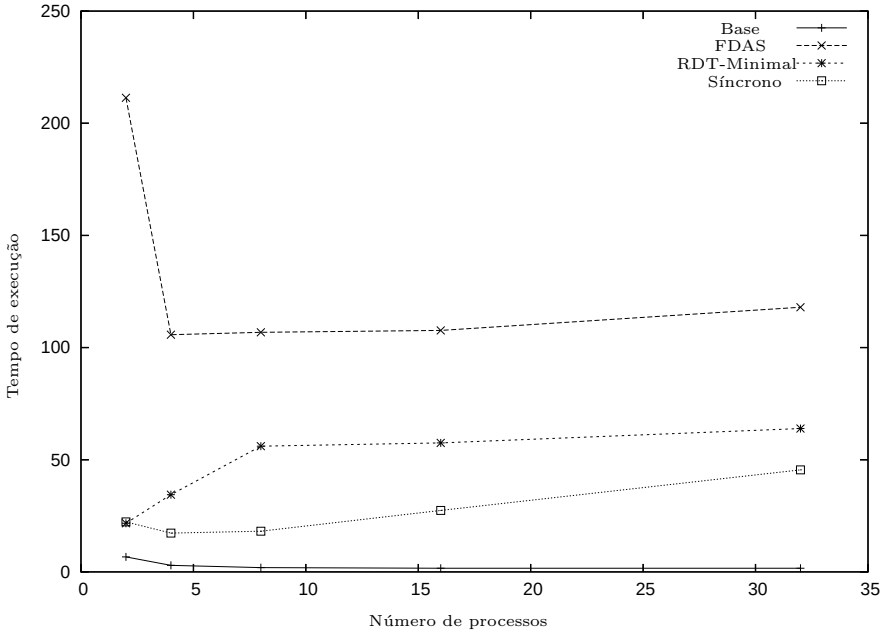


Figura 5.6: Gráfico da aplicação **Fractal**.

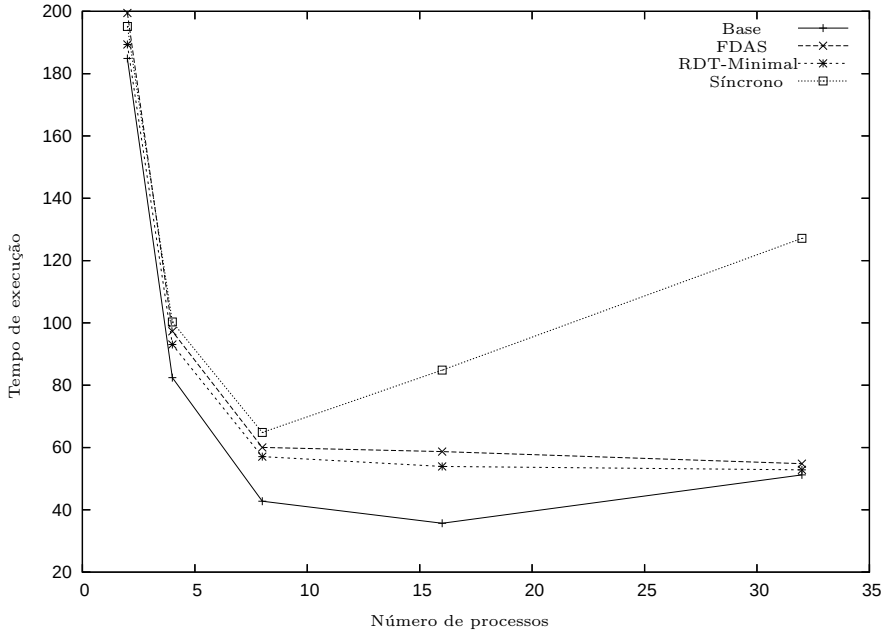


Figura 5.7: Gráfico da aplicação **Fecho**.

5.3 Sumário

Neste capítulo, o modelo de uma infra-estrutura para *checkpointing* quase-síncrono foi mostrado e sua implementação dentro do LAM/MPI foi detalhada. Essa infra-estrutura modifica como o LAM/MPI utiliza o BLCR para gravar localmente o estado dos processos da aplicação, e também insere as informações de controle do protocolo quase-síncrono em todas as mensagens, de modo que o protocolo possa processá-las sempre que uma mensagem é recebida. Trata-se de uma infra-estrutura que até onde se conhece não foi implementada em nenhuma outra biblioteca de passagem de mensagens do padrão MPI. Ela permite que a maioria dos protocolos de *checkpointing* quase-síncronos sejam implementados e testados juntamente com as aplicações MPI.

Outra contribuição mostrada neste capítulo foi a arquitetura de *software* Curupira, que usa a infra-estrutura para *checkpointing* quase-síncrono implementada no LAM/MPI junto com algoritmos de coleta de lixo e recuperação por retrocesso de estado para prover tolerância a falhas a algumas aplicações MPI. Os protocolos quase-síncronos que podem ser usados no Curupira são da classe RDT, de modo que o algoritmo RDT-LGC [42, 43, 44] pode realizar a coleta de lixo localmente em cada processo da aplicação. Isto permite que

em uma execução sem falhas, nenhuma mensagem de controle seja enviada pelos processos da aplicação e nenhum tipo de sincronização seja necessária.

Por fim, algumas comparações utilizando aplicações MPI foram feitas no CURUPIRA, de modo a avaliar o RDT-Minimal [26] e o RDT-LGC. Percebeu-se que o RDT-Minimal sempre induziu menos *checkpoints* forçados que o FDAS [49], mas a degradação provocada no desempenho da computação por ambos os protocolos foi grande. Embora o limite para o número de *checkpoints* mantidos pelo RDT-LGC seja n *checkpoints* por processo, apenas em uma aplicação esse limite foi atingido. Isto sugere que assim como o número de *checkpoints* induzidos pelo RDT-Minimal, o número máximo de *checkpoints* mantidos pelo RDT-LGC seja fortemente influenciado pelo padrão de comunicação da aplicação.

Embora a degradação no desempenho tenha sido considerável, em uma comparação com o protocolo de *checkpointing* síncrono do LAM/MPI, que foi modificado para poder ser ativado pela própria aplicação, o RDT-Minimal obteve resultados melhores que esse protocolo síncrono ou bem próximos. Outros estudos já utilizaram simulações para tentar avaliar o desempenho e a escalabilidade de protocolos quase-síncronos [9], chegando a concluir que protocolos quase-síncronos não escalam bem com o aumento no número de processos, em oposição aos protocolos síncronos. No entanto, os dados obtidos com o CURUPIRA parecem indicar o contrário, pois o RDT-Minimal se mostrou pelo menos tão eficiente quanto o protocolo síncrono do LAM/MPI, que é uma variação do algoritmo de Chandy e Lamport [17]. Desta forma, percebe-se que a utilização de protocolos de *checkpointing* quase-síncronos RDT com aplicações reais parece ser promissora, principalmente levando-se em consideração que o tempo de gravação dos *checkpoints* pode ser otimizado, utilizando-se a chamada de sistema `fork()` antes de gravar o *checkpoint*. No entanto, mais estudos com aplicações reais precisam ser realizados, levando-se em conta o tempo de execução e o padrão de comunicação das aplicações, para que uma conclusão melhor possa ser feita.

Capítulo 6

Conclusão e Trabalhos Futuros

Esta dissertação abordou vários aspectos teóricos e práticos envolvidos em uma arquitetura de *software* para recuperação de falhas que utilize protocolos de *checkpointing* quase-síncronos. No Capítulo 2 foram vistos os conceitos básicos que são utilizados na teoria dos protocolos de *checkpointing*. Além disso, também foram mostradas as abordagens diferentes que existem para *checkpointing*, dando ênfase na abordagem quase-síncrona, que não utiliza mensagens de controle e nenhum tipo de sincronização entre os processos da aplicação. Dentro dessa abordagem, foram mostradas as classes de protocolos existentes, que diferem entre si pelas dependências entre os *checkpoints* que são permitidas nos padrões de *checkpoints* e mensagens que são gerados. Para serem utilizados em tolerância a falhas com retrocesso de estado, as classes ZCF, ZPF e SZPF são as mais interessantes, pois não permitem a existência de *checkpoints* inúteis, que são aqueles que não podem ser utilizados em nenhum estado global consistente da aplicação formado por um *checkpoint* de cada processo, chamado de *checkpoint* global consistente. Estes estados globais consistentes da aplicação que são utilizados na recuperação por retrocesso de estado. Os protocolos das classes ZPF e SZPF possuem uma propriedade chamada de RDT [49] (*Rollback-Dependency Trackability*), que facilita a construção de *checkpoints* globais consistentes, pois as dependências entre os *checkpoints* são rastreáveis em tempo de execução utilizando-se vetores de dependência. Embora ofereçam essa facilidade, os protocolos RDT costumam induzir um número muito grande de *checkpoints* forçados. Assim, vários estudos foram realizados e o protocolo RDT-Minimal [26] foi descoberto por Garcia e Buzato. Ele implementa a caracterização minimal de um protocolo que garante a propriedade RDT, tentando minimizar o número de *checkpoints* forçados gerados por protocolos dessa classe.

No Capítulo 3 foram vistos alguns algoritmos de recuperação e de coleta de lixo que podem ser usados em conjunto com protocolos de *checkpointing*. Os algoritmos de recuperação são encarregados de retroceder a aplicação para o estado global consistente

o mais recente possível. Esse estado é o *checkpoint* global consistente mais recente no padrão de *checkpoints* e mensagens que minimiza o custo de retrocesso, e também é chamado de linha de recuperação. Em específico, foram vistos métodos para se calcular a linha de recuperação utilizando-se uma estrutura conhecida como *R-graph*, também chamada de grafo de dependências de retrocesso [48, 49]. Também foi visto que protocolos que geram padrões RDT facilitam o cálculo da linha de recuperação, que pode ser feito utilizando-se algumas relações com os vetores de dependências [48, 49]. A coleta de lixo é responsável por identificar e eliminar os *checkpoints* que se tornam obsoletos à medida que a computação progride. Um *checkpoint* é obsoleto se ele não será mais usado em nenhuma linha de recuperação que pode ser utilizada para retroceder a aplicação. A coleta de lixo para protocolos quase-síncronos necessita de informação global para que as linhas de recuperação sejam calculadas, e os *checkpoints* obsoletos sejam identificados. Porém, em protocolos RDT, foi descoberto que a coleta de lixo pode ser feita localmente em cada processo, utilizando apenas informações de controle que são propagadas junto com as mensagens da aplicação, assim como fazem os protocolos de *checkpointing* quase-síncronos. Este algoritmo, chamado de RDT-LGC [42, 43, 44], também estabelece um número máximo de *checkpoints* que precisam ser mantidos por processo, o que o torna interessante para ser utilizado em um sistema de recuperação de falhas que utilize *checkpointing* quase-síncrono RDT.

Como esta dissertação aborda questões práticas da utilização de protocolos de *checkpointing* em bibliotecas de passagem de mensagens, no Capítulo 4 foram analisados os trabalhos relacionados mais interessantes. Nestes trabalhos, protocolos de *checkpointing* síncrono e assíncronos foram utilizados para fornecer tolerâncias a falhas para aplicações distribuídas construídas, principalmente, com o auxílio de bibliotecas de passagem de mensagens do padrao MPI [2]. Até onde se conhece, nenhuma biblioteca de passagem de mensagens possui protocolos de *checkpointing* quase-síncronos implementados, como foi mostrado com a análise feita nesse capítulo.

No Capítulo 5 foram descritas as principais contribuições desta dissertação. Primeiramente, uma infra-estrutura para *checkpointing* quase-síncrono [18] foi modelada e implementada na biblioteca de passagem de mensagens do padrão MPI conhecida como LAM/MPI [1]. Com essa infra-estrutura é possível implementar a maioria dos protocolos de *checkpointing* quase-síncronos e utilizá-los durante a execução das aplicações. Para gravar o estado dos processos, uma abordagem que utiliza o sistema operacional é empregada por meio do BLCR [21]. Dessa maneira, o estado dos processos são gravados de forma transparente para a aplicação MPI que está sendo executada.

Ainda no Capítulo 5, uma outra contribuição foi mostrada, que é o modelo de uma arquitetura de *software* para recuperação de falhas utilizando protocolos de *checkpointing* quase-síncronos RDT. Esta arquitetura foi chamada de Curupira, e foi implementada

no LAM/MPI aproveitando-se a infra-estrutura para *checkpointing* quase-síncrono implementada anteriormente. A classe RDT de protocolos quase-síncronos foi escolhida, pois ela oferece várias vantagens para ser utilizada na prática. Entre as vantagens dos protocolos RDT estão a maior facilidade no cálculo da linha de recuperação [48, 49], a capacidade de realizar a coleta de lixo localmente em cada processo com o RDT-LGC [42, 43, 44] e o retrocesso da computação é menor no caso de uma falha [6]. O Curupira pode ser considerado, até onde se conhece, a primeira arquitetura de *software* para recuperação de falhas implementada que utiliza protocolos de *checkpointing* quase-síncronos. Além disso, também é a primeira que numa execução sem falhas da aplicação, permite que a coleta de lixo seja realizada localmente em cada processo. Dessa maneira, assim como os protocolos quase-síncronos, a coleta de lixo utilizando o RDT-LGC é realizada sem o envio de mensagens de controle ou a sincronização global dos processos.

O Curupira contém as primeiras implementações do protocolo RDT-Minimal e do algoritmo de coleta de lixo RDT-LGC. Assim, uma outra contribuição desta dissertação, também vista no Capítulo 5, são as comparações realizadas entre o RDT-Minimal e o FDAS [49] quanto ao número de *checkpoints* forçados e os dados obtidos do RDT-LGC de quantos *checkpoints* foram mantidos no máximo por algum processo durante a execução de algumas aplicações. Percebeu-se que o RDT-Minimal sempre induz menos *checkpoints* forçados que o FDAS, mas essa diferença depende muito do padrão de comunicação que a aplicação possui. Quanto ao RDT-LGC, pôde-se notar que em algumas aplicações o máximo teórico de n *checkpoints* era atingido, mas no geral poucos *checkpoints* eram mantidos em cada processo da aplicação. Além disso, uma comparação foi realizada entre os protocolos quase-síncronos RDT-Minimal e FDAS e o protocolo de *checkpointing* síncrono existente no LAM/MPI. Embora estudos teóricos e simulações indicassem que protocolos quase-síncronos não seriam interessantes para serem utilizados na prática [9], os resultados obtidos com as aplicações testadas mostraram o contrário, pois o RDT-Minimal teve sempre um desempenho melhor ou igual ao do protocolo síncrono do LAM/MPI.

Acredita-se que existam várias possibilidades de extensão deste trabalho. Uma delas seria implementar um algoritmo de recuperação melhor no Curupira, de modo que o retrocesso pudesse ser feito automaticamente. Aliado a isto, um mecanismo de detecção de falhas melhor do que o existente no Curupira poderia ser estudado e implementado, de modo que o retrocesso da computação no caso de uma falha pudesse ser minimizado. Uma outra linha de pesquisa poderia tentar ampliar o Curupira de modo a suportar todas as funcionalidades contidas no padrão MPI, dado que este não foi projetado considerando tolerância a falhas. Da maneira que foi implementado, o Curupira considera que os canais de comunicação entre os processos da aplicação são não-confiáveis, e assim uma extensão poderia adaptar o Curupira para que a aplicação pudesse ser retrocedida para um *snapshot* consistente [30, 35]. Por fim, uma otimização poderia ser feita no BLCR, para

que a gravação dos estados dos processos pudesse ser feita em paralelo com a execução deles. Dessa forma, o desempenho dos protocolos quase-síncronos implementados no Curupira poderia ser melhorado.

Bibliografia

[1] LAM/MPI Parallel Computing. Homepage oficial: `http://www.lam-mpi.org/`. (consultado em 13/12/2005).

[2] Message Passing Interface Forum. Homepage oficial: `http://www.mpi-forum.org/`. (consultado em 13/12/2005).

[3] PVM: Parallel Virtual Machine. Homepage oficial: `http://www.csm.ornl.gov/pvm/`. (consultado em 13/12/2005).

[4] The Globus Alliance. Homepage oficial: `http://www.globus.org/`. (consultado em 13/12/2005).

[5] The Linux Kernel Archives. Homepage oficial: `http://www.kernel.org/`. (consultado em 13/12/2005).

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

[7] A. Agbaria and R. Friedman. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. In *8th IEEE International Symposium on High Performance Distributed Computing*, pages 167–176, 1999.

[8] C. E. R. Alves, E. N. Cáceres, S. W. Song A. A. Castro Jr, and J. L. Szwarcfiter. Efficient Parallel Implementation of Transitive Closure of Digraphs. In *EuroPVM/MPI User's Group Meeting*, 2003.

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

[10] R. T. Aulwes, D. J. Daniel, N. N. Desai, R. L. Graham, L. D. Risinger, M. A. Taylor, and T. S. Woodall. Architecture of LA-MPI, A Network-Fault-Tolerant MPI. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, page 15, 2004.

[11] R. Baldoni, J. M. Helary, A. Mostefaoui, and M. Raynal. A Communication-Induced Checkpoint Protocol that Ensures Rollback Dependency Trackability. In *IEEE Symposium on Fault Tolerant Computing*, pages 68–77, 1997.

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

[13] R. Baldoni, J. M. Helary, and M. Raynal. Rollback-Dependency Trackability: A Minimal Characterization and its Protocol. *Information and Computation*, 165(2):144–173, March 2001.

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

[15] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fédak, C. Germain, T. Hérault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Néri, and A. Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *SuperComputing 2002*, Baltimore, November 2002.

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

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

[18] U. F. F. da Silva and I. C. Garcia. *Checkpointing* Quase-Síncrono no LAM/MPI. In *Workshop em Sistemas Computacionais de Alto Desempenho*, Foz do Iguaçu, Paraná, October 2004.

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

[20] R. Y. de Camargo, A. Goldchleger, F. Kon, and A. Goldman. Checkpointing-based Rollback Recovery for Parallel Applications on the InteGrade Grid Middleware. In *2nd Workshop on Middleware for Grid Computing*, pages 35–40, 2004.

[21] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Publicação eletrônica disponível em: `http://ftg.lbl.gov/twiki/pub/Whiteboard/CheckpointPapers/blcr.pdf`, 2003. (consultado em 13/12/2005).

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

[23] G. Fagg and J. Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *EuroPVM/MPI User's Group Meeting*, 2000.

[24] I. C. Garcia and L. E. Buzato. Using Common Knowledge to Improve Fixed-Dependency-After-Send. In *Workshop de Testes e Tolerância a Falhas*, Curitiba, Paraná, julho 2000.

[25] I. C. Garcia and L. E. Buzato. On the Minimal Characterization of Rollback-Dependency Trackability Property. In *Proceedings of the 21th IEEE Int. Conf. on Distributed Computing Systems*, pages 342–349, Phoenix, Arizona, EUA, April 2001.

[26] I. C. Garcia and L. E. Buzato. An Efficient Checkpointing Protocol for the Minimal Characterization of Operational Rollback-Dependency Trackability. In *23rd Symposium on Reliable Distributed Systems*, pages 126–135, October 2004.

[27] I. C. Garcia, G. M. D. Vieira, and L. E. Buzato. RDT-Partner: An Efficient Checkpointing Protocol that Enforces Rollback-Dependency Trackability. In *Simpósio Brasileiro de Redes de Computadores*, Florianópolis, Santa Catarina, May 2001.

[28] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G. C. Bezerra. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice and Experience*, 16:449–459, 2004.

[29] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[30] J. M. Helary, A. Mostefaoui, and M. Raynal. Communication-Induced Determination of Consistent Snapshots. *IEEE Trans. Parallel Distrib. Syst.*, 10(9):865–877, 1999.

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

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

[33] M. Litzkow, M. Livny, and M. Mutka. Condor – A Hunter of Idle Workstations. In *8th IEEE Int. Conf. on Distributed Computing Systems*, pages 104–111, 1988.

[34] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical report, Computer Sciences Department, University of Wisconsin, 1997.

[35] D. Manivannan and M. Singhal. A Low-overhead Recovery Technique Using Quasi-Synchronous Checkpointing. In *16th Int. Conference on Distributed Computing Systems*, pages 100–107, May 1996.

[36] D. Manivannan and M. Singhal. Quasi-Synchronous Checkpointing: Models, Characterization, and Classification. *IEEE Trans. Parallel Distrib. Syst.*, 10(7):703–713, 1999.

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

[38] N. Neves and W. K. Fuchs. RENEW: A Tool for Fast and Efficient Implementation of Checkpoint Protocols. In *Symposium on Fault-Tolerant Computing*, pages 58–67, 1998.

[39] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

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

[41] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. In *LACSI Symposium*, October 2003.

[42] R. M. Schmidt. Coleta de Lixo para Protocolos de *Checkpointing*. Master's thesis, Instituto de Computação—Universidade Estadual de Campinas, 2003.

[43] R. M. Schmidt, I. C. Garcia, F. Pedone, and L. E. Buzato. Optimal asynchronous garbage collection for checkpointing protocols with rollback-dependency trackability. In *23rd ACM Symposium on the Principles of Distributed Computing*, July 2004. (Brief Announcement).

[44] R. M. Schmidt, I. C. Garcia, F. Pedone, and L. E. Buzato. Optimal Asynchronous Garbage Collection for RDT Checkpointing Protocols. In *25th International Conference on Distributed Computing Systems*, June 2005. (Aceito para publicação).

[45] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *10th International Parallel Processing Symposium (IPPS)*, pages 526–531, Honolulu, Hawaii, 1996.

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

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

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

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

[50] N. Woo, H. Y. Yeom, and T. Park. MPICH-GF: Transparent Checkpointing and Rollback-Recovery for GRID-enabled MPI Processes. In *The 2nd Workshop on Hardware/Software Support for High Performance Scientific and Engineering Computing(SHPSEC03)*, New Orleans, Louisiana, September 2003.

[51] H. Zhong and J. Nieh. CRAK: Linux Checkpoint/Restart as a Kernel Module. Technical Report CUCS-014-01, Department of Computer Science, Columbia University, 2001.