Uma Ponte entre as Abordagens Síncrona e Quase-síncrona para Checkpointing

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Tiemi Christine Sakata e aprovada pela Banca Examinadora.

Campinas, 8 de Janeiro de 2007.

Profa. Dra. Islene Calciolari Garcia
(Orientadora)

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

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Sakata, Tiemi Christine

Sa29p Uma ponte entre as abordagens síncrona e quase-síncrona para checkpointing / Tiemi Christine Sakata -- Campinas, [S.P. :s.n.], 2006.

Orientador : Islene Calciolari Garcia

Tese (doutorado) - 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: Bridging the gap between synchronous and quase-synchronous checkpointing.

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

Área de concentração: Sistemas Distribuídos

Titulação: Doutora em Ciência da Computação

Banca examinadora: Profa. Dra. Islene Calciolari Garcia (IC-UNICAMP)
Profa. Dra. Fabíola Gonçalves Pereira Greve (DCC-UFBA)
Prof. Dr. Elias Procópio Duarte Júnior (DI-UFPR)
Prof. Dr. Ricardo de Oliveira Anido (IC-UNICAMP)
Prof. Dr. Edmundo Roberto Mauro Madeira (IC-UNICAMP)

Data da defesa: 21-12-2006

Programa de Pós-Graduação: Doutorado em Ciência da Computação

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 21 de dezembro de 2006, pela Banca examinadora composta pelos Professores Doutores:

Profa. Dra. Fabíola Gonçalves Pereira Greve
UFBA

Prof. Dr. Elias Procópio Duarte Júnior
UFPR

Prof. Dr. Ricardo de Oliveira Anido
IC - UNICAMP

Prof. Dr. Edmundo Roberto Mauro Madeira
IC - UNICAMP

Profa. Dra. Islene Calciolari Garcia
IC - UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Uma Ponte entre as Abordagens Síncrona e Quase-síncrona para Checkpointing

Tiemi Christine Sakata[1]

Dezembro de 2006

Banca Examinadora:

- Profa. Dra. Islene Calciolari Garcia (Orientadora)
- Profa. Dra. Fabíola Gonçalves Pereira Greve
  Departamento de Ciência da Computação—UFBA
- Prof. Dr. Elias Procópio Duarte Júnior
  Departamento de Informática—UFPR
- Prof. Dr. Ricardo de Oliveira Anido
  Instituto de Computação—UNICAMP
- Prof. Dr. Edmundo Roberto Mauro Madeira
  Instituto de Computação—UNICAMP
- Profa. Dra. Maria Beatriz Felgar de Toledo (Suplente)
  Instituto de Computação—UNICAMP
- Prof. Dr. Ivan Luiz Marques Ricarte (Suplente)
  Faculdade de Engenharia Elétrica e de Computação—UNICAMP

[1]Suporte financeiro de: Bolsa do CNPq (processo 141808/01-2) 2001–2005

Resumo

Protocolos de *checkpointing* são responsáveis pelo armazenamento de estados dos processos de um sistema distribuído em memória estável para tolerar falhas. Os protocolos síncronos minimais induzem apenas um número minimal de processos a salvarem *checkpoints* durante uma execução do protocolo bloqueando os processos envolvidos. Uma versão não-bloqueante desta abordagem garante a minimalidade no número de *checkpoints* salvos em memória estável com o uso de *checkpoints* mutáveis, *checkpoints* que podem ser salvos em memória não-estável. Porém, a complexidade deste protocolo e o fato de ele tolerar apenas a presença de uma execução de *checkpointing* a cada instante nos motivou a procurar soluções para estes problemas na teoria desenvolvida para os protocolos quase-síncronos.

A nova abordagem nos permitiu fazer uma revisão de alguns protocolos síncronos bloqueantes existentes na literatura que até então eram considerados minimais. Nesta mesma linha, obtivemos novos resultados na análise de minimalidade dos protocolos síncronos não-bloqueantes, ao considerarmos a aplicação como um todo e também a existência de execuções concorrentes de *checkpointing*.

Ao estabelecermos esta ponte entre as abordagens para *checkpointing*, conseguimos desenvolver dois novos protocolos síncronos não-bloqueantes. Ambos fazem uso de *checkpoints* mutáveis, permitem execuções concorrentes de *checkpointing* e possuem um mecanismo simples de coleta de lixo. No entanto, o fato de cada um dos protocolos derivar de classes diferentes de protocolos quase-síncronos leva a comportamentos distintos, como evidenciado por resultados de simulação.

Abstract

Checkpointing protocols are responsible for the selection of checkpoints in fault-tolerant distributed systems. Minimal checkpointing protocols minimize the number of checkpoints blocking processes during checkpointing. A non-blocking version of this approach assures a minimal number of checkpoints saved in stable memory using mutable checkpoints, those checkpoints can be saved in a non-stable storage. However, the complexity of this protocol and the absence of concurrent checkpointing executions have motivated us to find new solutions in the quasi-synchronous theory.

The new approach has allowed us to review some blocking synchronous protocols existent in the literature which were, until now, considered as minimals. In the same way, we present new results analysing the minimality on the number of checkpoints in non-blocking synchronous protocols, considering the whole application and also the existence of concurrent checkpointing executions.

On bridging the gap between the checkpointing approaches we could develop two new non-blocking synchronous protocols. Both use mutable checkpoints, allow concurrent checkpointing executions and have a simple mechanism of garbage collection. However, since each protocol derives from a different class of quasi-synchronous protocols, they present distinct behaviours, which are evident in the simulation results.

Agradecimentos

À Islene, pela sua sabedoria, compreensão e amizade. Ensinou-me a escrever melhor, ser justa, pensar e tentar entender os outros e, até mesmo a ser mãe.

Ao Buzato, que confiou na minha capacidade e acreditou antes de mim mesma na possiblidade deste doutorado.

Ao Raphael, Ulisses e Rodrigo por terem contribuído com importantes sugestões.

Ao Gustavo pela implementação do simulador ChkSim e por estar sempre à disposição para me ajudar em diversos aspectos.

À Cândida pela amizade, carinho e por compartilhar os momentos bons e os não tão bons.

A todos os amigos do Laboratório de Sistemas Distribuídos pela amizade.

A todos os professores do Instituto de Computação da Unicamp por participarem desse longo, mas válido caminho.

A todos os professores que participaram de alguma forma, na minha formação.

Aos colegas e funcionários da Unicamp pela colaboração ao longo desses anos.

Aos amigos que, de perto ou longe, acompanharam o meu crescimento e estiveram sempre presentes de forma calorosa em meu coração.

Ao meu pai e minha mãe pelo incentivo, amor e educação que foram fundamentais para que eu conseguisse caminhar até aqui.

Aos meus irmãos por estarem sempre ao meu lado e serem pessoas em que posso contar a todo momento.

A todos os membros da minha família pelo carinho caloroso de sempre.

Finalmente, mas não menos importante, quero agradecer ao Luiz, pelo amor, compreensão, carinho, apoio e agora, pelo bebê que certamente me motivou a terminar este trabalho.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Um sistema computacional distribuído é composto por uma coleção de computadores autônomos interligados por uma rede de comunicação e é visto pelos usuários como um único sistema coerente [40]. Sistemas distribuídos estão evoluindo e se diversificando com o advento de novas tecnologias como computação móvel, grades e aglomerados computacionais. Nestes ambientes podem ser executadas aplicações relacionadas a várias áreas, dentre as quais podemos citar otimização combinatória, genômica, processamento de imagens, controle aéreo ou pesquisas que derivam suas conclusões de simulações de longa duração. Estas aplicações podem se beneficiar de melhor desempenho, escalabilidade e compartilhamento transparente de recursos existentes no sistema, porém estão mais suscetíveis a falhas. Assim, o desafio de fornecer mecanismos de tolerância a falhas nessas aplicações se torna cada dia mais importante.

Na ocorrência de uma falha em um processo, o sistema deve encontrar uma maneira de prosseguir sua execução de forma coerente. Na recuperação por avanço, a aplicação deve possuir códigos específicos para ser capaz de contornar inconsistências e prosseguir seu processamento. Este mecanismo é dependente de aplicação e de um conjunto de falhas previstas. Na recuperação por retrocesso o sistema grava periodicamente o estado dos processos em memória estável e é capaz de retroceder para um estado consistente quando necessário [14]. Este mecanismo é mais geral, independente de aplicação e ele é o foco do nosso trabalho.

Um estado de um processo salvo em memória estável é chamado de *checkpoint*[1] e um *checkpoint* global é formado por um *checkpoint* por processo do sistema. Como os processos se comunicam via troca de mensagens, um *checkpoint* global da aplicação só é consistente se, para todo evento de recepção de mensagem foi também anotado o seu envio. Se os *checkpoints* dos processos de uma aplicação distribuída forem escolhidos ale-

[1]Em português: ponto de recuperação ou ponto de controle. Adotamos o termo em inglês por ser mais resumido e amplamente utilizado.

atoriamente, haverá o risco de que não se consiga formar um estado consistente [32]. Os protocolos para *checkpointing* são responsáveis pela seleção de *checkpoints* e apresentam características muito distintas de acordo com a abordagem utilizada. Uma dessas características é a facilidade com que a coleta de lixo pode ser realizada, ou seja, o descarte dos *checkpoints* obsoletos que não serão usados para recuperação devido ao progresso da aplicação.

Abordagens para *Checkpointg*

Os protocolos para *checkpointing* são classificados como: assíncronos, quase-síncronos e síncronos [25]. Na abordagem assíncrona, os processos têm autonomia na escolha de seus *checkpoints*, porém alguns desses *checkpoints*, chamados de *checkpoints* inúteis, podem não fazer parte de nenhum *checkpoint* global consistente [27]. Além disso, não há garantias de que seja possível formar um *checkpoint* global consistente a partir dos *checkpoints* salvos. Portanto, pode ocorrer o efeito dominó, ou seja, a aplicação pode ser obrigada a retornar ao estado inicial em caso de falha [32]. Os protocolos assíncronos não utilizam mensagens de controle nem necessitam enviar informações de controle junto com as mensagens da aplicação e devem ter mecanismos separados para construir *checkpoints* globais consistentes e para a coleta de lixo.

Os protocolos quase-síncronos (ou induzidos por comunicação) propostos na literatura reduzem ou eliminam o número de *checkpoints* inúteis, induzindo o armazenamento de *checkpoints* adicionais aos *checkpoints* do protocolo assíncrono [14, 24]. Nesta abordagem, os processos salvam seus *checkpoints* livremente, chamados *checkpoints* básicos e podem ser induzidos a armazenarem *checkpoints* adicionais, chamados *checkpoints* forçados, segundo predicados avaliados sobre informações de controle propagadas via *piggybacking* pelas próprias mensagens da aplicação. Posteriormente, *checkpoints* globais consistentes podem ser formados a partir dos *checkpoints* selecionados e a coleta de lixo pode ser realizada. Portanto, três protocolos distintos são necessários: um para a seleção de *checkpoints*, um para a construção de *checkpoints* globais consistentes e um para a coleta de lixo. Há um compromisso entre a autonomia dos processos para a escolha dos *checkpoints* e as garantias oferecidas para a formação de *checkpoints* globais consistentes.

Em contraste com as duas classes de protocolos já mencionadas, os protocolos síncronos constroem *checkpoints* globais consistentes utilizando fases de troca de mensagens de controle que sincronizam processos sinalizando o momento de início e término da execução do protocolo [21]. O início é determinado pela invocação por algum dos processos, o iniciador, do protocolo. O término é determinado pela verificação de uma condição que indica que todos os processos envolvidos na execução já armazenaram seus *checkpoints*. A execução do protocolo, desde o início até seu término é denominada construção consistente.

Ao final de uma construção consistente, o *checkpoint* global consistente representado pelos últimos *checkpoints* locais é o *checkpoint* global consistente mais recente, e portanto, em cada processo, a coleta de lixo restringe-se ao descarte do *checkpoint* anterior a este.

Os primeiros protocolos síncronos propostos na literatura fazem com que todos os processos da aplicação salvem seus *checkpoints* de forma coordenada para a construção de um *checkpoint* global consistente [11, 13, 39]. O alto número de mensagens de controle somado ao alto custo do armazenamento de *checkpoints* gerado por estes protocolos motivou o desenvolvimento de protocolos que reduzem o número de mensagens de controle e o número de *checkpoints* salvos durante uma construção consistente. Chamamos de protocolos minimais os protocolos que determinam o conjunto minimal de processos que devem gravar um *checkpoint* para garantir a construção de um *checkpoint* global consistente [8, 20, 22, 28, 35, 36].

Cao e Singhal demonstraram que todo protocolo minimal precisa ser bloqueante, ou seja, para armazenar um número minimal de *checkpoints* durante uma construção consistente é necessário que os processos envolvidos suspendam suas atividades da computação sob o ponto de vista da aplicação [10]. Porém, a suspensão da aplicação reduz o desempenho do sistema. Na tentativa de se obter um desempenho melhor para a aplicação, os protocolos síncronos não-bloqueantes permitem que todos os processos gravem *checkpoints* a cada construção consistente sem suspender as atividades da computação [9, 7, 8, 30]. Cao e Singhal propuseram um novo tipo de *checkpoint*, chamado de *checkpoint* mutável, que é facilmente manipulado pois pode ser salvo em memória não-estável [7, 8]. Os *checkpoints* mutáveis são armazenados para evitar a formação de *checkpoints* globais inconsistentes e são transferidos para memória estável apenas quando são necessários para uma construção consistente, garantindo assim a minimalidade no número de *checkpoints* em memória estável (*checkpoints* estáveis) [8]. Estes protocolos são baseados em protocolos minimais e são muito complexos, pouco eficientes e permitem apenas uma construção consistente a cada instante de tempo.

Até este momento, os protocolos quase-síncronos e síncronos propostos na literatura foram desenvolvidos e analisados utilizando modelos distintos, apesar do conceito de consistência de um *checkpoint* global ser único para os dois modelos (para todo evento de recepção de mensagem incluído no *checkpoint* global deve ser também anotado o seu envio). No modelo quase-síncrono, os protocolos evitam a formação de certas classes de *zigzag paths* [27]. As *zigzag paths* são seqüência de mensagens que indicam as condições necessárias e suficientes para que um par de *checkpoints* possa fazer parte de um mesmo *checkpoint* global consistente. A z-dependência é a dependência entre processos em intervalo de *checkpoints* e indicam se um processo deve salvar um *checkpoint* no momento da recepção de uma mensagem de requisição de uma construção consistente [8].

Contribuições

Este trabalho propõe o uso dos conceitos e da teoria existentes para os protocolos quase-síncronos para analisar e desenvolver protocolos síncronos. Desta forma, obtivemos as seguintes contribuições:

- Verificação da relação de continência do conceito de z-dependência proposto para os protocolos síncronos [10, 7] na relação de z-precedência utilizada pelos protocolos quase-síncronos [16]. A z-precedência é uma outra interpretação do conceito de *zigzag paths* e foi adotado na maioria das provas descritas nesta tese.

- Prova de que protocolos minimais que anotam a recepção de mensagens, mas não os intervalos de origem dessas mensagens, não têm informação suficiente para garantir minimalidade no número de *checkpoints*. Mostramos que os protocolos propostos em [8, 28] não são minimais e propomos dois novos protocolos minimais, chamados de VD-minimal [35] e Broad-minimal [36] para corrigir os problemas encontrados.

- Prova da impossibilidade de um protocolo síncrono não-bloqueante garantir um número mínimo de *checkpoints* estáveis durante toda a execução da aplicação, mesmo na presença de um único iniciador a cada instante.

- Prova da impossibilidade de um protocolo síncrono não-bloqueante salvar um número minimal de *checkpoints* estáveis na presença de iniciadores concorrentes.

- Proposta de dois novos protocolos síncronos não-bloqueantes baseados em protocolos quase-síncronos: RDT-NBS [33] e BCS-NBS. Estes protocolos foram baseados em classes distintas de protocolos quase-síncronos e obtivemos como resultado protocolos mais simples que os existentes e que permitem iniciadores concorrentes.

- Experimento inicial de simulação obtidos com o uso do ChkSim [43] que contrastam o comportamento dos protocolos quase-síncronos FDAS e BCS e suas respectivas versões síncronas não-bloqueantes RDT-NBS e BCS-NBS.

Estrutura da Tese

O Capítulo 2 apresenta o modelo computacional considerado em todos os protocolos descritos neste trabalho. Neste capítulo são descritas as definições de *checkpoint* e *checkpoint* global consistente e os mecanismos existentes para rastrear as dependências entre *checkpoints*. Em seguida mostramos que a definição de z-dependência do modelo síncrono está contida na noção de z-precedência utilizada na teoria dos protocolos quase-síncronos.

O Capítulo 3 descreve em detalhes as classes de protocolos de *checkpointing* existentes na literatura. Os protocolos quase-síncronos podem ser classificados de acordo com os conceitos de *zigzag path* e *zigzag cycle*: *Strictly Z-Path Free* (SZPF), *Z-Path Free* (ZPF), *Z-Cycle Free* (ZCF), *Partially Z-Cycle Free* (PZCF), sendo a classe SZPF a mais restritiva e a PZCF a menos restritiva [24]. Os protocolos síncronos analisados nesta tese garantem a construção de *checkpoints* globais consistentes reduzindo o número de *checkpoints* salvos durante uma execução de *checkpointing* e podem ser bloqueantes ou não-bloqueantes. Os protocolos minimais são bloqueantes e podem induzir um número minimal de *checkpoints* enquanto os protocolos não-bloqueantes não suspendem as atividades da computação nos processos porém não garantem a minimalidade no número de *checkpoints*.

O Capítulo 4 descreve as características de um protocolo minimal na presença de um único e de iniciadores concorrentes. Provamos que a minimalidade no número de *checkpoints* não pode ser garantida anotando-se apenas a recepção de mensagens. Encontramos na literatura, duas abordagens na construção de protocolos minimais: *broadcast* e em níveis. Propomos um protocolo minimal com a abordagem *broadcast* que corrige um erro encontrado na literatura. Em seguida, propomos um protocolo minimal com a abordagem em níveis que utiliza vetores de dependências e permite iniciadores concorrentes.

O Capítulo 5 descreve as características de um protocolo não-bloqueante que reduz o número de *checkpoints* durante uma execução na presença de um único iniciador e de iniciadores concorrentes. Estes protocolos podem garantir um número minimal de *checkpoints* em memória estável utilizando *checkpoints* mutáveis. Notamos que a minimalidade no número de *checkpoints* estáveis depende do ponto de execução em que os *checkpoints* mutáveis são salvos e provamos que não é possível garantir um número mínimo de *checkpoints* estáveis durante toda a execução da aplicação. Além disso, provamos que é impossível garantir minimalidade no número de *checkpoints* estáveis na presença de iniciadores concorrentes. Neste capítulo, mostramos que é possível desenvolver protocolos não-bloqueantes baseados em protocolos quase-síncronos. Propomos dois novos protocolos: RDT-NBS baseado no protocolo FDAS da classe RDT e BCS-NBS baseado no protocolo BCS da classe ZCF e apresentamos resultados de simulação para comparar esses protocolos.

O Capítulo 6 encerra o texto com a conclusão deste estudo e trabalhos futuros.

Capítulo 2

Checkpoints Globais Consistentes

Na ocorrência de uma falha é necessário que o sistema estabeleça um estado global recuperável formado por *checkpoints* pertencentes a cada processo da aplicação distribuída. No entanto, a troca de mensagens da aplicação dificulta a construção de *checkpoints* globais consistentes pois causa dependência entre eventos. Lamport definiu o conceito de precedência causal para a ordenação parcial dos eventos [21]. Assim, podemos afirmar que um *checkpoint* global é formado por um *checkpoint* por processo e é consistente se não existe relação de causalidade entre os *checkpoints* do conjunto.

Neste capítulo, apresentamos as definições e conceitos utilizados na teoria de *checkpointing* necessários para a compreensão dos resultados deste estudo. Inicialmente, apresentamos o modelo computacional adotado por qualquer protocolo proposto nesta tese (Seção 2.1). Em seguida, apresentamos a estrutura de um processo e o conceito de precedência causal (Seções 2.2 e 2.3). Na Seção 2.4 descrevemos o conceito *zigzag paths* e mostramos que o conceito de z-dependência está contido no conceito de z-precedência. Na seqüência, apresentamos o conceito de *checkpoints* globais consistentes (Seção 2.5).

2.1 Modelo Computacional

Um sistema distribuído é composto por n processos seqüenciais e autônomos que executam eventos internos, de envio e de recepção (troca) de mensagens. A troca de mensagens é o único mecanismo de comunicação utilizado pelos processos. Consideramos que a rede de comunicação é fortemente conexa (nenhum processo é isolado), mas não necessariamente completa (a comunicação entre um par de processos pode se dar por processos intermediários). Os canais de comunicação são unidirecionais e confiáveis, ou seja, há garantia de entrega de mensagens, mas estas podem sofrer atrasos arbitrários e chegar aos seus destinos fora de ordem. Em alguns poucos protocolos neste texto é necessária a propriedade FIFO (*First-In First-Out*) nos canais de comunicação entre pares de processos.

Consideramos que não existem mecanismos para compartilhamento de memória, acesso a um relógio global, sincronização de relógios locais ou conhecimento a respeito das diferenças de velocidade entre os processadores. A memória estável é suficiente para gravar todos os *checkpoints* necessários à computação, garantindo a correta recuperação de seu estado em caso de falha. Os processos podem sofrer falhas do tipo *crash* nas quais param e perdem o seu estado atual de execução. Quando um processo falha, algum outro processo deverá detectar a falha e informar os outros processos em um tempo finito. Assumimos que as falhas dos processos nunca particionam a rede de comunicação.

Uma computação distribuída pode ser representada por um diagrama espaço-tempo (Figura 2.1). As linhas do diagrama representam a execução dos processos ao longo do tempo e progridem da esquerda para a direita. Uma aresta liga o evento de envio ao evento de entrega de uma mensagem.

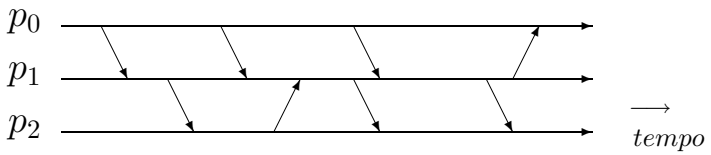


Figura 2.1: Diagrama espaço-tempo para uma computação distribuída

2.2 Processos

Uma aplicação que utiliza um protocolo de *checkpointing* deve implementar em cada processo, uma camada responsável pelo armazenamento de *checkpoints*. Uma estrutura geral de um processo que permite a implementação de qualquer protocolo de *checkpointing* abordado nesta tese é apresentada na Figura 2.2. A aplicação implementa primitivas de envio e entrega de mensagens para a comunicação entre processos e pode requisitar por salvar o estado do processo de tempos em tempos e/ou após um processamento relevante à execução da computação. A camada de *checkpointing* grava os *checkpoints* após uma requisição da aplicação ou de acordo com predicados de um protocolo de *checkpointing*. O sub-sistema de *checkpointing* intercepta as mensagens enviadas e recebidas pela aplicação podendo inserir informações adicionais às mensagens (informações de controle) ou gerar novas mensagens (mensagens de controle). A camada de rede é responsável pelo envio e recepção de mensagens por meio dos canais de comunicação.

A execução de um processo p_i é modelada como uma seqüência possivelmente infinita de eventos $(e_i^0, e_i^1, \dots)$, onde e_i^α representa o α-ésimo evento executado por p_i. Os eventos de um processo podem ser classificados como internos ou de comunicação. Os

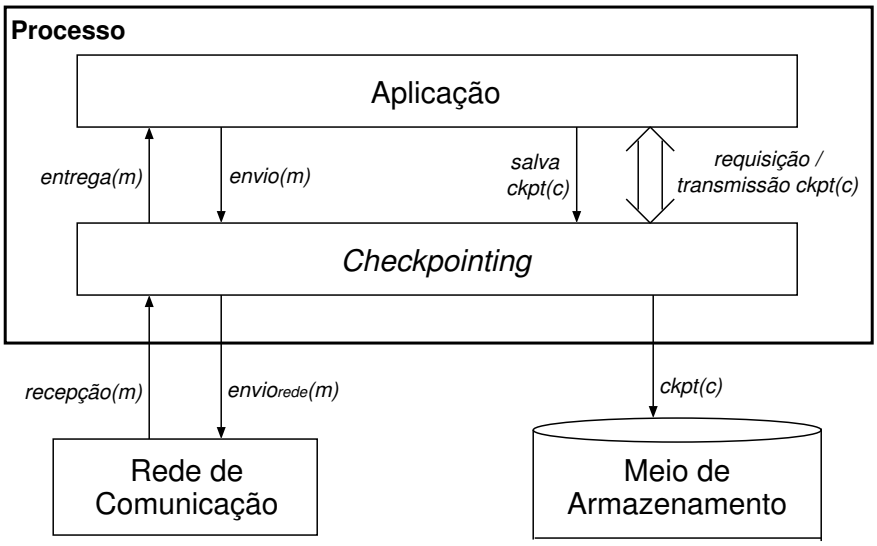


Figura 2.2: Estrutura de um processo

eventos internos são as ações executadas espontaneamente pelo processo e os eventos de comunicação são os eventos de envio e entrega de mensagens.

2.3 Precedência Causal

Em um sistema distribuído, nem sempre é possível determinar entre dois eventos, qual ocorreu primeiro. A verificação de tempo para cada evento por relógios reais não é possível pela dificuldade de sincronizar perfeitamente o tempo preciso dos relógios. Nesta seção, mostramos como é possível definir a ordenação de eventos em uma aplicação distribuída sem o uso de relógios físicos. Este conceito é também utilizado para determinar a ordenação de *checkpoints* que é fundamental para apresentar a noção de estado consistente de um sistema distribuído.

2.3.1 Precedência Causal entre Eventos

A relação de precedência causal foi proposta por Lamport com o objetivo de definir a ordenação parcial dos eventos em uma aplicação distribuída. [21].

Definição 1 Precedência causal—*Um evento* e_a^α *precede um evento* e_b^β ($e_a^\alpha \rightarrow e_b^\beta$) *se*

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

(ii) existe uma mensagem m *que foi enviada em* e_a^α *e recebida em* e_b^β, *ou*

(iii) existe um evento e_c^γ *tal que* $e_a^\alpha \rightarrow e_c^\gamma \wedge e_c^\gamma \rightarrow e_b^\beta$.

A relação de precedência causal entre eventos é ilustrada pela Figura 2.3. Pela Definição 1 (i), o evento e_0^0 precede causalmente o evento e_0^1 ($e_0^0 \rightarrow e_0^1$) pois são eventos do mesmo processo p_0 e e_0^0 ocorre imediatamente antes de e_0^1. Pela Definição 1 (ii), podemos concluir que $e_0^1 \rightarrow e_1^1$ pois m foi enviada em e_0^1 e recebida em e_1^1. A Definição 1 (iii) garante a existência de precedência causal entre $e_0^0 \rightarrow e_1^1$ pois $e_0^0 \rightarrow e_0^1$ e $e_0^1 \rightarrow e_1^1$.

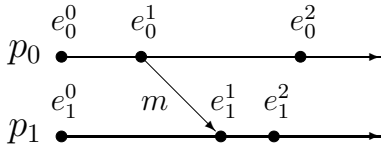


Figura 2.3: Precedência causal entre eventos

Quando não existe precedência causal entre dois eventos e e e', dizemos que esses eventos são *concorrentes* ($e \parallel e'$). Um evento e não precede causalmente ele mesmo, portanto $e \parallel e$. Na Figura 2.3, podemos observar que $e_0^2 \parallel e_1^1$ e $e_0^2 \parallel e_1^2$.

A relação de precedência causal entre eventos dos processos pode ser capturada com o uso de *relógios lógicos*[1] (RL), um mecanismo proposto por Lamport que independe do relógio físico dos processos [21]. Os relógios lógicos indicam quando um evento ocorreu antes de outro na computação distribuída. Em um modelo de sistema que utiliza relógios lógicos, cada processo possui um contador que atribui um número para os eventos do processo e é incrementado entre quaisquer dois eventos consecutivos. O número atribuído indica o tempo lógico em que cada evento ocorreu.

Definição 2 Relógio Lógico—*Um relógio lógico* $\mathtt{RL}_i$ *de um processo* p_i *é uma função que atribui um valor* $\mathtt{RL}_i(e)$ *para todo evento* e *de* p_i *respeitando as seguintes regras:*

(i) $\mathtt{RL}_i(e) < \mathtt{RL}_i(e')$ *se* e *e* e' *são eventos do mesmo processo* p_i *e* e *ocorreu antes de* e' *ou;*

(ii) $\mathtt{RL}_i(e) < \mathtt{RL}_j(e')$ *se* e *é o evento de envio de uma mensagem* m *de* p_i *e* e' *é o evento de recepção de* m *em* p_j.

O cenário da Figura 2.4 ilustra o uso de relógios lógicos implementados por números inteiros. Todos os processos possuem um relógio lógico, inicialmente igual a zero. Os relógios lógicos vão sendo incrementados de acordo com as regras (i) ou (ii) da Definição 2.

O mecanismo de relógios lógicos tem uma relação direta com precedência causal, ou seja, para dois eventos e e e', se $e \rightarrow e'$, então $\mathtt{RL}(e) < \mathtt{RL}(e')$.

[1]Em inglês: *logical clocks*

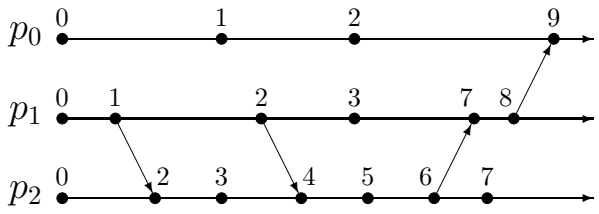


Figura 2.4: Relógios lógicos

Propriedade 1 $e \rightarrow e' \Rightarrow \mathtt{RL}(e) < \mathtt{RL}(e')$

A comparação entre relógios lógicos de dois eventos não implica necessariamente na existência da relação de precedência causal entre esses eventos ($\mathtt{RL}(e) < \mathtt{RL}(e') \not\Rightarrow e \rightarrow e'$). Por exemplo, na Figura 2.4, $\mathtt{RL}(e_1^3)$ é igual a 3 e $\mathtt{RL}(e_2^4)$ é igual a 5, o que implica que $\mathtt{RL}(e_1^3) < \mathtt{RL}(e_2^4)$. Porém $e_1^3 \not\rightarrow e_2^4$.

Para capturar completamente as precedências causais é necessário acrescentar para cada evento, informação sobre os relógios lógicos de outros eventos dos demais processos. Estas informações são mantidas e propagadas pelos processos por meio de vetores, chamados de *vetores de relógios*[2] (VR), de tamanho n, onde n é o número de processos do sistema distribuído. A entrada $\mathtt{VR}_i[i]$ do processo p_i representa o relógio lógico de p_i e é incrementada a cada novo evento local. As demais entradas são atualizadas no momento da recepção de uma mensagem m efetuando-se a operação de máximo para cada entrada do vetor recebido na mensagem e do vetor de p_i: $\mathtt{VR}_i[j] = \mathtt{max}(\mathtt{VR}_i[j], m.\mathtt{VR}[j])$, para todo $j \neq i$. A Figura 2.5 apresenta o uso de vetores de relógios para o mesmo padrão de mensagens da Figura 2.4.

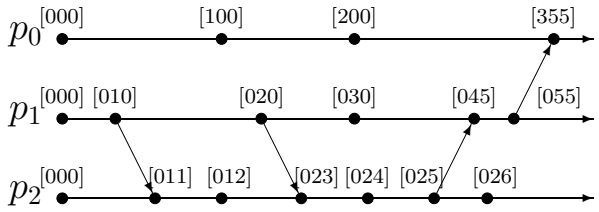


Figura 2.5: Vetores de relógios

Ao compararmos os relógios vetoriais de dois eventos e e e', é possível afirmar se $e \rightarrow e'$ ou $e' \rightarrow e$ ou ainda se $e \mid\mid e'$. A relação entre vetores de relógios e precedências causais é dada por:

[2]Em inglês: *vector clock*

Propriedade 2 $e \rightarrow e' \quad \Leftrightarrow \quad (e \neq e)' \wedge (\forall i : \ \mathtt{VR}(e)[i] \leq \mathtt{VR}(e')[i])$

Propriedade 3 $e_i \rightarrow e_j \Leftrightarrow \begin{cases} \mathtt{VR}(e_i)[i] \leq \mathtt{VR}(e_j)[i], & (i \neq j) \\ \mathtt{VR}(e_i)[i] < \mathtt{VR}(e_j)[i], & (i = j) \end{cases}$

2.3.2 Precedência Causal entre *Checkpoints*

Um *checkpoint* é um evento interno que armazena o estado do processo e pode ser utilizado como ponto de recuperação na ocorrência de falhas. Os *checkpoints* podem ser salvos em memória estável (disco em um servidor) ou não-estável (memória local ou disco local) de acordo com o protocolo de *checkpointing* e todo *checkpoint* salvo em memória não-estável pode ser transferido para memória estável em algum momento da computação. O custo de armazenamento em memória estável é mais alto comparado ao custo de armazenamento em memória não-estável, porém, apenas os *checkpoints* em memória estável representam pontos recuperáveis pelo processo falho.

Consideramos que todo processo armazena um *checkpoint* inicial imediatamente antes de iniciar sua computação e um *checkpoint* final imediatamente antes do término da computação. Utilizamos c_i^ι para denotar o ι-ésimo *checkpoint* gravado pelo processo p_i, sendo que $\iota = 0$ representa o *checkpoint* inicial. Um intervalo entre *checkpoints* é o conjunto de eventos ocorridos entre dois *checkpoints* consecutivos do mesmo processo. Intervalos de *checkpoint* podem ser rotulados de duas maneiras: à esquerda e à direita. Nesta tese, utilizamos a rotulação à esquerda, ou seja, o intervalo entre c_i^α e $c_i^{\alpha+1}$ é rotulado como I^α. O conjunto de todos os *checkpoints* e mensagens trocadas entre processos de uma computação distribuída forma um *padrão de checkpoints e mensagens*[3].

Dizemos que c_a^α precede c_b^β ($c_a^\alpha \rightarrow c_b^\beta$) se o evento que originou c_a^α precede o evento que originou c_b^β. Um *checkpoint* c_a^α precede diretamente um *checkpoint* c_b^β se o processo p_a enviou uma mensagem m para p_b após o armazenamento de c_a^α e p_b recebeu m antes de salvar c_b^β. Uma precedência transitiva de c_a^α para c_b^β é formada por uma seqüência de mensagens iniciada por p_a após c_a^α e terminada em p_b antes de c_b^β.

A Figura 2.6 ilustra um diagrama espaço-tempo para três processos e os *checkpoints* de cada processo são representados por quadrados preenchidos. Nesta figura, temos que o *checkpoint* c_0^α precede diretamente o *checkpoint* $c_1^{\beta+1}$ ($c_0^\alpha \ \rightarrow \ c_1^{\beta+1}$) e precede transitivamente o *checkpoint* $c_2^{\gamma+1}$ ($c_0^\alpha \rightarrow c_2^{\gamma+1}$).

As precedências causais entre *checkpoints* podem ser rastreadas durante uma computação distribuída por meio do uso de *vetores de dependências*[4] (VD). O mecanismo de vetores de dependências é semelhante ao de vetores de relógios, porém os vetores de dependências armazenam informações sobre as precedências causais entre *checkpoints*.

[3]Em inglês: *checkpoint and communication pattern*
[4]Em inglês: *dependency vector*

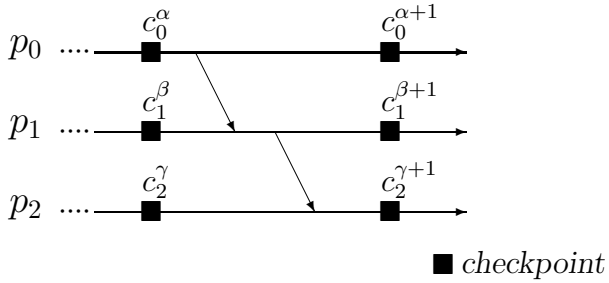


Figura 2.6: Precedência causal entre *checkpoints*

Cada processo p_i mantém e propaga um vetor de dependências $\texttt{VD}_i$ com n entradas, todas iniciadas com 0 e a entrada $\texttt{VD}_i[i]$ é incrementada imediatamente após a retirada de um *checkpoint*, incluindo o *checkpoint* inicial. A entrada $\texttt{VD}_i[i]$ indica o intervalo corrente de p_i e as outras entradas $\texttt{VD}_i[j]$, $j \neq i$, indicam o índice do último intervalo de p_j que p_i conhece.

Ao enviar uma mensagem m, o vetor de dependências $\texttt{VD}_i$ do processo p_i é agregado à mensagem, denotado por $m.\texttt{VD}$. Quando p_i recebe uma mensagem m, p_i atualiza o vetor $\texttt{VD}_i$ fazendo uma operação de máximo para cada entrada do vetor recebido na mensagem e do vetor de p_i: $\texttt{VD}_i[j] = \texttt{max}(\texttt{VD}_i[j], m.\texttt{VD}[j])$, para todos $j \neq i$. A Figura 2.7 ilustra uma computação distribuída com vetores de dependências.

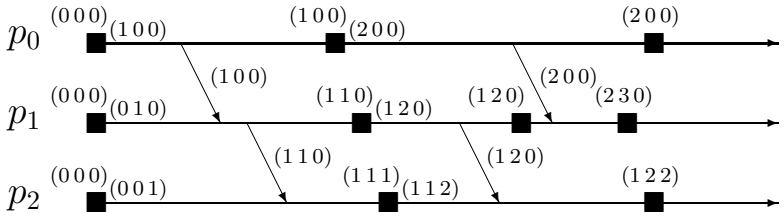


Figura 2.7: Vetores de dependências

Dado c_a^{α} o α-ésimo *checkpoint* do processo p_a e c_b^{β} o β-ésimo *checkpoint* do processo p_b, temos as seguintes propriedades para vetores de dependência:

Propriedade 4 $c_a^{\alpha} \rightarrow c_b^{\beta} \;\Rightarrow\; \texttt{VD}(c_b^{\beta})[a] \geq \alpha + 1$

Propriedade 5 $\texttt{VD}(c_b^{\beta})[a] = \alpha \;\Rightarrow \begin{cases} c_a^{\alpha-1} & \rightarrow & c_b^{\beta} \\ c_a^{\alpha} & \not\rightarrow & c_b^{\beta} \end{cases}$

Utilizando as Propriedades 4 e 5 do vetor de dependências é possível rastrear todas as dependências entre os *checkpoints* obtidos pelos processos da aplicação.

2.4 *Zigzag Paths*

O conceito de *zigzag path* introduzido por Netzer e Xu [27] é uma generalização do conceito de precedência causal introduzido por Lamport [21].

Definição 3 Zigzag Path—*Existe uma zigzag path a partir de um checkpoint* c_a^α *a um checkpoint* c_b^β *se existe uma seqüência de mensagens* $m_1, m_2, \ldots, m_k$ $(k \geq 1)$ *tal que:*

(i) m_1 *é enviada pelo processo* p_a *após* c_a^α*,*

(ii) se m_i $(1 \leq i < k)$ *é recebida pelo processo* p_j*, então* m_{i+1} *é enviada por* p_j *no mesmo ou no intervalo de checkpoints posterior à recepção de* m_i*, e*

(iii) m_k *é recebida pelo processo* p_b *antes de* c_b^β*.*

Existem dois tipos de *zigzag paths*: causais e não-causais. Uma *zigzag path* é causal se a recepção de toda mensagem m_i, $1 \leq i < k$, ocorre sempre antes do envio de m_{i+1}. Uma *zigzag path* é não-causal se a recepção de alguma mensagem m_i, $1 \leq i < k$, ocorre após o envio de m_{i+1}. Na Figura 2.8, existe uma *zigzag path* não-causal de c_0^α a $c_2^{\beta+1}$.

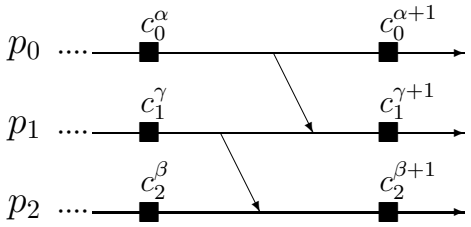
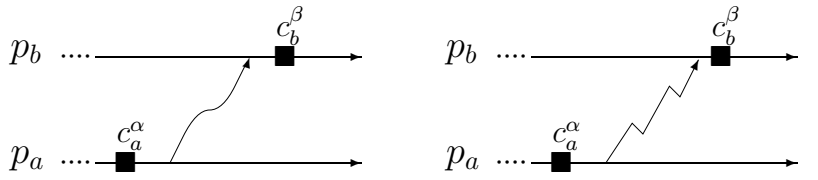


Figura 2.8: *Zigzag path* não-causal entre *checkpoints*

Uma *zigzag path* causal a partir de c_a^α a c_b^β, formada por uma seqüência de mensagens, equivale à precedência causal $(c_a^\alpha \rightarrow c_b^\beta)$ e é representada nesta tese por uma seta semelhante à ilustrada na Figura 2.9 (a). Similarmente, a Figura 2.9 (b) representa uma *zigzag path* não-causal entre c_a^α e c_b^β e, portanto, pelo menos uma das mensagens m_i que compõem a *zigzag path* foi recebida após o envio da mensagem subseqüente m_{i+1}.

Quando uma *zigzag path* conecta um *checkpoint* a si próprio, temos uma *zigzag cycle* (*Z-cycle*). Este tipo especial de *zigzag path* impede que o *checkpoint* envolvido faça parte



(a) *Zigzag path* causal (b) *Zigzag path* não-causal

Figura 2.9: *Zigzag paths*

de qualquer *checkpoint* global consistente (Seção 2.5). Um *checkpoint* que não faz parte de nenhum *checkpoint* global consistente é chamado de *checkpoint* inútil. Na Figura 2.10, o *checkpoint* c_b^β é inútil pois existe uma *zigzag path* de c_b^β a $c_a^{\alpha+1}$ e outra de c_a^α a c_b^β que forma um z-cycle que envolve c_b^β.

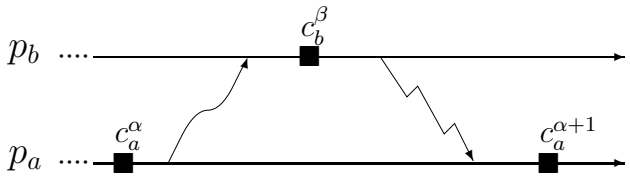


Figura 2.10: *Zigzag cycle*

2.4.1 Z-precedência

A relação de z-precedência entre *checkpoints* equivale ao conceito de *zigzag path*, mas tem sua definição baseada em precedências causais e não em seqüência de mensagens [16]. Nesta tese, utilizamos o símbolo $\rightsquigarrow$ para indicar a existência de uma *zigzag path* ou de uma z-precedência entre dois *checkpoints*.

Definição 4 Z-Precedência—*Um checkpoint c_a^α z-precede um checkpoint c_b^β ($c_a^\alpha \rightsquigarrow c_b^\beta$) se, e somente se,*

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

(ii) $\exists c_c^\gamma : (c_a^\alpha \rightsquigarrow c_c^\gamma) \wedge (c_c^{\gamma-1} \rightsquigarrow c_b^\beta)$

Na Figura 2.8, $c_0^\alpha \rightsquigarrow c_1^{\gamma+1}$ pois $c_0^\alpha \rightarrow c_1^{\gamma+1}$ (Definição 4 (i)). Pela Definição 4 (ii), $c_0^\alpha \rightsquigarrow c_2^{\beta+1}$ pois $(c_0^\alpha \rightsquigarrow c_1^{\gamma+1}) \wedge (c_1^\gamma \rightsquigarrow c_2^{\beta+1})$.

2.4.2 Z-dependência

Cao e Singhal introduzem o conceito de z-dependência para expressar uma relação de dependência entre processos em dois intervalos de *checkpoints* [9].

Definição 5 Z-Dependência—*Se um processo p_a envia uma mensagem para o processo p_b durante o seu α-ésimo intervalo de checkpoints I_a^α e p_b recebe a mensagem durante o seu β-ésimo intervalo de checkpoints I_b^β, então p_b z-depende de p_a durante I_b^β e I_a^α ($p_a \prec_\beta^\alpha p_b$).*

A z-dependência entre processos pode ser construída também de forma transitiva. Se $p_a \prec_\gamma^\alpha p_c$ e $p_c \prec_\beta^\gamma p_b$, então p_b z-depende transitivamente de p_a durante I_b^β e I_a^α. Esta relação é denotada por $p_a \overset{*}{\prec}{}_\beta^\alpha p_b$.

Definição 6 Z-Dependência Transitiva—*Um processo p_b z-depende transitivamente do processo p_a durante I_b^β e I_a^α ($p_a \overset{*}{\prec}{}_\beta^\alpha p_b$) se:*

(i) $p_a \prec_\beta^\alpha p_b$

(ii) $p_a \overset{*}{\prec}{}_\gamma^\alpha p_c \wedge p_c \overset{*}{\prec}{}_\beta^\gamma p_b$

A Figura 2.11 ilustra a relação de z-dependência entre processos. Notamos que p_1 z-depende diretamente de p_0 durante I_1^γ e I_0^α ($p_0 \prec_\gamma^\alpha p_1$). Pela Definição 6 ($i$), se $p_0 \prec_\gamma^\alpha p_1$, então, $p_0 \overset{*}{\prec}{}_\gamma^\alpha p_1$. Da mesma forma, $p_1 \prec_\beta^\gamma p_2 \Rightarrow p_1 \overset{*}{\prec}{}_\beta^\gamma p_2$. Pela Definição 6 ($ii$), $p_0 \overset{*}{\prec}{}_\gamma^\alpha p_1 \wedge p_1 \overset{*}{\prec}{}_\beta^\gamma p_2$ e portanto $p_0 \overset{*}{\prec}{}_\beta^\alpha p_2$.

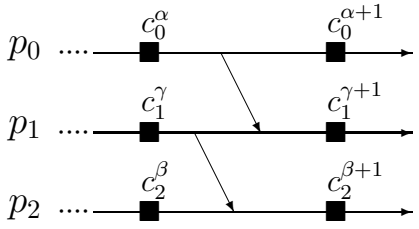


Figura 2.11: Z-dependência entre processos

2.4.3 Relação entre Z-precedência e Z-dependência

O conceito de z-precedência é baseado na noção de causalidade e foi proposto para relacionar os *checkpoints*. Cao e Singhal introduzem o conceito de z-dependência para definir a relação de dependência entre processos e afirmam que é impossível utilizar *zigzag paths* para mapear dependências dos protocolos síncronos [10]. Como visto na Seção 2.4.1, a relação de z-precedência entre *checkpoints* equivale ao conceito de *zigzag path*. Nesta seção, provamos que o conceito de z-dependência está contido no conceito de z-precedência sendo a relação de z-precedência mais abrangente e portanto pode ser utilizada também para a teoria dos protocolos síncronos.

Teorema 1 *Se p_b z-depende de p_a durante I_b^β e I_a^α então, c_a^α z-precede $c_b^{\beta+1}$, isto é,*

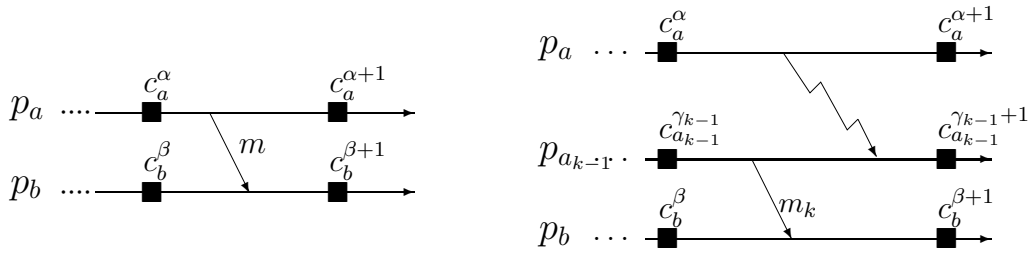
$$p_a \overset{*}{\prec}{}_\beta^\alpha\, p_b \Rightarrow c_a^\alpha \rightsquigarrow c_b^{\beta+1}$$

Prova: A relação $p_a \overset{*}{\prec}{}_\beta^\alpha\, p_b$ pode ser expressa por um conjunto de relações de z-dependências diretas:

$$p_{a_0} \prec_{\gamma_1}^{\gamma_0} p_{a_1} \ \wedge\ p_{a_1} \prec_{\gamma_2}^{\gamma_1} p_{a_2} \ \wedge\ \ldots\ \wedge\ p_{a_{k-1}} \prec_{\gamma_k}^{\gamma_{k-1}} p_{a_k}$$

onde $p_{a_0} = p_a$, $\gamma_0 = \alpha$, $p_{a_k} = p_b$ e $\gamma_k = \beta$. A prova será feita por indução no número de z-dependências diretas que forma a z-dependência entre os processos p_a e p_b.

Base ($k = 1$): Pela Definição 5, se $p_a \prec_\beta^\alpha p_b$, então existe uma mensagem m que foi enviada por p_a durante I_a^α e recebida por p_b em I_b^β (Figura 2.12 (a)). Neste caso, m foi enviada após c_a^α e recebida antes de $c_b^{\beta+1}$ e portanto, $c_a^\alpha \rightarrow c_b^{\beta+1}$. Pela Definição 4, se $c_a^\alpha \rightarrow c_b^{\beta+1}$, então $c_a^\alpha \rightsquigarrow c_b^{\beta+1}$. Portanto, $p_a \prec_\beta^\alpha p_b \Rightarrow c_a^\alpha \rightarrow c_b^{\beta+1} \Rightarrow c_a^\alpha \rightsquigarrow c_b^{\beta+1}$.



(a) $p_a \prec_\beta^\alpha p_b \Rightarrow c_a^\alpha \rightarrow c_b^{\beta+1}$ (b) $p_a \overset{*}{\prec}{}_\beta^\alpha\, p_b \Rightarrow c_a^\alpha \rightsquigarrow c_b^{\beta+1}$

Figura 2.12: Cenário para a prova do Teorema 1

Passo ($k > 1$): Suponha que se $p_a \overset{*}{\prec}{}_{\gamma_{k-1}}^\alpha\, p_{a_{k-1}}$, então $c_a^\alpha \rightsquigarrow c_{a_{k-1}}^{\gamma_{k-1}+1}$. Sabemos que existe uma z-dependência direta que indica que p_b z-depende de $p_{a_{k-1}}$ durante I_b^β e

$I_{a_{k-1}}^{\gamma_{k-1}}$ $(p_{a_{k-1}} \prec_{\beta}^{\gamma_{k-1}} p_b)$ e portanto, $c_{a_{k-1}}^{\gamma_{k-1}} \rightarrow c_b^{\beta+1}$. Sabemos também pela Definição 4 que $c_{a_{k-1}}^{\gamma_{k-1}} \rightsquigarrow c_b^{\beta+1}$. Além disso, se $c_a^{\alpha} \rightsquigarrow c_{a_{k-1}}^{\gamma_{k-1}+1} \wedge c_{a_{k-1}}^{\gamma_{k-1}} \rightsquigarrow c_b^{\beta+1}$ então $c_a^{\alpha} \rightsquigarrow c_b^{\beta+1}$. Assim, podemos concluir que $p_a \overset{* \alpha}{\prec}_{\beta} p_b \Rightarrow c_a^{\alpha} \rightsquigarrow c_b^{\beta+1}$. □

A existência da relação de z-precedência não implica na existência da relação de z-dependência. Na Figura 2.13, $c_a^{\alpha} \rightsquigarrow c_a^{\alpha+1}$ e $c_a^{\alpha+1} \rightsquigarrow c_b^{\beta+1}$ e portanto, $c_a^{\alpha} \rightsquigarrow c_b^{\beta+1}$. Porém, pela Definição 5 não existe relação de z-dependência entre c_a^{α} e $c_b^{\beta+1}$ pois a mensagem m foi enviada no intervalo de *checkpoints* $I_a^{\alpha+1}$.

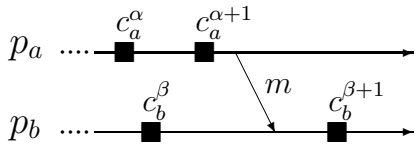


Figura 2.13: Existência de z-precedência não implica em z-dependência

2.5 *Checkpoint* Global Consistente

Um *checkpoint* global consistente é um estado do sistema que pode ser restabelecido de forma segura não permitindo que o evento de entrega de uma mensagem ocorra antes do evento de envio da mesma. Um *checkpoint* global é formado por um conjunto de *checkpoints*, um por processo, e é consistente se não existe relação de causalidade entre os *checkpoints* do conjunto.

Definição 7 *Checkpoint* global consistente—*Um checkpoint global* $\Sigma = \{c_0^{\iota 0}, \ldots, c_{n-1}^{\iota n-1}\}$ *é consistente se, e somente se,* $\forall i, j : 0 \leq i, j < n : c_i^{\iota i} \not\rightarrow c_j^{\iota j}$.

Informalmente, um estado global em um diagrama espaço-tempo é consistente se, ao traçar uma linha que une os *checkpoints* do conjunto, nenhuma aresta tem início no lado direito e termina no lado esquerdo dessa linha. Na Figura 2.14, a linha Σ forma um *checkpoint* global inconsistente pois o primeiro *checkpoint* de p_0 precede o segundo *checkpoint* de p_1. A linha Σ' mostra um *checkpoint* global consistente.

Dizemos que dois *checkpoints* são consistentes se eles podem fazer parte de um mesmo *checkpoint* global consistente. As condições necessárias e suficientes para determinar a consistência entre dois *checkpoints* podem ser definidas por meio de *zigzag paths* [27].

Definição 8 *Checkpoints* consistentes—*Os checkpoints* c_a^{α} *e* c_b^{β} *são consistentes se, e somente se,* $c_a^{\alpha} \not\rightsquigarrow c_b^{\beta} \wedge c_b^{\beta} \not\rightsquigarrow c_a^{\alpha}$.

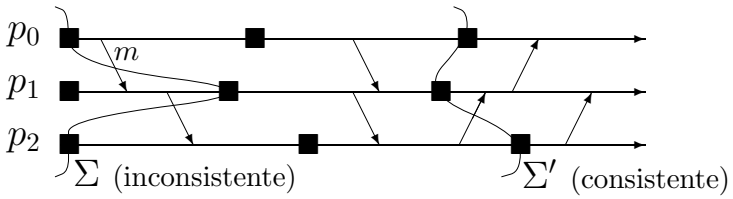


Figura 2.14: Consistência em *checkpoints* globais

Duas operações importantes sobre *checkpoints* globais são de união e de intersecção. A união de dois *checkpoints* globais resulta em um *checkpoint* global constituído pelos *checkpoints* mais recentes de cada processo, que está presente em um dos dois *checkpoints* globais. Analogamente, a intersecção de dois *checkpoints* globais é dada pelo *checkpoint* menos recente em cada processo (Figura 2.15).

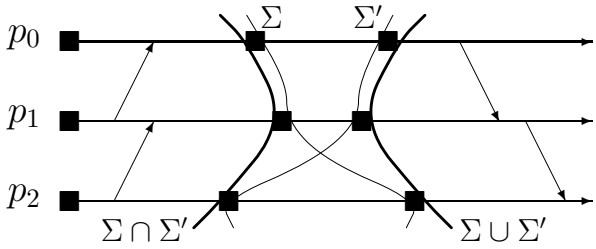


Figura 2.15: União e intersecção de *checkpoints* globais

Definição 9 União de *Checkpoints* **Globais**—*A união de dois checkpoints globais* Σ *e* Σ' *é dada por:*

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

Definição 10 Intersecção de *Checkpoints* **Globais**—*A intersecção de dois checkpoints globais* Σ *e* Σ' *é dada por:*

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

A união e a intersecção de dois *checkpoints* globais consistentes resulta sempre em outro *checkpoint* global consistente [26] .

2.6 Sumário

A necessidade de determinar a ordem em que os eventos ocorrem sem o uso de relógios físicos ou um relógio global em um sistema distribuído, fez com que surgisse a idéia de tempo lógico [21]. Pelo conceito de precedência causal é possível determinar se um evento ocorreu antes de um outro evento ou se eles são concorrentes (não existe dependência entre os eventos). Esse conceito é de suma importância na utilização de *checkpoints*, que são eventos internos de um processo, pois é por meio desta relação que definimos *checkpoint* global consistente, ou seja, um *checkpoint* global que pode ser utilizado como ponto de recuperação na ocorrência de uma falha. Mais tarde, Netzer e Xu introduzem o conceito de *zigzag paths* que determina as condições necessárias e suficientes para que um conjunto de *checkpoints* façam parte de um mesmo *checkpoint* global consistente. É possível relacionar os *checkpoints* de maneira equivalente ao conceito de *zigzag paths* por meio da relação de z-precedência [16].

O conceito de z-dependência foi proposto por Cao e Singhal para relacionar dois processos em seus respectivos intervalos de *checkpoint* [9]. Neste capítulo, provamos que o conceito de z-dependência está contido no conceito de z-precedência. Nos próximos capítulos, mostraremos que os conceitos definidos e adotados pelos protocolos quase-síncronos podem ser utilizados pelos protocolos síncronos.

Capítulo 3

Checkpointing

Os protocolos de *checkpointing* são responsáveis pela seleção dos *checkpoints* salvos em memória estável. Em caso de falha, o sistema deve ser restabelecido a partir de um estado consistente. Além disso, é necessário um protocolo de coleta de lixo para remover os *checkpoints* que não serão utilizados por nenhum protocolo de recuperação liberando espaço de armazenamento.

Na literatura, existem três abordagens para a seleção de *checkpoints*: assíncrona, quase-síncrona e síncrona [25]. Nos protocolos assíncronos, os *checkpoints* são salvos arbitrariamente e não há garantia da formação de *checkpoints* globais consistentes.

Na abordagem quase-síncrona, informações de controle são propagadas com as mensagens da aplicação de maneira que *checkpoints* adicionais podem ser induzidos pelo protocolo para garantir a formação de *checkpoints* globais consistentes. A recuperação e a coleta de lixo devem ser implementadas separadamente.

Os protocolos síncronos utilizam mensagens de controle para sincronizar as atividades de seleção de *checkpoints* garantindo a existência de um *checkpoint* global consistente para cada *checkpoint* requisitado pela aplicação. Um protocolo síncrono que salva apenas um número minimal de *checkpoints* a cada execução é chamado de minimal. Os protocolos não-bloqueantes podem garantir um número minimal de *checkpoints* salvos em memória estável quando utilizam também memória não-estável para o armazenamento de *checkpoints*. A coleta de lixo é normalmente embutida nesses protocolos.

As Seções seguintes estão divididas como a seguir. As Seções 3.1, 3.2 e 3.3, descrevem respectivamente as classes de protocolos de *checkpointing* assíncrona, quase-síncrona e síncrona, sendo que, para esta última, é dada a ênfase em protocolos minimais. Em seguida, discutimos como é possível restabelecer o sistema após a ocorrência de uma falha (Seção 3.4). Por último, descrevemos alguns mecanismos de coleta de lixo (Seção 3.5).

3.1 Protocolos Assíncronos

Na abordagem assíncrona, os processos têm autonomia na escolha de seus *checkpoints*, (cada processo salva um *checkpoint* quando for mais conveniente), porém alguns desses *checkpoints*, chamados de *checkpoints* inúteis, podem não fazer parte de nenhum *checkpoint* global consistente. Além disso, não há garantias de que seja possível formar um *checkpoint* global consistente a partir dos *checkpoints* salvos. Portanto, pode ocorrer o efeito dominó, ou seja, a aplicação pode ser obrigada a retornar ao estado inicial em caso de falha [32]. Os protocolos desta classe não necessitam de nenhum recurso adicional ao armazenamento de *checkpoints*, como propagação de informações de controle, porém devem ter mecanismos separados para verificar a formação de *checkpoints* globais consistentes que são utilizados em caso de falha e para a coleta de lixo. A Figura 3.1 ilustra um exemplo de padrão de *checkpoints* e mensagens gerado por um protocolo assíncrono. Apenas os *checkpoints* requisitados pela aplicação (*checkpoints* básicos) são salvos em memória estável. Neste cenário, o *checkpoint* global consistente mais recente é representado por Σ. Portanto, na ocorrência de uma falha, os processos deverão retroceder e se estabelecer a partir desse ponto, ou seja, os processos, exceto p_3, deverão retroceder ao início da aplicação. Este mesmo cenário será utilizado para exemplificar protocolos de *checkpointing* de outras classes descritos neste capítulo. A classe de protocolos assíncronos não garante a formação de *checkpoints* globais consistentes e portanto, não será discutida em maiores detalhes.

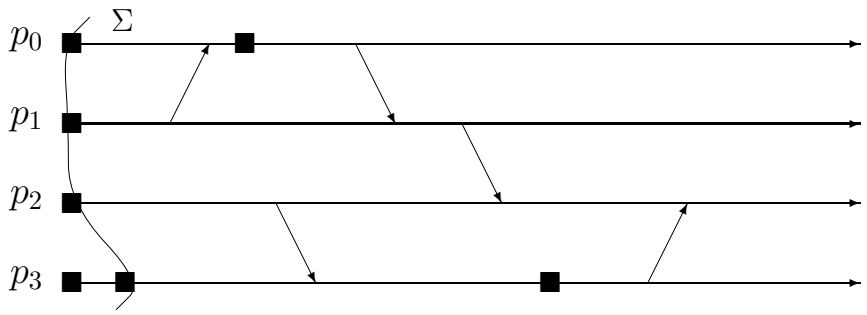


Figura 3.1: Exemplo do padrão gerado por um protocolo assíncrono

3.2 Protocolos Quase-Síncronos

Os protocolos quase-síncronos (ou induzidos por comunicação) propostos na literatura reduzem ou eliminam o número de *checkpoints* inúteis na tentativa de garantir que cada

checkpoint faça parte de um *checkpoint* global consistente. Este objetivo foi atingido induzindo-se o armazenamento de *checkpoints* adicionais aos *checkpoints* do protocolo assíncrono. Nesta abordagem, os processos salvam seus *checkpoints* livremente, chamados *checkpoints* básicos e podem ser induzidos a armazenarem *checkpoints* adicionais, chamados *checkpoints* forçados, segundo predicados avaliados sobre informações de controle propagadas pelas mensagens da aplicação [14, 25]. Posteriormente, *checkpoints* globais consistentes são formados a partir dos *checkpoints* selecionados e a coleta de lixo é realizada. Esta abordagem apresenta um compromisso entre a autonomia dos processos para a escolha dos *checkpoints* e as garantias oferecidas para a formação de *checkpoints* globais consistentes.

Os protocolos quase-síncronos podem ser classificados de acordo com os conceitos de *zigzag path* e *zigzag cycle* [24, 25]. A primeira classe *Strictly Z-Path Free* (SZPF) é a mais restritiva e impede a formação de *zigzag paths* não-causais entre *checkpoints*. A segunda classe *Z-Path Free* (ZPF) impede a existência de *zigzag path* não-causais não duplicadas causalmente[1]. A terceira classe *Z-Cycle Free* (ZCF) não permite a existência de *zigzag cycle* garantindo assim, a ausência de *checkpoints* inúteis. A Classe *Partially Z-Cycle Free* (PZCF) é a menos restritiva e apenas tenta quebrar algumas *Z-cycles*, aumentando as chances de existir, mas sem garantir, os *checkpoints* globais consistentes. A hierarquia entre essas classes é apresentada pela Figura 3.2 e detalhada a seguir.

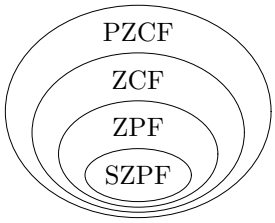


Figura 3.2: Classes de protocolos quase-síncronos

3.2.1 Classe SZPF (*Strictly Z-Path Free*)

A Classe SZPF gera um padrão de *checkpoints* e mensagens apenas com *zigzag paths* causais. Os protocolos desta classe garantem a inexistência de *checkpoints* inúteis e permitem que todas as dependências entre *checkpoints* possam ser rastreadas utilizando vetores de dependências. Estes protocolos são simples e possuem baixo *overhead* nas

[1]Em inglês: causally doubled

mensagens da aplicação, porém apresentam um número excessivo de *checkpoints* forçados, podendo degradar o desempenho do sistema.

O protocolo CAS (*Checkpoint-After-Send*) é um exemplo de protocolo pertencente à classe SZPF [45]. Este protocolo induz um *checkpoint* forçado após cada mensagem enviada. O Protocolo 3.1 descreve o protocolo CAS.

Protocolo 3.1 CAS

Início:
salvaCheckpoint()

Envio da mensagem (m) da aplic. para p_k:
envia a mensagem (m) da aplicação
salvaCheckpoint()

Recepção da mensagem (m) da aplic. de p_k:
entrega a mensagem m para a aplicação

salvaCheckpoint()
grava um checkpoint em dispositivo estável

A Figura 3.3 ilustra um padrão de *checkpoints* e mensagens gerado pelo protocolo CAS. Neste cenário, o *checkpoint* global consistente mais recente é representado por Σ.

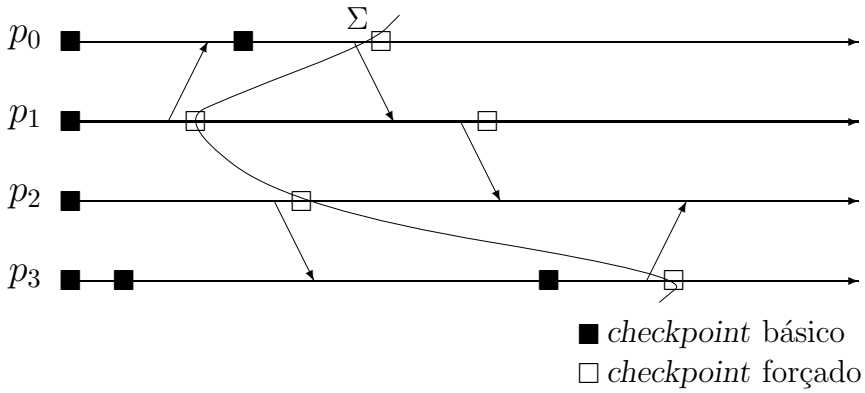


Figura 3.3: Exemplo de padrão gerado pelo protocolo CAS

Outros protocolos da Classe SZPF identificados por Wang são: CASBR (*Checkpoint-After-Send-Before-Receive*); CBR (*Checkpoint-Before-Receive*); NRAS (*No-Receive-After-Send*) [45]. A descrição destes protocolos pode ser encontrada nas Seções A.1, A.2 e A.3, respectivamente.

3.2.2 Classe ZPF (*Z-Path Free*)

Os protocolos da Classe ZPF garantem que todas as dependências entre *checkpoints* são causais. Neste padrão, se existe uma *zigzag path* não-causal entre c_a^α e c_b^β, então esta

relação está duplicada por um caminho causal ($c_a^\alpha \rightsquigarrow c_b^\beta \Rightarrow c_a^\alpha \rightarrow c_b^\beta$).

Os protocolos desta classe propostos na literatura utilizam vetores de dependências para identificar a ocorrência de uma nova precedência e impedir a existência de *checkpoints* inúteis. Wang identificou uma propriedade interessante presente nos padrões ZPF e SZPF chamada *Rollback-Dependency Trackability* (RDT), que permite, em tempo de execução, rastrear as dependências entre *checkpoints* por meio do uso de vetores de dependências [45].

Notamos na literatura um esforço em tentar diminuir o número de *checkpoints* forçados em busca de uma caracterização minimal dos padrões de *checkpoints* e mensagens que satisfazem a propriedade RDT [1, 2]. Porém, um *checkpoint* forçado salvo em um determinado momento pode diminuir a necessidade de armazenar outros *checkpoints* no futuro. Ou seja, um protocolo que deixa de salvar um *checkpoint* forçado desnecessário em um determinado momento pode ter que salvar mais *checkpoints* forçados no futuro. Com base nesses argumentos, Tsai formaliza a prova de que é impossível desenvolver um protocolo que armazena apenas um número mínimo de *checkpoints* forçados para todos os padrões de *checkpoints* e mensagens possíveis [41].

Um protocolo simples e bastante conhecido da classe ZPF é o protocolo FDAS (*Fixed-Dependency-After-Send*) [45]. Este protocolo utiliza o mecanismo de vetores de dependências e cada processo não modifica o seu vetor após enviar uma mensagem da aplicação durante um intervalo de *checkpoints*. Ou seja, se a mensagem da aplicação enviada por p_i e recebida por p_j carrega uma informação não conhecida por p_j, p_j salva um *checkpoint* forçado imediatamente antes de receber a mensagem. Uma otimização na condição de indução de *checkpoints* forçados foi proposto em [17] e sua descrição é apresentada pelo Protocolo 3.2.

Protocolo 3.2 FDAS

Variáveis do processo:
VD ≡ vetor[0 . . . $n-1$] de inteiros
enviou ≡ booleano
pid ≡ inteiro

Início:
$\forall i$: VD[i] ← 0
salvaCheckpoint()

Envio da mensagem (m) da aplic. para p_k:
enviou ← *verdadeiro*
m.VD ← VD
envia a mensagem (m) da aplicação

Recepção da mensagem (m) da aplic. de p_k:
se (enviou *e* m.VD[k] > VD[k])
  salvaCheckpoint()
$\forall i$: VD[i] ← max(VD[i], m.VD[i])
entrega a mensagem m para a aplicação

salvaCheckpoint()
grava um checkpoint em dispositivo estável
VD[pid] ← VD[pid] + 1
enviou ← *falso*

A Figura 3.4 ilustra um padrão de *checkpoints* e mensagens gerado pelo protocolo FDAS. Neste cenário, o *checkpoint* global consistente mais recente é representado por Σ.

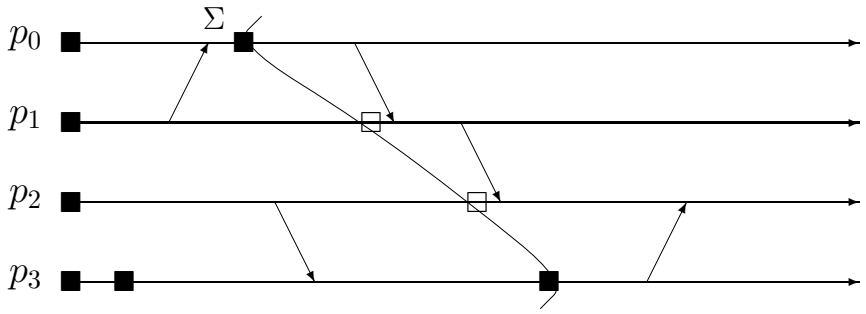


Figura 3.4: Exemplo de padrão gerado pelo protocolo FDAS

O custo associado às mensagens da aplicação é, em geral, a propagação do vetor de dependências. Resultados de simulação mostram que estes protocolos salvam um alto número de *checkpoints* forçados comparados aos protocolos da Classe ZCF [44, 43]. Outros exemplos de protocolos da Classe ZPF são: FDI (*Fixed-Dependency-Interval*) [45]; RDT-Partner [15]; RDT-minimal [18], e são descritos respectivamente nas Seções A.4, A.5 e A.6. O protocolo BHMR (*Baldoni, Helary, Mostefaoui, Raynal*) [4] pertence à esta classe porém seu código não é apresentado, pois possui um comportamento equivalente ao do protocolo RDT-minimal.

3.2.3 Classe ZCF (*Z-Cycle Free*)

Este padrão garante a ausência de *checkpoints* inúteis impedindo a ocorrência de *zigzag cycles*. Assim, todo *checkpoint* salvo por um protocolo ZCF irá fazer parte de algum *checkpoint* global consistente.

Um dos protocolos mais citados desta classe foi proposto por Briatico, Ciuffoletti e Simoncini e é chamado de BCS [6]. Neste protocolo, cada processo mantém e propaga apenas um índice que é incrementado a cada novo *checkpoint*. Um novo *checkpoint* é induzido imediatamente antes da recepção de uma mensagem da aplicação se o índice da mensagem for maior do que o índice mantido pelo processo. Entre os protocolos quase-síncronos que livram o padrão de *checkpoints* e mensagens de *Z-cycles*, o BCS produz o máximo de $n-1$ *checkpoints* forçados para cada *checkpoint* básico. Um mecanismo de otimização que impede a indução de *checkpoints* se nenhuma mensagem foi enviada no mesmo intervalo de *checkpoints* foi apresentado em [42] e este protocolo passou a ser chamado de BCS-Aftersend. O Protocolo 3.3 descreve o protocolo BCS-Aftersend.

Protocolo 3.3 BCS-Aftersend

Variáveis do processo:
ind ≡ inteiro
enviou ≡ booleano
pid ≡ inteiro

Início:
ind ← 0
salvaCheckpoint()

Envio da mensagem (m) da aplic. para p_k:
enviou ← *verdadeiro*
m.ind ← ind
envia a mensagem (m) da aplicação

Recepção da mensagem (m) da aplic. de p_k:
se (m.ind > ind)
se (enviou)
salvaCheckpoint()
ind ← m.ind
entrega a mensagem m para a aplicação

salvaCheckpoint()
grava um checkpoint em dispositivo estável
ind ← ind + 1
enviou ← *falso*

Um exemplo de padrão de *checkpoints* e mensagens gerado por este protocolo é apresentado pelo Figura 3.5. Neste cenário, o *checkpoint* global consistente mais recente é representado por Σ.

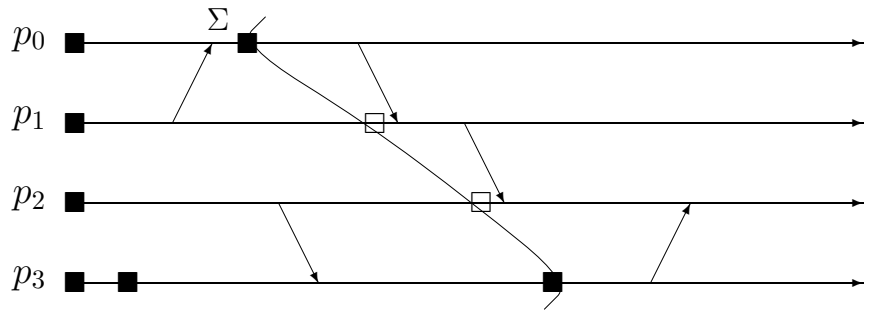


Figura 3.5: Exemplo de padrão gerado pelo protocolo BCS-Aftersend

Outro protocolo ZCF altamente custoso proposto na literatura por Baldoni, Quaglia e Ciciani (BQC), controla a indução de *checkpoints* por meio da propagação de matrizes de relógios [3, 31].

3.2.4 Classe PZCF (*Partially Z-Cycle Free*)

Os protocolos da Classe PZCF não garantem a ausência de *Z-cycles*, porém existe um esforço para que pelo menos uma parte dos *checkpoints* sejam úteis. Normalmente, esses protocolos são simplificações dos protocolos das Classes ZPF e ZCF. Esta é a classe menos restritiva e se aproxima dos protocolos assíncronos pois há dificuldade para a obtenção de

checkpoints globais consistentes e está sujeita ao efeito dominó.

Um exemplo de protocolo desta classe é chamado de WF e foi proposto por Wang e Fuchs [46]. Este protocolo é baseado no protocolo BCS, porém, a indução de *checkpoints* forçados só ocorre para índices múltiplos de um determinado valor γ. Apenas o conjunto de *checkpoints* múltiplos de γ podem garantidamente fazer parte de um *checkpoint* global consistente. O Protocolo 3.4 descreve o protocolo WF.

Protocolo 3.4 WF

Variáveis do processo:
ind ≡ inteiro
γ ≡ inteiro
pid ≡ inteiro

Início:
ind ← 0
γ ← 2
salvaCheckpoint()

Envio da mensagem (m) da aplic. para p_k:
m.ind ← ind
envia a mensagem (m) da aplicação

Recepção da mensagem (m) da aplic. de p_k:
se (m.ind > ind *e* m.ind % γ = 0)
salvaCheckpoint()
ind ← m.ind
entrega a mensagem m para a aplicação

salvaCheckpoint()
grava um checkpoint em dispositivo estável
ind ← ind + 1

A Figura 3.6 ilustra um padrão de *checkpoints* e mensagens gerado pelo protocolo com $\gamma = 2$. Neste cenário, o *checkpoint* global consistente mais recente é representado por Σ.

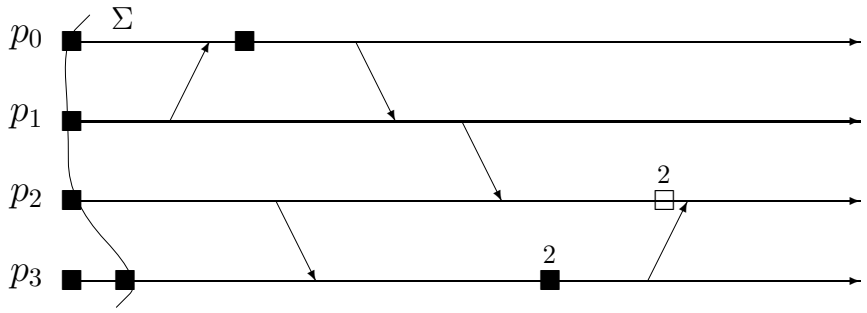


Figura 3.6: Exemplo de padrão gerado pelo protocolo WF com $\gamma = 2$

3.3 Protocolos Síncronos

Em contraste com as classes de protocolos de *checkpointing* assíncronos e quase-síncronos, os protocolos síncronos utilizam fases de troca de mensagens de controle para sincronizar o armazenamento de *checkpoints* nos processos e construir *checkpoints* globais consistentes. O início é determinado pela invocação por um dos processos, chamado iniciador, e o término é determinado pela a verificação de uma condição que indica que todos os processos envolvidos na execução já armazenaram seus *checkpoints*. A execução do protocolo, desde o início por um dos processos até o seu término é denominada nesta tese apenas de *construção consistente*.

Definição 11 Construção Consistente—*Uma construção consistente é a execução de um protocolo síncrono desde o momento em que o processo iniciador armazena um checkpoint até o momento em que um novo checkpoint global consistente é construído e todas as mensagens de controle propagadas para esta execução são entregues.*

Quando um processo p_{ini} requisita o armazenamento de um *checkpoint* c^{ι}_{ini}, um protocolo síncrono é invocado e inicia-se um procedimento coordenado para sincronizar as atividades de *checkpointing*. Após a execução do protocolo síncrono, c^{ι}_{ini} é consistente com o último *checkpoint* de cada processo. Esse procedimento é normalmente executado em três fases, como ilustrado na Figura 3.7.

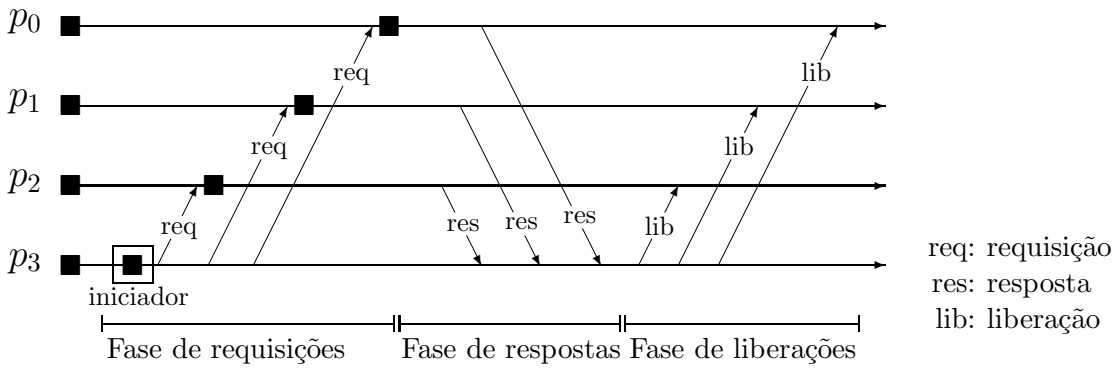


Figura 3.7: Protocolo de *checkpointing* síncrono

Na primeira fase, também chamada de fase de requisições, o iniciador p_{ini} armazena um *checkpoint* local, chamado de *checkpoint* provisório e envia mensagens de requisição para os processos com o objetivo de construir um *checkpoint* global consistente. Cada processo que recebe uma mensagem de requisição é chamado de *participante* da construção

consistente e, para atender a requisição, grava um *checkpoint* provisório e envia uma mensagem de resposta para p_{ini}.

A segunda fase, conhecida como fase de respostas, é composta pelo envio e recepção de mensagens de respostas e é utilizada para detectar se um novo *checkpoint* global consistente foi construído. A terceira fase, chamada de fase de liberações é iniciada após p_{ini} receber uma mensagem de resposta de cada um dos processos participantes. Nesse momento, p_{ini} transforma seu *checkpoint* provisório em permanente e envia uma mensagem de liberação para todos os processos participantes. Um processo, ao receber uma mensagem de liberação, transforma seu *checkpoint* provisório em permanente.

Ao final de uma construção consistente, o *checkpoint* global consistente representado pelos últimos *checkpoints* permanentes de cada processo forma o *checkpoint* global consistente mais recente existente até o momento, e portanto, em cada processo, a coleta de lixo restringe-se ao descarte do *checkpoint* anterior a este.

3.3.1 Protocolos Minimais

No início da execução de qualquer sistema distribuído, todo processo pode armazenar um *checkpoint* inicial e o conjunto formado por esses *checkpoints* representa um *checkpoint* global consistente. A cada invocação de um protocolo síncrono, um novo *checkpoint* global consistente é construído. Nesta seção, mostramos que para se construir um *checkpoint* global consistente que inclui o *checkpoint* do processo iniciador de um protocolo síncrono, nem sempre é necessário induzir *checkpoints* em todos os processos.

Vamos supor que existe um *checkpoint* global consistente Σ e o processo p_{ini} inicia uma construção consistente. O processo p_{ini}, ao requisitar o armazenamento de um *checkpoint* c^{ι}_{ini}, deve propagar mensagens de requisição com o objetivo de induzir *checkpoints* para formar um novo *checkpoint* global consistente Σ'. Porém, se p_{ini} não trocou mensagens da aplicação durante o seu último intervalo de *checkpoints*, c^{ι}_{ini} é consistente com os *checkpoints* pertencentes a Σ e portanto, Σ' pode ser formado pelos *checkpoints* de Σ substituindo o *checkpoint* de p_{ini} por c^{ι}_{ini} (Figura 3.8).

Mesmo que ocorra troca de mensagens durante o último intervalo de *checkpoints* do processo iniciador p_{ini}, é possível caracterizar um protocolo minimal, protocolo que induz um número minimal de *checkpoints* para construir um *checkpoint* global consistente, utilizando o conceito de *zigzag paths*. Considere Σ o *checkpoint* global consistente existente antes da execução do protocolo síncrono iniciado por p_{ini} e Σ' o *checkpoint* global consistente que inclui o *checkpoint* c^{ι}_{ini} salvo por p_{ini} durante sua construção consistente $\mathcal{C}$. Um *checkpoint* c faz parte de Σ' se $c \not\rightsquigarrow c^{\iota}_{ini}$. Note que $c^{\iota}_{ini} \rightsquigarrow c$ é impossível pois estamos considerando que p_{ini} não envia mensagens da aplicação durante $\mathcal{C}$. Observe também que c pode ter sido salvo antes ou durante $\mathcal{C}$. Portanto, o conjunto Σ' contém o

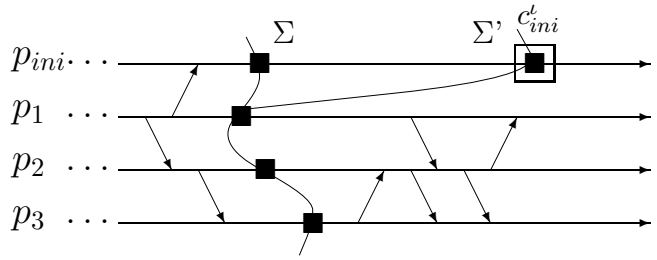


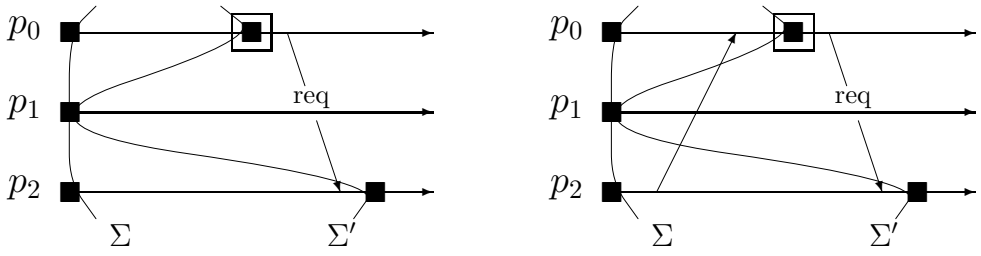
Figura 3.8: Construção consistente limitada a c^{ι}_{ini}

checkpoint salvo pelo iniciador, os *checkpoints* salvos durante a construção consistente $\mathcal{C}$ e pode incluir alguns *checkpoints* de Σ.

$\Sigma' = \{c_a^{\alpha_a} \mid c_a^{\alpha_a} \not\rightsquigarrow c^{\iota}_{ini}$ e $c_a^{\alpha_a}$ foi salvo durante $\mathcal{C}$ ou $c_a^{\alpha_a} \in \Sigma\}$, $\forall a$.

Definição 12 Protocolo Minimal—*Seja Σ o checkpoint global consistente mais à direita formado pelos checkpoints salvos antes da execução de um protocolo síncrono $\mathcal{C}$. Se o processo p_a possui um checkpoint que faz parte do conjunto Σ e que z-precede o checkpoint salvo pelo iniciador de $\mathcal{C}$, então p_a deve armazenar um novo checkpoint durante $\mathcal{C}$ para compor Σ'. O protocolo é minimal se apenas esses processos salvam checkpoints durante $\mathcal{C}$.*

A Figura 3.9 (a) ilustra um cenário de execução de um protocolo síncrono não-minimal pois Σ' engloba um *checkpoint* salvo por p_2 durante a construção consistente porém o *checkpoint* inicial de p_2 que pertence a Σ não z-precede o *checkpoint* do iniciador. Já a Figura 3.9 (b) ilustra uma execução de um protocolo síncrono minimal pois o *checkpoint* inicial de p_2 z-precede o *checkpoint* do iniciador e portanto, p_2 salva um *checkpoint* durante a construção consistente de p_0.



(a) não-minimal (b) minimal

Figura 3.9: Protocolos síncronos

Os protocolos minimais são bloqueantes [10], ou seja, durante uma construção consistente, os processos envolvidos devem permanecer bloqueados. O bloqueio de um processo implica na suspensão da execução das atividades da aplicação, sem interromper o mecanismo de troca de mensagens de controle. O desbloqueio do processo permite que o processo volte à execução de sua computação normal. Dado um padrão de *checkpoints* e mensagens, todos os protocolos síncronos minimais constroem os mesmos *checkpoints* globais consistentes e o número de *checkpoints* induzidos por qualquer um desses protocolos será mínimo para esse padrão.

Um dos primeiros protocolos minimais foi proposto por Koo e Toueg [20]. Este protocolo considera canais confiáveis FIFO (*First-In First-Out*). Neste protocolo, atribui-se índices às mensagens da aplicação possibilitando assim o rastreamento das precedências diretas entre os *checkpoints* no seu último intervalo. Cada processo possui um índice único de mensagens da aplicação, sendo que a primeira mensagem enviada pelo processo propaga o valor 1, a segunda propaga 2 e assim sucessivamente. Cada processo mantém dois vetores de inteiros com n entradas: uma para armazenar o índice da última mensagem recebida de cada processo e outra para armazenar o índice da primeira mensagem enviada para cada processo. A descrição do protocolo é apresentada pelo Protocolo 3.5. Consideramos que todo procedimento deste protocolo é executado de forma atômica.

Protocolo 3.5 Koo e Toueg (declarações)

Variáveis do processo:
- ultima_msg_rec ≡ vetor$[0 \ldots n-1]$ de inteiros
- primeira_msg_env ≡ vetor$[0 \ldots n-1]$ de inteiros
- num_colaboradores ≡ inteiro
- ind_msg ≡ inteiro
- bloqueado ≡ booleano
- iniciador ≡ booleano
- req_pid ≡ inteiro
- pid ≡ inteiro

Tipos de mensagem:
- aplicação:
  - ind_msg ≡ inteiro
- requisição:
  - ind_msg ≡ inteiro
- resposta
- liberação

A construção consistente é iniciada por um processo iniciador p_{ini} que armazena um *checkpoint* provisório, fica bloqueado e envia mensagens de requisição para todos os processos dos quais recebeu mensagem no seu último intervalo de *checkpoints*. Um processo p_j não bloqueado, ao receber a mensagem de requisição de p_{ini}, verifica se o valor do índice da requisição é maior ou igual ao índice da primeira mensagem enviada por p_j para p_{ini} no seu último intervalo de *checkpoints*. Se esta condição é falsa, o processo simplesmente envia uma mensagem de resposta para emissor da requisição. Senão, o processo grava um *checkpoint* provisório, propaga a mensagem de requisição de forma semelhante ao iniciador e fica bloqueado. Um processo p_j participante da construção consistente deve

Protocolo 3.5 Koo e Toueg

Início:
  $\forall i$: ultima_msg_rec[i] ← 0
  num_colaboradores ← 0
  ind_msg ← 0
  bloqueado ← *falso*
  iniciador ← *falso*
  salvaCkpt(permanente)

Envio da mensagem (m) da aplic. para p_k:
  ind_msg ← ind_msg + 1
  se (primeira_msg_env[k] = 0)
    primeira_msg_env[k] ← ind_msg
  m.ind_msg ← ind_msg
  envia mensagem da aplicação (m)

Recepção da mensagem (m) da aplic. de p_k:
  ultima_msg_rec[k] ← m.ind_msg
  processa a mensagem da aplicação (m)

Início da construção consistente:
  iniciador ← *verdadeiro*
  salvaCkpt(provisório)
  num_colaboradores ← número de entradas
    tais que ultima_msg_rec[i] ≠ 0
  se (num_colaboradores > 0)
    bloqueado ← *verdadeiro*
    $\forall i$: se (ultima_msg_rec[i] ≠ 0)
      req.ind_msg ← ultima_msg_rec[i]
      envia requisição (req) para p_i

Recepção da requisição (req) de p_k**:**
  bloqueado ← *verdadeiro*
  se (primeira_msg_env[k] > 0 *e*
      req.ind_msg ≥ primeira_msg_env[k])
    req_pid ← k
    salvaCkpt(provisório)
    num_colaboradores ← número de processos
      p_i tais que ultima_msg_rec[i] ≠ 0
    se (num_colaboradores > 0)
      $\forall i$: se (ultima_msg_rec[i] ≠ 0)
        req.ind_msg ← ultima_msg_rec[i]
        envia requisição (req) para p_i
    senão
      envia resposta (res) para p_k
  senão
    envia resposta (res) para p_k

Recepção da resposta (res) de p_k**:**
  num_colaboradores ← num_colaboradores - 1
  se (num_colaboradores = 0)
    se (iniciador)
      transformaCkpt()
      iniciador ← *falso*
      $\forall i$: se (ultima_msg_rec[i] ≠ 0)
        envia liberação (lib) para p_i
        ultima_msg_rec[i] ← 0
    senão
      envia resposta (res) para p_{req_pid}

Recepção da liberação (lib) de p_k**:**
  se (bloqueado)
    transformaCkpt()
    $\forall i$: se (ultima_msg_rec[i] ≠ 0)
      envia liberação (lib) para p_i
      ultima_msg_rec[i] ← 0

salvaCkpt(tipo)
  armazena um checkpoint de acordo com tipo
  $\forall i$: primeira_msg_env[i] ← 0

transformaCkpt()
  transforma o *checkpoint* provisório
    em permanente
  bloqueado ← *falso*

aguardar uma mensagem de resposta de cada um dos processos para os quais enviou uma mensagem de requisição e só então, deve enviar uma mensagem de resposta ao processo do qual recebeu uma mensagem de requisição. O iniciador, ao receber todas as mensagens de resposta que aguardava, inicia a fase de liberações: transforma seu *checkpoint* provisório em permanente, é desbloqueado e a mensagem de liberação é propagada de forma semelhante às mensagens de requisição.

A Figura 3.10 ilustra um padrão de *checkpoints* e mensagens gerado pelo protocolo de Koo e Toueg. Nesta figura, o bloqueio é representado por pontilhados na linha de execução de cada processo. Os *checkpoints* representados como provisórios são transformados em permanentes no final de cada construção consistente. Neste cenário, o *checkpoint* global consistente mais recente é representado por Σ.

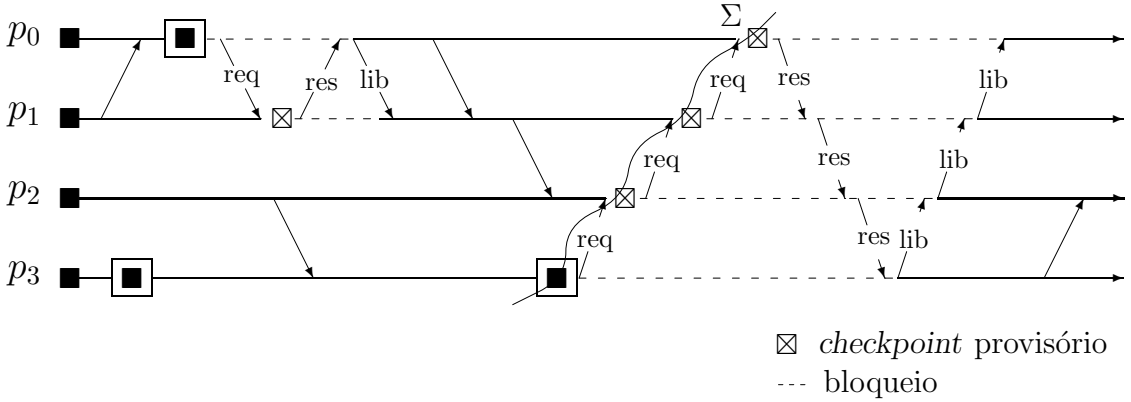


Figura 3.10: Exemplo do padrão gerado pelo protocolo de Koo e Toueg

Um protocolo minimal baseado no protocolo de Koo e Toueg foi proposto por Leu e Bhargava [22]. Este protocolo requer uma fase adicional de mensagens de controle para a formação de uma árvore que indica as precedências entre os processos que serão participantes nesta construção consistente. Este protocolo é apresentado na Seção B.1.

O número de mensagens de controle é reduzido no protocolo proposto por Prakash e Singhal por meio da utilização da detecção de terminação proposta por Huang [19] e da propagação de um vetor (vetor de participantes) que indica quais os processos que já estão participando da construção consistente [28]. Este protocolo é apresentado na Seção B.2.

Cao e Singhal propuseram uma nova abordagem, a qual chamamos de abordagem *broadcast*, que permite a um único processo (o iniciador) decidir quais os processos que devem armazenar *checkpoints* durante uma construção consistente [8]. Este protocolo é apresentado na Seção B.3. Neste protocolo, todos os processos devem ficar bloqueados para a aplicação enquanto o iniciador toma essa decisão. Provaremos na Seção 4.1.2 que estes dois últimos protocolos não são minimais.

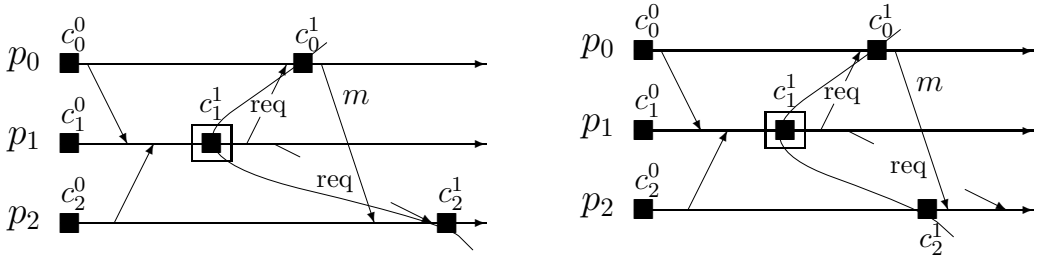
3.3.2 Protocolos Não-Bloqueantes

Um dos primeiros protocolos de *checkpointing* síncronos não-bloqueantes foi proposto por Chandy e Lamport e requer que todos os processos da aplicação armazenem seus

checkpoints durante uma construção consistente [11]. Recentemente, outros protocolos síncronos não-bloqueantes baseados nos protocolos síncronos bloqueantes minimais foram propostos com o objetivo de diminuir o número de *checkpoints* a cada construção consistente [8, 30]. Notamos que esta abordagem apresenta um compromisso entre a sincronia no armazenamento de *checkpoints* como nos protocolos síncronos e as garantias oferecidas para a formação de *checkpoints* globais consistentes como nos protocolos quase-síncronos. No Capítulo 5, mostramos que é possível desenvolver protocolos síncronos não-bloqueantes baseando-se em protocolos quase-síncronos.

De maneira análoga aos protocolos síncronos bloqueantes, os protocolos síncronos não-bloqueantes utilizam as três fases (fase de requisições, respostas e liberações) para sincronizar o armazenamento dos *checkpoints* e construir um *checkpoint* global consistente. Porém, a característica não-bloqueante desses protocolos permite que os processos continuem enviando e recebendo mensagens da aplicação, antes mesmo do término da construção consistente, podendo gerar uma inconsistência. Na Figura 3.11 (a), o iniciador p_1 envia mensagens de requisição para p_0 e p_2. O processo p_0 salva o *checkpoint* c_0^1 e envia uma mensagem da aplicação m para p_2 que recebe m antes da mensagem de requisição. Quando p_2 recebe a mensagem de requisição, armazena um *checkpoint* gerando uma inconsistência: $\{c_0^1, c_1^1, c_2^1\}$ forma um *checkpoint* global inconsistente pois engloba a recepção da mensagem m mas não o seu envio. Este problema pode ser solucionado pelo armazenamento de um *checkpoint* imediatamente antes da recepção de m. A Figura 3.11 (b) ilustra o mesmo padrão de *checkpoints* e mensagens da Figura 3.11 (a), porém p_2 salva um *checkpoint* provisório imediatamente antes de receber m. Quando p_2 recebe a mensagem de requisição de p_1, p_2 inclui c_2^1 na construção consistente iniciada por p_1. No final da construção consistente de p_1, o conjunto de *checkpoints* permanentes $\{c_0^1, c_1^1, c_2^1\}$ forma um *checkpoint* global consistente. Uma observação importante neste ponto é notar que apenas as mensagens da aplicação geram precedência causal entre os *checkpoints*, ou seja, mensagens geradas pelo protocolo de *checkpointing* (mensagens de controle) não introduzem a relação de precedência causal entre *checkpoints*.

O primeiro protocolo síncrono que tentou combinar a característica não-bloqueante e minimalidade no número de *checkpoints* foi proposto por Prakash e Singhal [30] e sua descrição se encontra na Seção C.1. Porém, Cao e Singhal mostraram que este protocolo pode resultar em inconsistências e propuseram uma correção para o protocolo de Prakash e Singhal [9] (Seção C.2). Cao e Singhal introduziram o conceito de *checkpoints* mutáveis e propuseram então um protocolo que relaxa a condição de minimalidade, garantindo apenas um número minimal de *checkpoints* salvos em memória estável [7, 8]. Um *checkpoint* mutável é salvo em memória não-estável e pode ser facilmente manipulado. Os *checkpoints* mutáveis são induzidos no momento da recepção de uma mensagem da aplicação de acordo com predicados do protocolo e podem ser transferidos para memória estável durante



(a) *Checkpoints* inconsistentes (b) Impedindo inconsistências

Figura 3.11: Impedindo a geração de *checkpoints* inconsistentes

uma construção consistente. Por exemplo, ao introduzirmos *checkpoints* mutáveis no padrão de *checkpoints* e mensagens da Figura 3.11 (b), p_2 salva um *checkpoint* mutável c_2^1 imediatamente antes de receber m. Quando p_2 recebe uma mensagem de requisição do iniciador p_1, p_2 transforma c_2^1 em provisório e ao receber uma mensagem de liberação, c_2^1 é salvo como permanente. Este protocolo considera que qualquer processo pode iniciar uma construção consistente, porém permite a presença de apenas um único iniciador a cada instante de tempo.

No protocolo de Cao e Singhal, se nenhum processo está envolvido em uma construção consistente, então, nenhum *checkpoint* é induzido. Um processo p_j salva um *checkpoint* mutável imediatamente antes de receber uma mensagem da aplicação de p_i se p_j não conhece nem o índice do último *checkpoint* de p_i nem a informação do iniciador conhecido por p_i e p_j enviou uma mensagem durante o intervalo de *checkpoints* corrente. Este *checkpoint* mutável é transformado em provisório se p_j recebe uma mensagem de requisição da construção consistente atual. Caso contrário, esse *checkpoint* mutável é descartado na fase de liberações. A detecção da terminação de uma construção consistente é baseada no protocolo proposto por Huang [19]. A descrição do protocolo de Cao e Singhal é apresentada pelo Protocolo 3.6.

Protocolo 3.6 Cao e Singhal 2003 (declarações)

Variáveis do processo:
VP ≡ vetor[0 . . . $n-1$] de bits
emCC ≡ de booleano
enviou ≡ de booleano
VI ≡ vetor[0 . . . $n-1$] de inteiros
ind_estavel ≡ inteiro
peso ≡ real
pid ≡ inteiro
inic { pid ≡ inteiro
ind ≡ inteiro }
MR { VI ≡ vetor[0 . . . $n-1$] de inteiros
VP ≡ vetor[0 . . . $n-1$] de bits }
mutavel { VP ≡ vetor[0 . . . $n-1$] de bits
inic ≡ inic
enviou ≡ booleano }

Tipos de mensagem:
aplicação:
ind ≡ inteiro
inic { pid ≡ inteiro
ind ≡ inteiro }
requisição:
peso ≡ real
MR { VI ≡ vetor[0 . . . $n-1$] de inteiros
VP ≡ vetor[0 . . . $n-1$] de bits }
ind ≡ inteiro
indreq ≡ inteiro
inic { pid ≡ inteiro
ind ≡ inteiro }
resposta:
peso ≡ real
liberação:
inic { pid ≡ inteiro
ind ≡ inteiro }

Protocolo 3.6 Cao e Singhal 2003

Início:
$\forall i$: VI[i] ← 0
emCC ← *falso*
inic.pid ← pid
inic.ind ← 0
peso ← 0
ind_estavel ← VI[i]
salvaCkpt(permanente)

Envio da mensagem (m) da aplic. para p_k:
enviou ← *verdadeiro*
se (emCC)
m.inic ← inic
senão
m.inic ← *nulo*
m.ind ← VI[pid]
envia mensagem da aplicação (m)

Recepção da mensagem (m) da aplic. de p_k:
se (m.ind ≤ VI[k])
VP[k] ← 1
processa mensagem da aplicação (m)
senão
se (VI[m.inic.pid] ≥ m.inic.ind)
VI[k] ← m.ind
VP[k] ← 1
processa a mensagem da aplicação (m)
senão
VI[k] ← m.ind
se (m.inic ≠ *nulo e* enviou = 1 *e*
m.inic ≠ inic)
salvaCkpt(mutável)
mutavel.inic ← m.inic
mutavel.VP ← VP
mutavel.enviou ← enviou
se m.inic ≠ *nulo e não* emCC
emCC ← *verdadeiro*
VI[pid] ← VI[pid] + 1
inic ← m.inic
VP[k] ← 1
processa a mensagem da aplicação (m)

Protocolo 3.6 Cao e Singhal 2003 (continuação)

Início da construção consistente:
VI[pid] ← VI[pid] + 1
inic.pid ← pid
inic.ind ← VI[pid]
emCC ← *verdadeiro*
peso ← 1
$\forall i$: MR.VI[i] ← 0
MR.VP[i] ← 0
MR.VI[pid] ← VI[pid]
MR.VP[pid] ← 1
propReq(VP, MR, inic, peso)
salvaCkpt(provisório)
ind_estavel ← VI[pid]
se (peso = 1)
transforma ckpt provisório em permanente

Recepção da requisição (req) de p_k:
VI[k] ← req.ind
se (ind_estavel > req.indreq)
res.peso ← req.peso
envia resposta (res) para $p_{req.inic.pid}$
senão
emCC ← *verdadeiro*
se (req.inic = inic)
se (mutavel.inic = req.inic)
propReq(mutavel.VP,
req.MR, req.inic, req.peso)
transforma mutavel em estável
ind_estavel ← VI[pid]
mutavel ← *nulo*
res.peso ← peso
envia resposta (res) para $p_{req.inic.pid}$
senão
res.peso ← req.peso
envia resposta (res) para $p_{req.inic.pid}$
senão
VI[pid] ← VI[pid] + 1
inic ← req.inic
propReq(VP, req.MR, req.inic, req.peso)
salvaCkpt(provisório)
ind_estavel ← VI[pid]
res.peso ← peso
envia resposta (res) para $p_{req.inic.pid}$

Recepção da resposta (res) de p_k:
peso ← peso + res.peso
se (peso = 1)
transforma ckpt provisório em permanente
emCC ← *falso*
$\forall i$: lib.inic ← inic
envia liberação (lib) para p_i

Recepção da liberação (lib) de p_k:
transforma ckpt provisório em permanente
VI[lib.inic.pid] ← lib.inic.ind
emCC ← *falso*
se (mutavel.inic = lib.inic *e* mutavel ≠ *nulo*)
enviou ← enviou ∪ mutavel.enviou
VP ← VP ∪ mutavel.VP
mutavel ← *nulo*

propReq(p, MR, i, peso_rec)
peso ← peso_rec
$\forall i$: tmp.VI[i] ← max(MR.VI[i], VI[i])
tmp.VP[i] ← max(MR.VP[i], p[i])
$\forall i$: se (p[i] = 1 *e*
max(MR.VI[i], VI[i]) ≠ MR.VI[i])
peso ← peso / 2
req.peso ← peso
req.MR ← tmp
req.ind ← VI[pid]
req.indreq ← VI[i]
req.inic ← i
envia requisição (req) para p_i

salvaCkpt(tipo)
armazena um checkpoint de acordo com tipo
$\forall i$: VP[i] ← 0
enviou ← *falso*

A Figura 3.12 ilustra um padrão de *checkpoints* e mensagens gerado pelo protocolo de Cao e Singhal. O *checkpoint* mutável salvo por p_1 é transformado em provisório no momento em que p_1 recebe uma mensagem de requisição de p_0. Os *checkpoints* representados como provisórios são transformados em permanentes no final da construção consistente iniciada por p_3 e o *checkpoint* global consistente mais recente está representado por Σ.

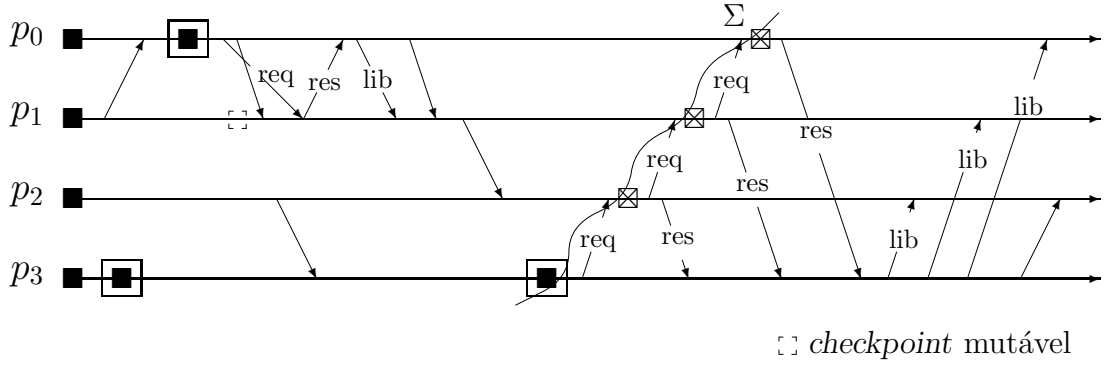
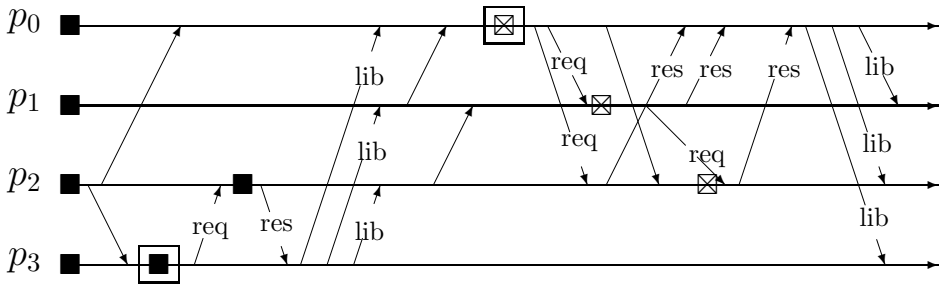


Figura 3.12: Exemplo de padrão gerado pelo protocolo de Cao e Singhal

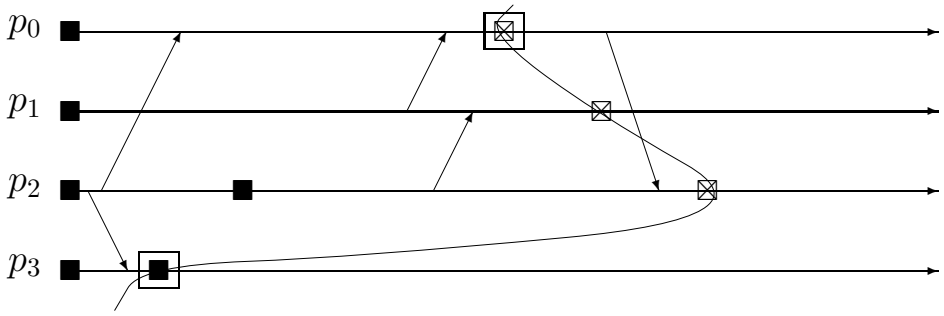
O protocolo de Cao e Singhal, como originalmente escrito, pode gerar uma inconsistência como apresenta a Figura 3.13 (a). Neste cenário, quando p_2 recebe uma mensagem de requisição de p_0, p_2 não salva um *checkpoint* provisório, pois p_0 não conhece o índice do último *checkpoint* permanente de p_2. Quando p_2 recebe uma mensagem da aplicação de p_0, p_2 não salva um *checkpoint* mutável, pois p_2 já tem conhecimento sobre o índice do último *checkpoint* de p_0. Quando p_2 recebe uma mensagem de requisição de p_1, p_2 salva um *checkpoint* provisório pois p_1 conhece o índice do seu último *checkpoint* permanente e p_2 não salvou nenhuma informação sobre a construção consistente de p_0, o que gera a inconsistência. Como as mensagens de controle não geram inconsistências, a Figura 3.13 (b) apresenta o mesmo cenário da Figura 3.13 (a) sem as mensagens de controle, possibilitando visualizar melhor a inconsistência gerada por este protocolo.

Esta inconsistência pode ser facilmente corrigida, alterando na recepção de uma mensagem de requisição, o momento em que o processo atualiza a informação sobre o índice do último *checkpoint* do emissor da mensagem de requisição. O procedimento da recepção da mensagem de requisição com esta pequena correção é descrito pelo Protocolo 3.7.

Um protocolo de *checkpointing* não-bloqueante não garante um número minimal de *checkpoints* para uma determinada construção consistente [10]. Cao e Singhal introduziram o conceito de *checkpoints* mutáveis para armazenar um número minimal de *checkpoints* em memória estável. Uma das contribuições desta tese é mostrar que isso só é possível se considerarmos um único iniciador a cada instante (Capítulo 5).



(a) com mensagens de controle



(b) sem mensagens de controle

Figura 3.13: Inconsistência gerada pelo protocolo de Cao e Singhal

Protocolo 3.7 Cao e Singhal 2003 - correção

Recepção da requisição (req) de p_k:
se (ind_estavel > req.indreq)
res.peso ← req.peso
envia resposta (res) para $p_{req.inic.pid}$
senão
VI[k] ← **req.ind**
em CC ← *verdadeiro*
se (req.inic = inic)
se (mutavel.inic = req.inic)
propReq(mutavel.VP,
req.MR, req.inic, req.peso)
transforma mutavel em estável
ind_estavel ← VI[pid]
mutavel ← *null*
res.peso ← peso
envia resposta (res) para $p_{req.inic.pid}$
senão
res.peso ← req.peso
envia resposta (res) para $p_{req.inic.pid}$
senão
VI[pid] ← VI[pid] + 1
inic ← req.inic
propReq(VP, req.MR, req.inic,
req.peso)
salvaCkpt(provisório)
ind_estavel ← VI[pid]
res.peso ← peso
envia resposta (res) para $p_{req.inic.pid}$

3.4 Recuperação de Falhas

As aplicações distribuídas possuem melhor desempenho, são escaláveis e permitem compartilhamento transparente de recursos existentes no sistema, porém estão mais suscetíveis a falhas. Na ocorrência de falhas, o mecanismo de recuperação é iniciado com o objetivo de evitar perda de computação. A recuperação de falhas pode ser realizada por dois mecanismos:

- *forward* – quando existe a possibilidade de remover os erros do estado corrente para então habilitar o processo a prosseguir com sua computação.
- *backward* – o estado sem erros do processo salvo anteriormente é utilizado para restabelecer o processo. Este mecanismo é mais conhecido como recuperação por retrocesso[2]. A recuperação por retrocesso é classificada como baseada em *checkpoint* que é menos restritiva e mais simples de implementar; ou baseada em *log*, onde todos os eventos são identificados e escritos na forma de *logs* em memória estável [14].

A recuperação por retrocesso baseada em *checkpoint* garante o armazenamento de estados corretos em memória estável. Para garantir a recuperação de um sistema distribuído com qualquer padrão de comunicação é necessário que o sistema retroceda para o estado global consistente mais recente. Em caso de falha global, basta que o mecanismo de recuperação selecione o último *checkpoint* global consistente formado por *checkpoints* estáveis e indique a cada processo para qual estado deve retroceder. Em caso de falha parcial, ou seja, apenas uma parte dos processos sofreram falhas, os estados correntes dos processos não-falhos também podem ser utilizados como estado correto para formar um novo *checkpoint* global consistente garantindo assim, o mínimo de perda de computação possível (menor custo de retrocesso). O conjunto de estados utilizados para restabelecer o sistema é chamado de *linha de recuperação*. Portanto, uma linha de recuperação é o *checkpoint* global mais recente (podendo incluir estados correntes do sistema), tendo em vista as dependências existentes entre *checkpoints*. Existem duas abordagens para a execução do mecanismo de recuperação: síncrona e assíncrona.

Na abordagem síncrona, um processo iniciador é responsável por calcular a linha de recuperação por meio das dependências entre os *checkpoints* existentes e controlar o retrocesso dos processos. Quando o mecanismo de retrocesso é iniciado, o iniciador faz um *broadcast* para requisitar as dependências entre *checkpoints* mantidas pelos processos. Um processo, ao receber essa mensagem de requisição, fica bloqueado e envia uma mensagem de resposta ao iniciador com as informações requisitadas. Pelas dependências recebidas por todos os processos, o iniciador calcula a linha de recuperação, possivelmente

[2]Em inglês: *rollback-recovery*

construindo um grafo de dependências de retrocesso, ou *R-graph* [5, 45]. Então, o iniciador envia uma mensagem para cada processo indicando para qual estado ele deve retroceder e todo processo deve obedecer às instruções contidas nessa mensagem.

No mecanismo de retrocesso assíncrono, o processo falho p_f retrocede ao seu último estado salvo em memória estável e faz um *broadcast* avisando a todos os processos para qual estado ele retrocedeu. Todo processo ao receber a mensagem de p_f, continua sua execução se seu estado atual é consistente com o estado de p_f ou retrocede para garantir a consistência do estado global da computação distribuída e também distribui o aviso de seu retrocesso. Esse procedimento é repetido até que um *checkpoint* global consistente seja recuperado. Estes protocolos são complexos e muitas vezes menos eficientes [23].

O mecanismo de recuperação deve tratar também as mensagens que se encontram em trânsito. Para isso, é necessário que as informações sobre os eventos de envio e recepção de mensagens sejam gravadas em memória estável na forma de *logs*. Assim, ao recomeçar a execução, os processos podem ser requisitados a reenviar as mensagens ou podem ser replicadas no receptor garantindo que elas não sejam perdidas.

3.5 Coleta de Lixo

Checkpoints e *logs* de eventos consomem recursos de armazenamento e quanto mais a aplicação progride, mais informações de recuperação devem ser armazenadas. O mecanismo de coleta de lixo identifica e remove os *checkpoints* obsoletos, ou seja, *checkpoints* que não serão utilizadas por nenhuma linha de recuperação. Um algoritmo para coleta de lixo é ótimo se é capaz de identificar e eliminar todos os *checkpoints* obsoletos de um padrão de *checkpoints* e mensagens.

Um *checkpoint* que está no passado de um *checkpoint* global consistente formado por apenas *checkpoints* estáveis é um *checkpoint* obsoleto. Esta condição é suficiente para a implementação de um algoritmo simples de coleta de lixo, conhecido como coleta de lixo ingênua. Este algoritmo verifica qual o *checkpoint* global consistente mais recente formado apenas por *checkpoints* estáveis para remover os *checkpoints* que estão em seu passado. Este mecanismo é facilmente implementado por protocolos de *checkpointing* síncronos que durante sua execução, requerem que cada processo mantenha, no máximo, dois *checkpoints* estáveis garantindo assim a existência de pelo menos um *checkpoint* global consistente. Apenas quando todos os *checkpoints* de um novo *checkpoint* global consistente foram induzidos e salvos em memória estável é que os *checkpoints* anteriores tornam-se obsoletos e podem ser removidos. Uma atenção especial deve ser dada aos protocolos síncronos não-bloqueantes. Esses protocolos induzem *checkpoints* adicionais no intervalo de duas execuções do protocolo para garantir consistência. Portanto, um número maior de *checkpoints* devem ser mantidos pelos processos e os *checkpoints* são

removidos apenas no final de uma construção consistente.

A coleta de lixo ingênua também pode ser utilizada em aplicações que implementam protocolos de *checkpointing* quase-síncronos ou assíncronos. Neste caso, a verificação do *checkpoint* global consistente mais recente pode ser feita de forma semelhante ao algoritmo de recuperação por retrocesso síncrono. Escolhe-se um processo inicial que requisita pelas dependências entre *checkpoints* conhecidas por todos os processos e por meio dessa informação, calcula-se a linha de recuperação formada por *checkpoints* estáveis, possivelmente utilizando *R-graph* [5, 45]. O resultado é propagado para todos os processos de forma que eles possam remover os *checkpoints* salvos no passado da linha de recuperação obtida. Outra maneira de realizar a coleta de lixo ingênua é implementando-se um monitor da computação distribuída. Quando um *checkpoint* é salvo, suas informações são enviadas ao monitor que constrói a linha de recuperação progressivamente. A cada novo *checkpoint* global consistente detectado pelo monitor, os processos são avisados para que possam efetuar a coleta.

A coleta de lixo ingênua é simples, porém requer conhecimento global do sistema distribuído para o cálculo da linha de recuperação. Além disso, não existe um número máximo de *checkpoints* não coletados, ou seja, não existe limite máximo de *checkpoints* que o processo deve manter em memória estável. O mecanismo de coleta de lixo ótimo deve remover, não somente os *checkpoints* que estão no passado de uma linha de recuperação, mas também os *checkpoints* que não serão usados por nenhuma outra linha de recuperação, mesmo que estes estejam no futuro do *checkpoint* global consistente mais recente formado por *checkpoints* estáveis. O algoritmo de coleta de lixo ótimo constrói o *R-graph* de forma semelhante ao algoritmo de recuperação por retrocesso síncrono para identificar os *checkpoints* estáveis que fazem parte de alguma linha de recuperação (e não apenas do *checkpoint* global consistente mais recente). Todos os outros *checkpoints* são obsoletos [38]. Em particular, a coleta de lixo ingênua ou a ótima em padrões de *checkpoints* e mensagens que satisfazem a propriedade RDT podem ser simplificadas. Neste caso, os processos devem manter, no máximo, n *checkpoints* e a linha de recuperação pode ser calculada apenas com informações locais sem a necessidade de troca de mensagens ou bloqueio dos processos durante sua execução [37].

3.6 Sumário

Uma possível maneira de evitar perda de computação na recuperação de um sistema distribuído após a ocorrência de uma falha é implementar um mecanismo que envolve: (i) seleção de *checkpoints* (onde e como armazenar o estado corrente de um processo); (ii) garantia da existência de *checkpoints* globais consistentes; (iii) recuperação a partir de um checkpoint global consistente e (iv) coleta de lixo.

Os protocolos de *checkpointing* são responsáveis pela seleção de *checkpoints* e são classificados como assíncronos, quase-síncronos e síncronos. Na abordagem assíncrona, apenas os *checkpoints* requisitados pela aplicação são salvos em memória estável e não há garantia da formação de *checkpoints* globais consistentes, ou seja, as aplicações que implementam protocolos desta classe estão sujeitas ao efeito dominó.

Em geral, os protocolos quase-síncronos adicionam informações de controle às mensagens para detectar a existência de *zigzag paths* entre os *checkpoints* dos processos emissor e receptor da mensagem. Assim, na recepção de uma mensagem da aplicação, o protocolo pode salvar ou não um *checkpoint* antes de entregar a mensagem para garantir a posterior formação de *checkpoints* globais consistentes [6, 47, 17]. A maioria dos protocolos desta classe considera que o sistema possui meio de armazenamento estável suficiente para gravar todos os *checkpoints* requisitados. Na prática, aplicações que implementam protocolos quase-síncronos devem implementar também um protocolo de coleta de lixo para remover os *checkpoints* obsoletos (que não serão utilizados para recuperação).

Os protocolos síncronos utilizam mensagens de controle adicionais para sincronizar o armazenamento dos *checkpoints* garantindo a construção de um *checkpoint* global consistente para cada *checkpoint* requisitado pela aplicação. Uma maneira simples de coordenar o armazenamento de *checkpoints* é propagar mensagens de controle para que todos os processos salvem seus *checkpoints* suspendendo os procedimentos de entrega de mensagens para a aplicação e de envio de mensagens para o subsistema de rede de comunicação durante sua execução. Para reduzir o custo de armazenamento de *checkpoints*, protocolos síncronos minimais induzem apenas um número minimal de processos a armazenarem *checkpoints* durante sua execução [20, 22, 28, 35]. A coleta de lixo normalmente é implementada pelo próprio protocolo de *checkpointing* síncrono que, em sua fase final (após armazenar os *checkpoints* estáveis que fazem parte de um *checkpoint* global consistente) remove os *checkpoints* salvos anteriormente. Nestes protocolos, os processos mantêm, no máximo, dois *checkpoints* em memória estável para garantir tolerância a falhas durante uma execução do protocolo.

Os protocolos síncronos não-bloqueantes permitem que a aplicação continue sua computação mesmo durante uma execução de *checkpointing*. O primeiro e mais conhecido protocolo síncrono não-bloqueante utiliza canais de comunicação FIFO confiáveis e o envio de controle por todos os canais de comunicação para que todos os processos armazenem seus *checkpoints* na formação de um *checkpoint* global consistente [11]. Recentemente, foram propostos protocolos síncronos não-bloqueantes que reduzem o número de *checkpoints* salvos em memória estável e ainda assim, garantem a formação de *checkpoints* globais consistentes [9, 7, 8, 28]. Estes protocolos originalmente foram baseados em protocolos síncronos existentes, porém notamos que estes possuem não somente características de protocolos síncronos (como sincronizar o armazenamento de *checkpoints*), mas também

necessitam propagar informações de controle com as mensagens da aplicação para induzir ou não *checkpoints* forçados impedindo inconsistências, ou seja, os protocolos síncronos não-bloqueantes necessitam de mecanismos como os propostos pelos protocolos quase-síncronos. Nos próximos capítulos, mostraremos que além de ser possível desenvolver protocolos síncronos baseados em protocolos quase-síncronos, esses protocolos são mais simples, permitem iniciadores concorrentes e não requerem que todos os processos salvem *checkpoints* durante uma execução do protocolo.

Capítulo 4

Protocolos de *Checkpointing* Síncronos Minimais

Os protocolos síncronos simplificam a recuperação de uma falha e evitam o efeito dominó mantendo *checkpoints* globais consistentes em memória estável. Para reduzir o custo de armazenamento a cada construção consistente, os protocolos minimais bloqueiam os processos para induzir apenas um número minimal de processos a armazenarem *checkpoints* [8, 20, 22, 35, 36].

Na literatura, existem dois protocolos que foram propostos com o objetivo de alcançar um número minimal de *checkpoints* [8, 28], porém provamos por meio de contra-exemplos, que estes protocolos não são minimais. Notamos que a estrutura de dados utilizada por ambos os protocolos não é suficiente para garantir a minimalidade no número de *checkpoints*. Este problema nos motivou a procurar um novo mecanismo de rastreamento, baseado no uso de vetores de dependências, para propor novos protocolos [35, 36].

Este capítulo é dividido em duas partes: a Seção 4.1 considera um único iniciador e a Seção 4.2 considera iniciadores concorrentes. Para cada uma delas, descrevemos suas características e discutimos sobre a questão da minimalidade no número de *checkpoints* e propomos novos protocolos: Broad-minimal que permite apenas um único iniciador a cada instante e VD-minimal que permite a execução de iniciações concorrentes.

4.1 Único Iniciador

Nesta Seção, consideramos a presença de um único iniciador a cada instante de tempo. Isto significa que, uma nova construção consistente não inicia enquanto todas as mensagens de controle da construção consistente anterior não tiverem sido entregues.

4.1.1 Características

A busca pela minimalidade no número de *checkpoints* salvos durante uma construção consistente gerou duas abordagens para a organização dos processos nos protocolos síncronos minimais: (i) em níveis e (ii) *broadcast*. A abordagem em que os processos se organizam em níveis bloqueia apenas os processos participantes da construção consistente e a decisão de salvar um *checkpoint* ou não é determinada pelos processos participantes [20, 22, 35]. Em contraste, a abordagem baseada em *broadcast* bloqueia todos os processos e transfere exclusivamente para o iniciador a tarefa de determinar quais processos devem armazenar *checkpoints* durante uma construção consistente [8, 36].

Protocolos Minimais com Abordagem em Níveis

Na abordagem em níveis, cada processo deve capturar e manter as precedências causais entre *checkpoints* formadas no seu último intervalo de *checkpoints*. Quando o iniciador p_{ini} salva um *checkpoint* para iniciar uma construção consistente de um protocolo minimal, p_{ini} utiliza o seu conhecimento sobre as precedências causais para selecionar os processos participantes e propagar-lhes mensagens de requisição (primeiro nível). Cada processo que recebe uma mensagem de requisição também é capaz de selecionar os processos de interesse formando outros níveis de propagação de mensagens de requisição.

Na literatura, notamos diferentes mecanismos para rastrear as dependências entre *checkpoints*:

- índices nas mensagens da aplicação — este mecanismo é utilizado pelo protocolo minimal proposto por Koo e Toueg [20]. Cada processo mantém um contador único para as mensagens da aplicação e dois vetores para anotar o índice da primeira mensagem enviada e da última recebida pelos processos. A entrada a do vetor de mensagens recebidas de p_{ini} indica o índice da última mensagem enviada por p_a e recebida por p_{ini} e a entrada a do vetor de mensagens enviadas de p_{ini} indica o índice da primeira mensagem enviada por p_{ini} para p_a no último intervalo de *checkpoints*. Assim, quando p_a recebe uma mensagem de requisição do iniciador p_{ini}, p_a verifica no seu vetor de primeiras mensagens enviadas se a dependência foi formada após o seu último *checkpoint*, ou seja, se p_{ini} conhece o índice da primeira mensagem que p_a enviou para p_{ini}. Neste caso, p_a salva um *checkpoint* e propaga mensagens de requisição para todos os processos dos quais recebeu mensagens no seu último intervalo de *checkpoints*. Essa propagação das mensagens de requisição em níveis é feita até que todos os participantes da construção consistente tenham recebido pelo menos uma mensagem de requisição. Na Figura 4.1, o vetor com o índice das primeiras mensagens enviadas e das últimas mensagens recebidas são representados por $(\ )_p$ e $(\ )_u$, respectivamente.

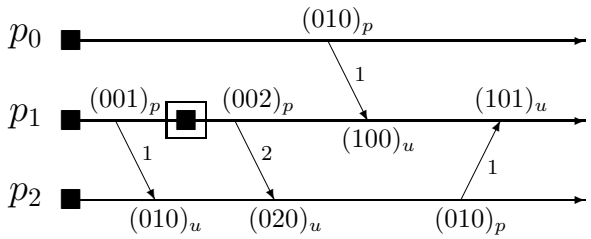


Figura 4.1: Rastreamento de dependências por meio de índices nas mensagens

- índices dos *checkpoints* — este mecanismo é semelhante ao mecanismo de relógios lógicos e é utilizado por Leu e Bhargava [22]. Cada mensagem da aplicação propaga o índice do último *checkpoint* salvo pelo seu emissor. O processo que recebe a mensagem da aplicação mantém um vetor com o maior índice recebido pelos processos. Quando um processo iniciador p_{ini} salva um *checkpoint*, uma árvore virtual representando as dependências estabelecidas no intervalo de *checkpoints* corrente é construída. Assim, os índices maiores que zero do vetor do iniciador (dos quais recebeu informação sobre um novo índice após seu último *checkpoint*) indicam seus potenciais filhos, para os quais envia uma mensagem de requisição. Se o índice do último *checkpoint* do processo p_a que recebeu a requisição é menor ou igual ao índice conhecido pelo pai p_{ini}, então p_a envia uma mensagem de confirmação positiva ao pai p_{ini} e p_{ini} inclui p_a como seu filho e p_a propaga a mensagem de requisição de forma semelhante a p_{ini}. A árvore construída é utilizada para propagar as mensagens de resposta e liberação. A Figura 4.2 ilustra um cenário com a propagação dos índices do *checkpoints*.

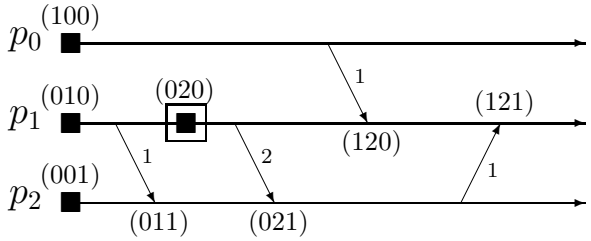


Figura 4.2: Rastreamento de dependências por meio de índices dos *checkpoints*

- vetores de bits — neste mecanismo, cada processo p_a mantém um vetor de bits que, quando ativo na posição i, indica que p_a recebeu pelo menos uma mensagem de p_i

em seu último intervalo de *checkpoints* [8, 28]. Assim, todos os processos mantêm informações de quais processos receberam mensagens após o seu último *checkpoint*. Quando o iniciador salva um *checkpoint*, envia mensagens de requisição para todos os processos dos quais recebeu pelo menos uma mensagem no seu último intervalo de *checkpoints*. Neste capítulo, mostramos que o uso deste mecanismo não garante a minimalidade no número de *checkpoints* (Seção 4.1.2). A Figura 4.3 ilustra o uso de vetores de bits.

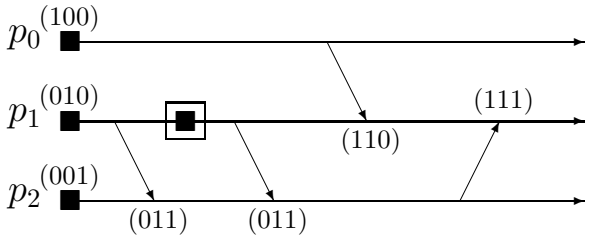


Figura 4.3: Rastreamento de dependências por meio de vetores de bits

- vetores de dependências — o rastreamento das precedências causais também pode ser realizado por meio do uso de vetores de dependências como visto na Seção 2.3. Os vetores de dependências capturam precedências causais transitivas e, portanto, se existe uma *zigzag path* causal a partir do último *checkpoint* de p_a até o *checkpoint* do iniciador, então p_a receberá uma mensagem de requisição do iniciador no primeiro nível. Assim, os protocolos que utilizam vetores de dependências para capturar as precedências causais podem utilizar um número menor de níveis, comparado aos protocolos que utilizam captura de precedências diretas, para atingir todos os participantes de uma construção consistente. A Figura 4.4 ilustra o uso de vetores de dependências.

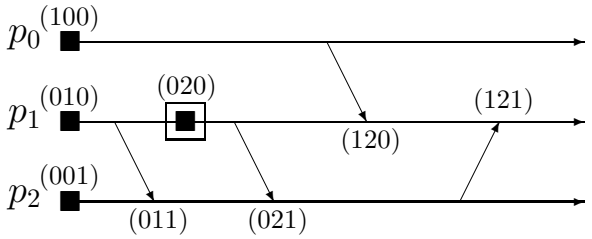


Figura 4.4: Rastreamento de dependências por meio de vetores de dependências

As principais características de cada um dos mecanismos citados podem ser resumidas da seguinte maneira:

Mecanismo de rastreamento	número de fases	precedência capturada	informação de controle	variáveis do processo
índices nas mensagens	3	direta	inteiro	2 vetores (inteiros)
índices dos *checkpoints*	4	direta	inteiro	vetor (inteiros)
vetores de bits	3	direta	inteiro	vetor (bits)
vetores de dependências	3	transitiva	vetor (inteiros)	vetor (inteiros)

Diferentes mecanismos para detecção da formação de um novo *checkpoint* global consistente são utilizados pelos protocolos síncronos. A determinação de para quais processos as mensagens de resposta devem ser enviadas e o número de mensagens de resposta e de liberação dependem do mecanismo de detecção de terminação utilizado. Uma maneira simples de detectar a terminação de uma construção consistente é permitir que um processo só envie a mensagem de resposta ao processo do nível anterior após receber mensagens de respostas de todos os processos do nível posterior [20, 22]. Outros mecanismos foram baseados em protocolos de detecção de terminação em computação por difusão [12, 19].

Os protocolos minimais propostos na literatura que utilizam a abordagem em níveis são executados em três fases [20, 22, 35]:

1. fase de requisições – um processo iniciador, ao invocar o protocolo síncrono, armazena um *checkpoint* provisório, fica bloqueado e propaga as mensagens de requisição. No primeiro nível, o iniciador envia mensagens de requisição para os processos que possuem um *checkpoint* que precede causalmente o seu último *checkpoint*. No segundo nível, os processos, ao receberem uma mensagem de requisição do iniciador, verificam a necessidade de armazenar um *checkpoint* provisório para então ficarem bloqueados e propagarem mensagens de requisição formando os níveis subseqüentes. Se existe uma *zigzag path* do último *checkpoint* de p_a até o *checkpoint* do iniciador de tamanho $n-1$ no qual cada par de mensagens é formado por uma *zigzag path* não-causal, então p_b receberá uma mensagem de requisição no $(n-1)$-ésimo nível de propagação de requisição. Note que $n-1$ é o número limitante para o número de níveis em um protocolo minimal.

2. fase de respostas – todo processo participante da construção consistente envia uma mensagem de resposta de acordo com o protocolo de detecção de terminação escolhido.

3. fase de liberações – quando o predicado de terminação de uma construção de um *checkpoint* global consistente é satisfeito, o iniciador passa para a terceira fase,

propagando mensagens de liberação. Todo processo que recebe uma mensagem de liberação, transforma o seu *checkpoint* provisório em permanente, é desbloqueado e volta à sua computação normal.

A Figura 4.5 ilustra as fases de um protocolo minimal com a abordagem em níveis. O processo p_3 é o iniciador e envia uma mensagem de requisição para p_2 no primeiro nível. No segundo nível, p_2, após salvar um *checkpoint*, envia uma mensagem de requisição para p_1 e para p_0. Note que nem todos os processos necessitam armazenar *checkpoints* em uma construção consistente. Os processos que armazenam *checkpoints* ficam bloqueados e propagam a mensagem de requisição. Todo processo que recebe uma mensagem de requisição, envia uma mensagem de resposta ao iniciador que encerra a construção consistente por meio do envio das mensagens de liberação.

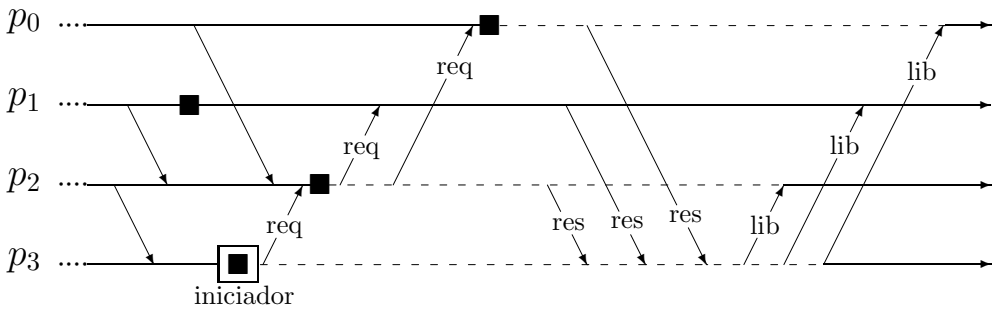


Figura 4.5: Protocolo minimal com abordagem em níveis

Protocolos Minimais com Abordagem *Broadcast*

Cao e Singhal propuseram uma nova abordagem, a qual chamamos de abordagem *broadcast*, que permite a um único processo (o iniciador) decidir quais processos devem armazenar *checkpoints* durante uma construção consistente [8]. Esta abordagem utiliza duas fases adicionais iniciais para que o iniciador receba informações sobre as dependências conhecidas pelos processos e verifique quais processos devem participar de sua construção consistente. Todos os processos devem ficar bloqueados enquanto o iniciador toma essa decisão. As fases utilizadas por protocolos minimais com abordagem *broadcast* são descritas a seguir:

1. fase de *broadcast* – o iniciador salva um *checkpoint* provisório, faz um *broadcast* da mensagem de bloqueio requisitando informações de controle e fica bloqueado.

2. fase de respostas ao bloqueio – todo processo que recebe uma mensagem de bloqueio envia ao iniciador uma mensagem de resposta ao bloqueio com suas informações de dependência entre *checkpoints* e fica bloqueado.

3. fase de requisições – o iniciador verifica quais os processos cujo último *checkpoint* z-precede o seu último *checkpoint* provisório, ou seja, os processos que devem armazenar *checkpoints* para a construção de um *checkpoint* global consistente garantindo apenas um número minimal de *checkpoint* salvos. O iniciador envia uma mensagem de requisição para cada processo que deve salvar um *checkpoint* e uma mensagem de liberação aos outros processos.

4. fase de respostas – todo processo participante da construção consistente deve, após salvar um *checkpoint* provisório, enviar uma mensagem de resposta ao iniciador.

5. fase de liberações – o iniciador, após receber mensagens de resposta de cada processo participante, transforma seu *checkpoint* provisório em permanente, é desbloqueado e envia uma mensagem de liberação aos participantes da construção consistente. Os processos que recebem uma mensagem de liberação transformam seus *checkpoints* provisórios em permanentes e são desbloqueados.

Nesta abordagem, todos os processos ficam bloqueados em um primeiro momento, porém apenas os processos que necessitam armazenar *checkpoints* permanecem bloqueados até o fim da construção consistente. Além disso, as mensagem de controle são propagadas diretamente do iniciador aos participantes da construção consistente, e nenhum processo recebe duas ou mais mensagens de controle do mesmo tipo. Um exemplo de execução da abordagem *broadcast* é ilustrado pela Figura 4.6.

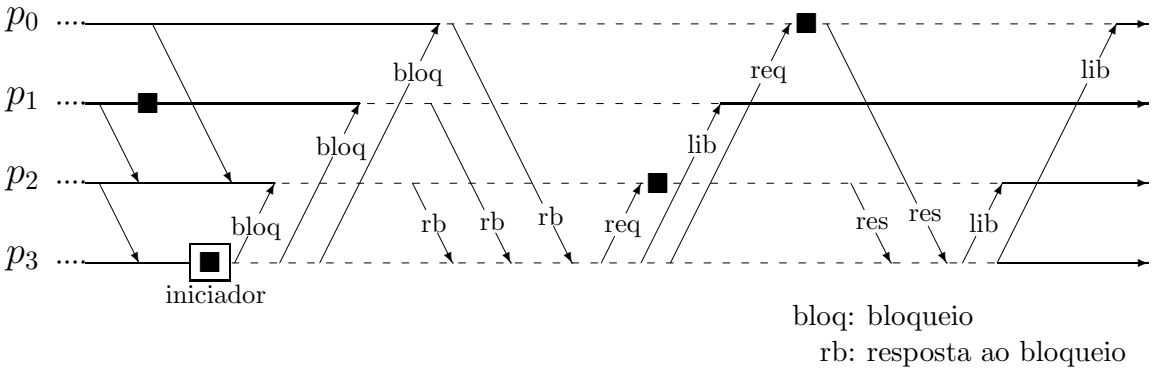


Figura 4.6: Protocolo minimal com abordagem *broadcast*

4.1.2 Minimalidade no Número de *Checkpoints*

Na seção anterior, descrevemos algumas características dos protocolos síncronos minimais em função das fases e estratégias de troca de mensagens utilizadas para a obtenção de *checkpoints* globais consistentes. Nesta seção, descrevemos esses protocolos sob o ponto de vista da busca pela minimalidade. Um protocolo minimal pode deixar de ser correto durante a construção consistente se:

- deixa de incluir entre os participantes um processo cujo último *checkpoint* z-precede o *checkpoint* armazenado pelo iniciador e/ou
- inclui no conjunto de participantes um processo que salva novo *checkpoint*, porém seu último *checkpoint* não z-precede o *checkpoint* armazenado pelo iniciador.

Um protocolo que exibe o primeiro defeito, obterá ao final da construção consistente um *checkpoint* global *inconsistente*. Um protocolo que exibe apenas o segundo defeito deixa de ser minimal porque incluiu no conjunto de participantes um processo que salva um novo *checkpoint* desnecessariamente, porém constrói um *checkpoint* global *consistente*.

Prakash e Singhal propuseram um protocolo com abordagem em níveis que utiliza vetores de bits para rastrear as dependências entre *checkpoints* [28] (Seção B.2). Porém provamos a sua não-minimalidade pelo contra-exemplo ilustrado pela Figura 4.7 [35]. Nesta figura, p_1 armazena um *checkpoint* por causa da requisição enviada por p_0 e, na recepção da requisição de p_2 armazena novamente outro *checkpoint*, sem ter enviado ou recebido nenhuma mensagem da aplicação durante esse intervalo. A união dos *checkpoints* requisitados pelo iniciador p_2 é um *checkpoint* global consistente mas não minimal.

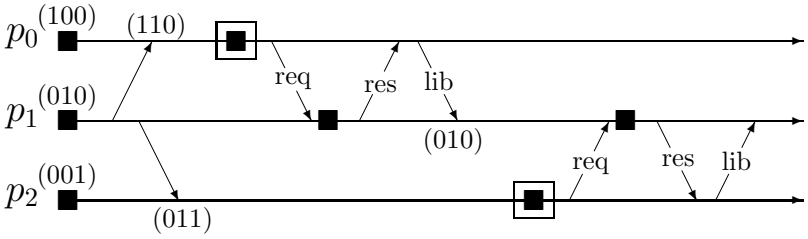


Figura 4.7: Não-minimalidade do protocolo proposto por Prakash e Singhal

Outro protocolo sem êxito no objetivo de alcançar a minimalidade no número de *checkpoints* foi proposto por Cao e Singhal [8] (Seção B.3). Este protocolo possui abordagem *broadcast* e também utiliza vetores de bits para rastrear as dependências entre *checkpoints*. Provamos, pela Figura 4.8, que este protocolo também não é minimal [36]. Nesta

figura, quando p_0 inicia uma construção consistente, p_0 envia uma mensagem de requisição para p_1 e p_1 salva um novo *checkpoint* d, porém, o *checkpoint* b de p_1 é consistente com o *checkpoint* salvo pelo iniciador p_0.

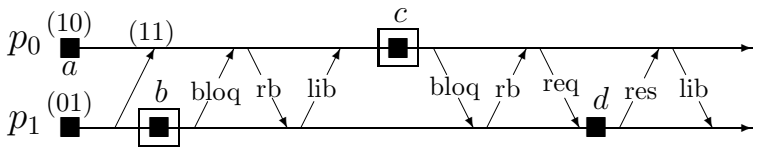


Figura 4.8: Não-minimalidade do protocolo proposto por Cao e Singhal

A seguir, mostramos que os protocolos que anotam a recepção de mensagens, mas não os intervalos de origem dessas mensagens podem incluir no conjunto de participantes um processo cujo último *checkpoint* não z-precede o *checkpoint* armazenado pelo iniciador, deixando portanto, de serem minimais. Isto ocorre pois um processo que recebe uma mensagem de requisição, não tem como avaliar se a precedência reconhecida foi estabelecida após o armazenamento do seu último *checkpoint*.

Teorema 2 *Protocolos que anotam a recepção de mensagens, mas não os intervalos de origem dessas mensagens não têm informação suficiente para serem minimais.*

Prova: Por contradição, vamos supor que anotando apenas a recepção de mensagens é suficiente para desenvolver um protocolo minimal. Sabemos que um protocolo minimal induz *checkpoints* apenas nos processos cujo último *checkpoint* z-precede o *checkpoint* do iniciador. Quando uma mensagem da aplicação m é enviada de p_a para p_b, nenhuma informação de controle necessita ser propagada e o processo p_b deve anotar a recepção de m após o armazenamento do seu último *checkpoint* (Figura 4.9). Portanto, se p_b inicia uma construção consistente após receber m, p_b salva um *checkpoint* e envia uma mensagem de requisição para p_a. Se p_a salva um *checkpoint* neste ponto, então o protocolo não é minimal. □

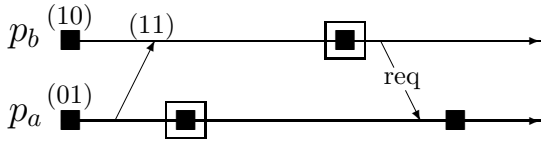


Figura 4.9: Cenário para a prova do Teorema 2

Através do Teorema 2, concluímos que a abordagem utilizada para rastrear as precedências entre *checkpoints* por alguns protocolos não garante minimalidade como descritos na literatura [8, 28]. A seguir, propomos dois novos protocolos: um com a abordagem em níveis e outro com abordagem *broadcast*. Esses protocolos utilizam vetores de dependências para rastrear as precedências e garantir um número minimal de *checkpoints* salvos durante uma construção consistente.

4.1.3 Protocolo Broad-minimal

A abordagem *broadcast* foi proposta originalmente por Cao e Singhal [8]. Este protocolo tenta reduzir o tempo de bloqueio nos processos fazendo do iniciador o único responsável em decidir quais processos devem induzir *checkpoints* para a construção de *checkpoints* globais consistentes. A não-minimalidade deste protocolo (Seção 4.1.2) nos motivou a propor um novo protocolo, chamado de Broad-minimal que garante um número minimal de *checkpoints* e considera que há um único iniciador a cada instante de tempo [36].

Descrição do Protocolo

Quando um processo p_{ini} inicia sua construção consistente, armazena um *checkpoint* provisório e faz um *broadcast* de mensagens de bloqueio a todos os processos do sistema. Cada processo, ao receber a mensagem de bloqueio, fica bloqueado e envia seu vetor de dependências e seu vetor de participantes (informação de precedências causais do último intervalo de *checkpoints* do processo) para o iniciador. O iniciador, ao receber mensagens de resposta ao bloqueio de todos os processos, define o conjunto de processos que devem salvar *checkpoints* para induzir um número minimal de *checkpoints* e construir um *checkpoint* global consistente. Os processos selecionados recebem mensagens de requisição e os outros processos recebem mensagens de liberação. Um processo que recebe uma mensagem de requisição, salva um *checkpoint* e envia uma mensagem de resposta ao iniciador. O iniciador, após receber mensagens de resposta de todos os participantes, propaga mensagens de liberação. Todo processo que recebe uma mensagem de liberação, transforma seu *checkpoint* provisório em permanente (se este existir) e é desbloqueado.

Variáveis do processo

As variáveis utilizadas por este protocolo são descritas a seguir:

- $\texttt{VD}_i$ – vetor de dependências mantido pelo processo p_i para capturar as dependências causais entre *checkpoints*. Este vetor é propagado com as mensagens da aplicação e quando p_i recebe uma mensagem, seu vetor é atualizado da seguinte maneira: $\texttt{VD}_i$ `= max(m.VD,` $\texttt{VD}_i$`)`

- $\texttt{perm\_VD}_i$ – vetor de dependências salvo com o *checkpoint* permanente mantido pelo processo p_i.

- $\texttt{VP}_i$ – vetor de participantes que indica se p_i recebeu uma mensagem com informação de novo *checkpoint* dos processos durante o seu intervalo de *checkpoints* corrente. Este vetor é atualizado da seguinte maneira:

$$\texttt{VP}_\texttt{i}[\texttt{j}] = \begin{cases} \texttt{VD}_\texttt{i}[\texttt{j}] & \text{se } \texttt{VD}_\texttt{i}[\texttt{j}] > \texttt{perm\_VD}_\texttt{i}[\texttt{j}], \\ 0 & \text{caso contrário.} \end{cases}$$

- MVD – matriz de vetores de dependências atualizada pelo iniciador. A linha i de MVD contém o vetor de dependências de p_i, informação recebidas com a resposta ao bloqueio enviada por p_i. Quando o iniciador recebe mensagens de resposta ao bloqueio de todos os processos, deve verificar se as precedências foram estabelecidas no último intervalo de *checkpoints* dos processos. Assim, a matriz MVP é atualizada da seguinte maneira: para todo i, j com $i \neq j$

$$\texttt{MVP[i][j]} = \begin{cases} 0 & \text{se } \texttt{MVP[i][j]} = \texttt{1 e MVD[j][j]} > \texttt{MVD[i][j]}, \\ \texttt{MVP[i][j]} & \text{caso contrário.} \end{cases}$$

- MVP – matriz de vetores de participantes atualizada pelo iniciador. A linha i de MVP contém o vetor de participantes de p_i, informação recebida com a resposta ao bloqueio enviada por p_i. O iniciador verifica quais são os participantes de sua construção consistente da seguinte maneira:

  repita
  `MVP[`*ini*`] * MVP`
  até `que os valores de MVP[`*ini*`] não se alterem`

- respostas – vetor mantido pelo iniciador para detecção da terminação da construção consistente.

Início da construção consistente

Um processo p_{ini}, ao iniciar uma construção consistente, armazena um *checkpoint* provisório. Se p_{ini} não recebeu nenhuma mensagem no último intervalo de *checkpoints*, o *checkpoint* provisório é transformado em permanente e a construção consistente é encerrada. Caso contrário, p_{ini} fica bloqueado e faz um *broadcast* da mensagem de bloqueio.

Recepção da mensagem de bloqueio

Quando um processo recebe uma mensagem de bloqueio, fica bloqueado e envia o seu vetor de dependências VD e seu vetor de participantes VP ao iniciador por meio da mensagem de resposta ao bloqueio.

Recepção da mensagem de resposta ao bloqueio

O iniciador p_{ini} constrói as matrizes `MVD` e `MVP` a partir dos vetores `VD` e `VP`, respectivamente, recebidos de todos os processos. Após receber mensagens de resposta ao bloqueio de todos os processos, p_{ini} utiliza as matrizes `MVD` e `MVP` para selecionar os participantes de sua construção consistente. Após executar os cálculos descritos anteriormente, se entrada i do vetor $\texttt{MVP}_{ini}$ é igual a 1, então p_{ini} envia uma mensagem de requisição a p_i. Caso contrário, p_{ini} envia uma mensagem de liberação a p_i.

Recepção da mensagem de requisição

Todo processo que recebe uma mensagem de requisição, salva um *checkpoint* provisório e envia uma mensagem de resposta ao iniciador.

Recepção da mensagem de resposta

O iniciador, após receber uma mensagem de resposta de cada um dos participantes, passa para a última fase de sua construção consistente: transforma seu *checkpoint* provisório em permanente, envia uma mensagem de liberação para todos os participantes e volta a processar a aplicação.

Recepção da mensagem de liberação

Todo processo que recebe uma mensagem de liberação, transforma seu *checkpoint* provisório em permanente, fica desbloqueado voltando à sua computação normal.

A descrição do protocolo Broad-minimal é apresentada pelo Protocolo 4.1. Consideramos que todo procedimento deste algoritmo é executado de forma atômica.

Protocolo 4.1 Broad-minimal (declarações)

Variáveis do processo:
- VD ≡ vetor[0...$n-1$] de inteiros
- perm_VD ≡ vetor[0...$n-1$] de inteiros
- VP ≡ vetor[0...$n-1$] de bits
- MVD ≡ matriz[0...$n-1$][0...$n-1$] de inteiros
- MVP ≡ matriz[0...$n-1$][0...$n-1$] de bits
- bloqueado ≡ booleano
- pid ≡ inteiro

Tipos de mensagem:
- aplicação:
  - VD ≡ vetor[0...$n-1$] de inteiros
- bloqueio
- resposta ao bloqueio:
  - VD ≡ vetor[0...$n-1$] de inteiros
  - VP ≡ vetor[0...$n-1$] de bits
- requisição
- resposta
- liberação

Protocolo 4.1 Broad-minimal

Início:
  $\forall i$: VD[i] $\leftarrow$ 0, perm_VD[i] $\leftarrow$ 0
  $\forall i$: VP[i] $\leftarrow$ 0
  bloqueado $\leftarrow$ *falso*
  salvaCkpt(permanente)

Envio da mensagem da aplic. (m) para p_k:
  se (*não* bloqueado)
    m.VD $\leftarrow$ VD
    envia mensagem da aplicação (m)

Recepção da mensagem da aplic. (m) de p_k:
  se (*não* bloqueado)
    VD $\leftarrow$ max(m.VD, VD)
    processa mensagem da aplicação (m)

Início da construção consistente:
  salvaCkpt(provisório)
  MVD[pid] $\leftarrow$ VD
  $\forall i$: se (VD[i] > perm_VD[i])
      MVP[pid][i] $\leftarrow$ 1
    senão
      MVP[pid][i] $\leftarrow$ 0
  se ($\exists i \neq$ pid: MVP[pid][i] = 1)
    bloqueado $\leftarrow$ *verdadeiro*
    envia broadcast do bloqueio (bloq)
  senão
    transformaCkpt()

Recepção do bloqueio (bloq) de p_{ini}:
  bloqueado $\leftarrow$ *verdadeiro*
  rb.VD $\leftarrow$ VD
  $\forall i$: se (VD[i] > perm_VD[i])
      rb.VP[i] $\leftarrow$ 1
  envia resposta ao bloqueio (rb) para p_{ini}

Recepção da resp. ao bloqueio (rb) de p_k:
  MVD[k] $\leftarrow$ rb.VD
  MVP[k] $\leftarrow$ rb.VP
  VP[k] $\leftarrow$ 1
  se ($\forall l$: VP[l] = 1)
    $\forall i, j$: se ($i \neq j$ *e* MVP[i][j] = 1)
        se (MVD[j][j] > MVD[i][j])
          MVP[i][j] $\leftarrow$ 0
    VP $\leftarrow$ MVP[pid]
    faça
      aux $\leftarrow$ VP
      VP $\leftarrow$ VP $*$ MVP
    enquanto (VP $\neq$ aux)
    $\forall i \neq$ pid: se (VP[i] = 1)
        envia requisição (req) para p_i
      senão
        envia liberação (lib) para p_i
    se ($\forall i \neq$ pid: VP[i] = 0)
      transformaCkpt()

Recepção da requisição (req) de p_{ini}:
  salvaCkpt(provisório)
  envia resposta (res) para p_{ini}

Recepção da resposta (res) de p_k:
  respostas[k] $\leftarrow$ 1
  se (VP = respostas)
    transformaCkpt()
    $\forall i \neq$ pid: se (VP[i] = 1)
        envia liberação (lib) para p_i
    $\forall i$: VP[i] $\leftarrow$ 0, respostas[i] $\leftarrow$ 0

Recepção da liberação (lib) de p_{ini}:
  transformaCkpt()

salvaCkpt(tipo)
  armazena um checkpoint de acordo com tipo
  VD[pid] $\leftarrow$ VD[pid] + 1

transformaCkpt()
  se ($\exists$ checkpoint c provisório)
    transforma c em permanente
  perm_VD $\leftarrow$ c.VD
  bloqueado $\leftarrow$ *falso*

Exemplo

Uma possível execução deste protocolo é ilustrada pela Figura 4.10. Neste cenário, a construção consistente iniciada por p_2 é encerrada após p_2 salvar um *checkpoint*, pois p_2 não recebeu mensagens no último intervalo de *checkpoints* e portanto não tem participantes. A construção consistente de p_0 é iniciada quando p_0 salva um *checkpoint* e faz um *broadcast* da mensagem de bloqueio. Ao receber a mensagem resposta ao bloqueio de todos os processos, p_0 possui as matrizes de Vetores de Dependências `MVD` (Figura 4.11 (a)) e de Vetores de Participantes `MVP` (Figura 4.11 (b)).

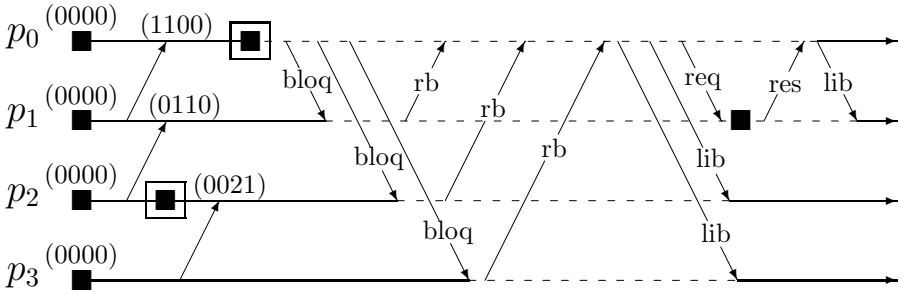


Figura 4.10: Exemplo de padrão gerado pelo protocolo Broad-minimal

Para obter um número minimal de processos participantes, o iniciador desconsidera as precedências estabelecidas antes do último intervalo de *checkpoints* de cada processo analisando a matriz `MVD`. No exemplo, temos `MVP[1][2]` $= 1$ e `MVD[2][2]` $>$ `MVD[1][2]`, o que indica que a dependência conhecida por p_1 foi estabelecida em um intervalo anterior ao último intervalo de *checkpoints* de p_2 e portanto, deve ser desconsiderada. Após essa verificação, a nova `MVP` do iniciador é como mostra a Figura 4.11 (c).

Para determinar o conjunto de seus participantes, p_0 multiplica recursivamente seu vetor de participantes `MVP[`ini`]` pela matriz de vetores de participantes `MVP` até que não haja mudança no vetor de participantes. Este cálculo feito pelo iniciador p_0 é descrito pela Figura 4.11 (d). Ao final da multiplicação das matrizes, temos como resultado que p_1 é participante da construção consistente iniciada por p_0 e portanto, p_0 envia uma mensagem de requisição a p_1 e envia mensagens de liberação aos outros processos.

Prova de Correção

Nesta seção, mostramos que o protocolo Broad-minimal é correto e é minimal.

$$\texttt{MVD} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(a) Matriz de Vetores de Dependências

$$\texttt{MVP} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(b) Matriz de Vetores de Participantes

$$\texttt{MVP} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(c) Após desconsiderar precedência

$$\begin{pmatrix} 1 & 1 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 \end{pmatrix}$$

(d) Cálculo dos participantes

Figura 4.11: Matrizes construídas pelo iniciador

Lema 1 *No protocolo Broad-minimal, caso o iniciador, após atualizar* `MVP`*, obtenha como resultado* `MVP[b][a] = 1`*, então* p_a *enviou uma mensagem* m *após o seu último checkpoint e* p_b *recebeu* m *no seu último intervalo de checkpoints.*

Prova: Vamos supor que c_a^α é o último *checkpoint* de p_a. No protocolo Broad-minimal, o processo iniciador recebe os vetores de dependências de todos os processos. O vetor $\texttt{VP}_b$ indica de quais processos p_b recebeu mensagem após o seu último *checkpoint*, ou seja, `MVP[b][a] = 1` se p_a enviou uma mensagem m e p_b recebeu m no seu último intervalo de *checkpoints*. Se c_a^α não fosse o último *checkpoint* de p_a, essa precedência seria desconsiderada pelo iniciador (`MVP[b][a]` $\leftarrow$ `0`) pois o índice do último *checkpoint* de p_a seria maior do que o índice de p_a conhecido por p_b (`MVD[a][a]` $>$ `MVD[b][a]`). Portanto, o *checkpoint* c_a^α é o último *checkpoint* de p_a. □

Lema 2 *No protocolo Broad-minimal,* `MVP[`ini`][a] = 1` *se, e somente se,* $c_a^\alpha \rightsquigarrow c_{ini}^\iota$*, onde* c_a^α *é o último checkpoint de* p_a *e* c_{ini}^ι *é o checkpoint provisório do iniciador* p_{ini}*.*

Prova: Seja c^{ι}_{ini} o *checkpoint* salvo pelo iniciador p_{ini} e seja c^{α}_{a} o último *checkpoint* salvo por p_a.

Necessidade ($\Rightarrow$): Vamos provar que se `MVP[`ini`][a] = 1`, então $c^{\alpha}_{a} \rightsquigarrow c^{\iota}_{ini}$. Vamos supor que `MVP[`$ini$`][a] = 1` após o passo k da multiplicação entre `MVP[`ini`]` e `MVP`.

Base ($k = 0$): Pelo Lema 1, p_a enviou uma mensagem m para p_{ini} após c^{α}_{a} e p_{ini} recebeu m durante o seu último intervalo de *checkpoints*. Assim, $c^{\alpha}_{a} \rightarrow c^{\iota}_{ini}$ e portanto $c^{\alpha}_{a} \rightsquigarrow c^{\iota}_{ini}$

Passo ($k > 0$): Suponha que após $k-1$ passos da multiplicação, o valor de `MVP[`ini`][b]` foi alterado para 1 e $c^{\beta}_{b} \rightsquigarrow c^{\iota}_{ini}$. No passo k, se `MVP[`ini`][a]` se torna 1, então, `MVP[b][a] = 1`. Portanto, p_a enviou uma mensagem para p_b após c^{α}_{a} e p_b recebeu m durante o seu último intervalo de *checkpoints*. Assim, podemos concluir que $c^{\alpha}_{a} \rightsquigarrow c^{\iota}_{ini}$.

Suficiência ($\Leftarrow$): Vamos provar que se $c^{\alpha}_{a} \rightsquigarrow c^{\iota}_{ini}$ então `MVP[`ini`][a] = 1`. Sabemos que se $c^{\alpha}_{a} \rightsquigarrow c^{\iota}_{ini}$ então existe uma seqüência de mensagens $m_1, \ldots, m_k$ tal que m_1 foi enviada por p_a após c^{α}_{a} e m_k foi recebida por p_{ini} antes de c^{ι}_{ini}.

Base ($k = 1$): m_1 foi enviada por p_a após c^{α}_{a} e recebida por p_{ini} antes de armazenar c^{ι}_{ini}. Portanto, p_{ini} anotou no seu vetor de participantes que recebeu uma mensagem de p_a no seu último intervalo e a matriz `MVP[`ini`][a] = 1`.

Passo ($k > 1$): Suponha que se m_{k-1} foi enviada por p_b após c^{β}_{b} e m_1 foi recebida por p_{ini} antes de c^{ι}_{ini} ($c^{\beta}_{b} \rightsquigarrow c^{\iota}_{ini}$), então `MVP[`$ini$`][b] = 1`. Se m_k foi enviada por p_a após c^{α}_{a} e foi recebida por p_b no seu último intervalo de *checkpoints* ($c^{\alpha}_{a} \rightsquigarrow c^{\iota}_{ini}$), então p_b anotou no seu vetor de participantes a recepção de m. Neste caso, `MVP[b][a] = 1` e esta precedência não é desconsiderada pois c^{α}_{a} é o último *checkpoint* salvo por p_a e `MVD[a][a] = MVD[b][a]`. Como `MVP[`ini`][b] = 1`, a multiplicação de `MVP[`ini`]` com `MVP` resultará em `MVP[`ini`][a] = 1`. □

Sabemos que a união dos *checkpoints* iniciais representa um *checkpoint* global consistente. Desta forma, para provar que o protocolo é correto, basta provar que após a execução de uma construção consistente, a união dos últimos *checkpoints* de cada processo formará um novo *checkpoint* global consistente.

Teorema 3 *Suponha que uma invocação do protocolo Broad-minimal foi iniciada a partir de um checkpoint global consistente. Imediatamente após o fim desta construção consistente, a união dos últimos checkpoints de cada processo formará um novo checkpoint global consistente.*

Prova: Sabemos que os últimos *checkpoints* dos processos que não armazenaram *checkpoints* durante a construção consistente são consistentes. Pelo Lema 2, se existe *zigzag path* entre o último *checkpoint* de um processo p_a e o *checkpoint* do iniciador, então `MVP[`ini`][a] = 1` e portanto, p_{ini} envia uma mensagem de requisição para p_a e p_a salva um

checkpoint durante sua construção consistente. Além disso, não existe *zigzag path* entre o último *checkpoint* c_b^β de um processo não participante p_b e um processo participante dessa construção consistente, pois senão c_b^β z-precede o *checkpoint* do iniciador e p_b deveria também ser participante. Podemos concluir portanto que não existe *zigzag path* entre o último *checkpoint* de cada processo formando assim um *checkpoint* global consistente. □

Teorema 4 *O protocolo Broad-minimal é minimal.*

Prova: Para provar que o protocolo Broad-minimal é minimal, basta provar que um processo p_a armazena um *checkpoint* durante a construção consistente de p_{ini} apenas se existe uma *zigzag path* entre o último *checkpoint* de p_a salvo antes da construção consistente de p_{ini} e o *checkpoint* do iniciador p_{ini}. No protocolo Broad-minimal, esta é a condição para p_a ser participante da construção consistente de p_{ini} (Lema 2). Portanto, o protocolo Broad-minimal é minimal. □

Teorema 5 *O protocolo Broad-minimal termina corretamente uma construção consistente.*

Prova: O iniciador envia uma mensagem de requisição para todos os processos participantes de sua construção consistente e uma mensagem de liberação aos outros processos. Cada participante, após armazenar um *checkpoint*, envia uma mensagem de resposta ao iniciador. O iniciador, ao receber mensagens de respostas de todos os participantes envia uma mensagem de liberação a cada participante. Portanto, todos os processos da aplicação recebem em algum momento uma mensagem de liberação e voltam à sua computação normal. □

4.2 Iniciadores Concorrentes

A maioria dos protocolos minimais são propostos para permitir um único iniciador a cada instante sendo extensíveis, por meio de diferentes estratégias, para permitir a presença de iniciadores concorrentes. O protocolo de Koo e Toueg, por exemplo, aborta uma das construções consistentes toda vez que mais de um iniciador inicia sua construção consistente concorrentemente [20]. Leu e Bhargava propõem a construção de um *checkpoint* global consistente a partir da intersecção dos *checkpoints* salvos pelos processos envolvidos [22]. Outra opção para permitir iniciadores concorrentes é a implementação do protocolo que obtém como resultado um *checkpoint* global consistente a partir da união dos *checkpoints* salvos durante as construções consistentes concorrentes [29].

As características descritas para os protocolos minimais com um único iniciador também são válidas na presença de iniciadores concorrentes. Nesta Seção, avaliamos e descrevemos algumas considerações adicionais que devem ser respeitadas pelos protocolos síncronos minimais na presença de iniciadores concorrentes.

4.2.1 Características

Na presença de iniciadores concorrentes, é interessante identificar de qual construção consistente uma determinada mensagem de controle faz parte. Na fase de requisições, se um processo recebe duas mensagens de controle de duas construções consistentes concorrentes, o processo deve salvar apenas um *checkpoint*, porém deve propagar as mensagens de requisição para cada uma das construções consistentes. Por exemplo, no cenário da Figura 4.12, o processo p_2 recebe uma mensagem de requisição de p_1 e propaga a mensagem de requisição para p_3. Posteriormente, p_2 recebe uma mensagem de requisição de p_0 e não salva um *checkpoint* pois o seu último *checkpoint* não é conhecido por p_0. Nesse momento, se p_2 apenas envia uma mensagem de resposta para p_0, p_0 pode concluir sua construção consistente antes de ter obtido um *checkpoint* global consistente, pois a mensagem de requisição enviada por p_2 pode não ter sido recebida por p_3.

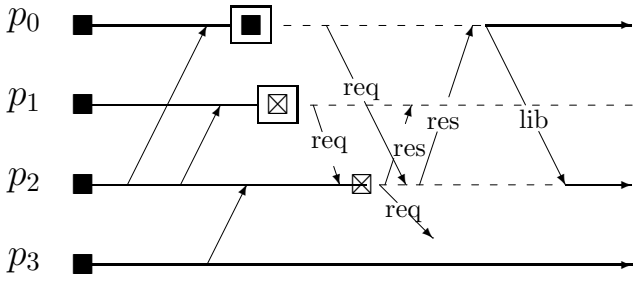


Figura 4.12: Inconsistência – p_2 não repropaga mensagem de requisição

Para evitar esse problema, p_2, ao receber a mensagem de requisição de p_0, deve repropagar a mensagem de requisição para p_3 incluindo p_3 como participante da construção consistente de p_0 e portanto p_0 não deve encerrar sua construção consistente antes de receber uma mensagem de requisição de p_3 garantindo o *checkpoint* global consistente (Figura 4.13). Note que não basta p_2 enviar uma mensagem de resposta para p_0 incluindo p_3 como participante sem enviar uma mensagem de requisição para p_3, pois p_3 também pode incluir outros participantes para essa construção consistente.

Analisando o cenário descrito acima, sabemos que se um processo p_a recebe duas mensagens de requisição de diferentes construções consistentes, este receberá também duas mensagens de liberação. Notamos que, como o processo p_a repropaga as mensagens de requisição, quando p_a recebe uma mensagem de liberação da construção consistente iniciada por p_{ini}, todos os processos que p_a incluiu como participantes já enviaram mensagens de resposta para p_{ini} e portanto, possuem *checkpoints* consistentes com o *checkpoint* salvo

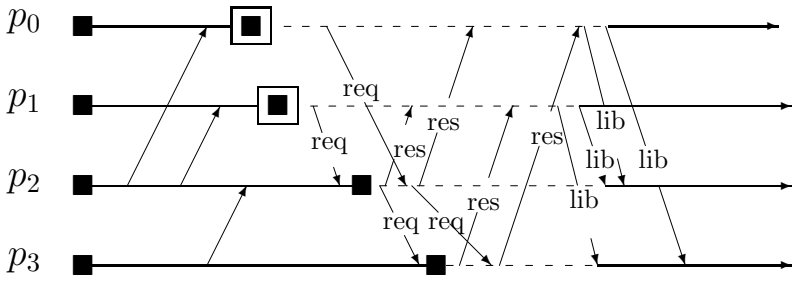


Figura 4.13: Repropagação das mensagens de requisição

por p_a. Portanto, p_a não necessita aguardar por todas as mensagens de liberação de todas as construções consistentes que participa para ficar desbloqueado. Na Figura 4.13, os processos p_2 e p_3 ficam desbloqueados ao receberem a mensagem de liberação de p_1, ou seja, antes de receberem a mensagem de liberação de p_0. A única precaução que deve ser tomada na fase de liberações é na recepção de mensagens atrasadas. Se um processo recebe uma mensagem de liberação que indica a conclusão de uma construção consistente que inclui um *checkpoint* anterior ao seu último *checkpoint*, então essa mensagem deve ser ignorada.

4.2.2 Minimalidade no Número de *Checkpoints*

A minimalidade no número de *checkpoints* pode ser garantida se, na ocorrência de execuções concorrentes em um instante de tempo, cada processo envolvido salvar no máximo, um *checkpoint*. O cenário da Figura 4.14 ilustra a execução de duas construções consistentes iniciadas por p_0 e p_3. Apenas o processo p_1 é participante da construção consistente iniciada por p_0 e portanto, os *checkpoints* salvos durante essa construção consistente e os *checkpoints* iniciais de p_2, p_3 e p_4 formam o *checkpoint* global consistente construído por p_0. O processo p_3 inicia uma construção consistente concorrente com a construção consistente de p_0. O processo p_3 envia uma mensagem de requisição para p_2 que salva um *checkpoint* e envia uma mensagem de requisição para p_1. Quando p_1 recebe a mensagem de requisição de p_2, p_1 está bloqueado (já está participando de outra construção consistente) e possui um *checkpoint* não conhecido por p_2. Portanto, p_1 pode aproveitar o seu último *checkpoint* salvo para fazer parte também da construção consistente iniciada por p_3. Neste caso, p_1 apenas envia uma mensagem de resposta ao iniciador p_3. Note que, como vimos na seção anterior, esse seria o caso em que um processo deve repropagar mensagens de requisição, porém, p_1 não necessitou enviar nenhuma mensa-

gem desse tipo. Outra observação é que mesmo com as duas construções consistentes, o processo p_4 não necessitou salvar *checkpoint*, ou seja, apenas um número minimal de processos salvam *checkpoints* em cada construção consistente em execução. Quando ambas as construções consistentes terminam, a união dos *checkpoints* globais consistentes construídos pelas construções consistentes forma um *checkpoint* global consistente.

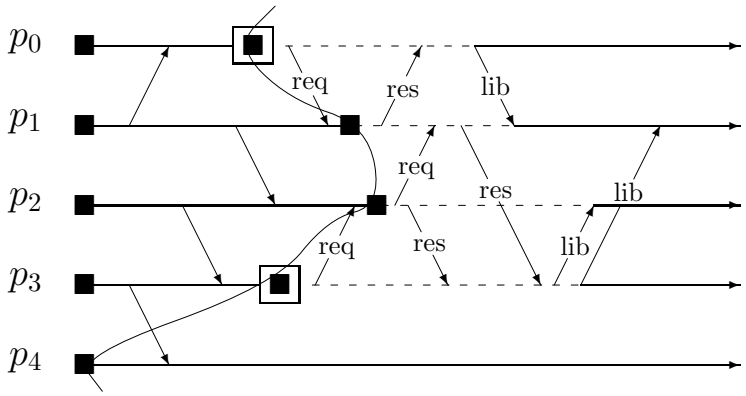


Figura 4.14: Minimalidade na presença de iniciadores concorrentes

4.2.3 Protocolo VD-minimal

O protocolo VD-minimal é um protocolo síncrono minimal baseado na abordagem em níveis e permite iniciadores concorrentes [35]. Este protocolo utiliza vetores de dependências para rastrear as precedências entre os *checkpoints* do sistema e garantir um número minimal de *checkpoints* na construção de *checkpoints* globais consistentes. Além disso, utilizamos um novo mecanismo de detecção de terminação que requer um número menor de mensagens de controle comparado aos protocolos similares existentes na literatura [20, 22]. O mecanismo de terminação descrito nesta tese corrige um problema encontrado no artigo [35].

Descrição do Protocolo

O protocolo VD-minimal utiliza vetores de dependências para capturar as precedências causais entre os *checkpoints* dos processos. Quando um processo inicia sua construção consistente, verifica para quais processos deve propagar mensagens de requisição de acordo com as precedências causais estabelecidas no seu último intervalo de *checkpoints*. As mensagens de requisição são propagadas pelos processos participantes de forma semelhante ao iniciador. Todo processo que recebe uma mensagem de requisição pode salvar um

checkpoint provisório e deve enviar uma mensagem de resposta ao iniciador. O iniciador, ao receber uma mensagem de resposta de cada participante da construção consistente, encerra a execução do protocolo enviando uma mensagem de liberação aos participantes. Esta implementação de detecção de terminação necessita de $\mathcal{O}(\alpha.n)$ mensagens de respostas, enquanto outros mecanismos utilizados em protocolos semelhantes requerem $\mathcal{O}(\alpha.n^2)$ mensagens na fase de respostas, onde α representa o número de construções consistentes. A seguir, descrevemos cada procedimento do protocolo.

Variáveis do processo

As variáveis utilizadas por este protocolo são descritas a seguir:

- $\texttt{VD}_i$ – vetor de dependências mantido pelo processo p_i para capturar as precedências causais entre *checkpoints*. Este vetor é propagado com as mensagens da aplicação e quando p_i recebe uma mensagem, seu vetor é atualizado da seguinte maneira: $\texttt{VD}_i$ `= max(m.VD,` $\texttt{VD}_i$`)`
- $\texttt{perm\_VD}_i$ – vetor salvo com o *checkpoint* permanente mantido pelo processo p_i.
- $\texttt{VP}_i$ – vetor de participantes que indica se p_i recebeu uma mensagem com informação de novo *checkpoint* dos processos durante o seu intervalo de *checkpoints* corrente. Este vetor é atualizado da seguinte maneira:

$$\texttt{VP}_\texttt{i}[\texttt{j}] = \begin{cases} \texttt{VD}_\texttt{i}[\texttt{j}] & \text{se } \texttt{VD}_\texttt{i}[\texttt{j}] > \texttt{perm\_VD}_\texttt{i}[\texttt{j}], \\ 0 & \text{caso contrário.} \end{cases}$$

- `respostas` – vetor mantido pelo iniciador para detecção da terminação da construção consistente.
- `inics` – vetor que indica o índice dos processos iniciadores. A entrada $\texttt{inics}_i[i]$ indica o índice de p_i quando p_i iniciou sua última construção consistente.

Início da construção consistente

Um processo p_{ini}, ao iniciar uma construção consistente $\mathcal{C}$, armazena um *checkpoint* provisório, incrementa o índice do seu *checkpoint* e atualiza o seu vetor de participantes `VP`. Se p_{ini} não possui participantes, $\mathcal{C}$ é encerrada. Caso contrário, p_{ini} envia uma mensagem de requisição para cada um dos participantes e fica bloqueado. Cada mensagem de requisição propaga o índice do iniciador p_{ini} (`req.iind`) e seu `VP`.

Recepção da mensagem de requisição

Ao receber uma mensagem de requisição de p_k, o processo p_i:

- se não está bloqueado e p_k conhece o índice atual de p_i, então p_i armazena um *checkpoint* provisório, fica bloqueado, e propaga mensagens de requisição aos seus participantes não conhecidos por p_k.
- se está bloqueado e p_k conhece o índice do seu último *checkpoint* permanente, então p_i já participa de uma construção consistente. Neste caso, se p_i não conhece o índice do iniciador de $\mathcal{C}$, então p_i deve repropagar mensagens de requisição para os seus participantes não conhecidos por p_k (Seção 4.2.1).
- para qualquer outro caso, p_i apenas atualiza a informação do iniciador de $\mathcal{C}$, se necessário.

Após analisar a necessidade em armazenar um *checkpoint* provisório, p_i envia uma mensagem de resposta ao iniciador p_{ini} com o seu `VP`.

Recepção da mensagem de resposta

A terminação do protocolo é garantida utilizando-se dois vetores: `VP` (que indica o índice do *checkpoint* dos processos participantes da construção consistente) e `respostas` (que indica o índice do *checkpoint* dos processos que já enviaram mensagens de resposta ao iniciador). Quando p_{ini} recebe uma mensagem de resposta de p_k, p_{ini} inclui no seu vetor `VP` todos os participantes conhecidos por p_k e anota que já recebeu uma mensagem de resposta de p_k no vetor `respostas`. Quando o vetor `respostas` tem valores maiores ou iguais ao vetor `VP` para todas as entradas, todos os participantes já armazenaram os *checkpoints* com o índice esperado e p_{ini} recebeu mensagens de respostas de todos eles. Então, p_{ini} transforma se *checkpoint* provisório em permanente, envia mensagens de liberação para todos os processos participantes, atualiza o vetor `perm_VD` e fica desbloqueado.

Recepção da mensagem de liberação

Todo processo que recebe uma mensagem de liberação, transforma seu *checkpoint* provisório em permanente e atualiza as variáveis de forma semelhante ao iniciador e fica desbloqueado.

A descrição do protocolo VD-minimal é apresentada pelo Protocolo 4.2. Consideramos que todo procedimento deste algoritmo é executado de forma atômica. A versão simplificada deste protocolo para apenas um único iniciador a cada instante de tempo foi originalmente chamada de VR-minimal simples [35]. Este código apresenta um erro que foi corrigido na Seção B.4.

Protocolo 4.2 VD-minimal (declarações)

Variáveis do processo:
VD ≡ vetor[0 . . . n − 1] de inteiros
perm_VD ≡ vetor[0 . . . n − 1] de inteiros
VP ≡ vetor[0 . . . n − 1] de inteiros
respostas ≡ vetor[0 . . . n − 1] de inteiros
bloqueado ≡ booleano
inics ≡ vetor[0 . . . n − 1] de inteiros
inic ≡ booleano
pid ≡ inteiro

Tipos de mensagem:
aplicação:
VD ≡ vetor[0 . . . n − 1] de inteiros
requisição:
ipid ≡ inteiro
iind ≡ inteiro
VP ≡ vetor[0 . . . n − 1] de inteiros
resposta:
VP ≡ vetor[0 . . . n − 1] de inteiros
liberação
ind ≡ inteiro

Exemplos

Na Figura 4.15, não existem participantes para o iniciador p_0 e sua construção consistente é encerrada sem a necessidade de propagar mensagens de controle. Na mesma figura, o iniciador p_3 grava um *checkpoint* provisório e envia mensagens de requisição para os seus participantes p_1 e p_2. O processo p_1, ao receber uma mensagem de requisição de p_3, grava um *checkpoint* provisório pois `req.VP[1] = VD`$_1$`[1]`, propaga uma mensagem requisição para o seu participante p_0 e envia uma mensagem de resposta para o iniciador p_3. De forma semelhante, o processo p_2 salva um *checkpoint*, envia uma mensagem de requisição para p_0 e uma mensagem de resposta para p_3. O processo p_0, ao receber a mensagem de requisição de p_1, apenas atualiza o índice do iniciador pois `req.VP[0]` $<$ `VD`$_0$`[0]` e envia uma mensagem de resposta ao iniciador p_3. A mensagem de requisição enviada por p_2 é ignorada por p_0 pois p_0 conhece o índice do iniciador naquele instante, o que implica que p_0 já enviou uma mensagem de resposta ao iniciador. O iniciador, após receber respostas de todos os processos participantes, retorna-lhes uma mensagem de liberação.

Na Figura 4.16, ilustramos a execução do protocolo VD-minimal na presença de iniciadores concorrentes. Neste cenário, p_0 salva um *checkpoint* ao receber uma mensagem de requisição de p_1. Quando p_1 recebe a mensagem de requisição de p_2, p_1 repropaga a mensagem de requisição para p_0. Já quando p_1 recebe a mensagem de requisição de p_3, p_1 apenas envia uma mensagem de resposta ao iniciador p_3, pois o último *checkpoint* conhecido por p_3 é anterior ao último *checkpoint* salvo antes das construções consistentes correntes de p_1. Neste caso, p_1 descarta a mensagem de liberação de p_3 e fica bloqueado aguardando pela mensagem de liberação de p_2, ou pela mensagem de resposta de p_0.

Protocolo 4.2 VD-minimal na presença de iniciadores concorrentes

Início:
  $\forall i$: VD[i] $\leftarrow$ 0
  $\forall i$: perm_VD[i] $\leftarrow$ 0
  $\forall i$: respostas[i] $\leftarrow$ 0
  $\forall i$: inics[i] $\leftarrow$ 0
  bloqueado $\leftarrow$ *falso*
  salvaCkpt(permanente)

Envio da mensagem da aplic. (m) para p_k:
  se (*não* bloqueado)
    m.VD $\leftarrow$ VD
    envia mensagem da aplicação (m)

Recepção da mensagem da aplic. (m) de p_k:
  se (*não* bloqueado)
    VD $\leftarrow$ max(m.VD, VD)
    processa mensagem da aplicação (m)

Início da construção consistente:
  bloqueado $\leftarrow$ *verdadeiro*
  salvaCkpt(provisório)
  inics[pid] $\leftarrow$ VD[pid]
  verificaVP()
  $\forall i \neq$ pid: se (VP[i] $\neq$ 0)
      req.ipid $\leftarrow$ pid
      req.iind $\leftarrow$ VD[pid]
      req.VP $\leftarrow$ VP
      envia requisição (req) para p_i
  se ($\forall i \neq$ pid: VP[i] = 0)
    transformaCkpt()

Recepção da requisição (req) de p_k:
  se (*não* bloqueado *e* req.VP[pid] = VD[pid]
    ou bloqueado *e* req.VP[pid] = perm_VD[pid])
    se (*não* bloqueado)
      bloqueado $\leftarrow$ *verdadeiro*
      salvaCkpt(provisório)
      verificaVP()
    se (inics[req.ipid] < req.iind)
      inics[req.ipid] $\leftarrow$ req.iind
      VP[i] $\leftarrow$ max(VP, req.VP)
      $\forall i$: se (VP[i] $\neq$ 0 *e* req.VP[i] = 0)
          p_req.ipid $\leftarrow$ req.ipid
          p_req.iind $\leftarrow$ req.iind
          p_req.VP $\leftarrow$ VP
          envia requisição (p_req) para p_i
      res.VP $\leftarrow$ VP
      envia resposta (res) para $p_{req.ipid}$
  senão
    se (inics[req.ipid] < req.iind)
      inics[req.ipid] $\leftarrow$ req.iind
      res.VP $\leftarrow$ req.VP
      envia resposta (res) para $p_{req.ipid}$

Recepção da resposta (res) de p_k:
  se (bloqueado)
    se (res.VP[pid] = VD[pid])
      VP $\leftarrow$ max(VP, res.VP)
      respostas[k] $\leftarrow$ res.VP[k]
      se ($\forall i \neq$ pid: respostas[i] $\geq$ VP[i])
        transformaCkpt()
        $\forall i$: se (VP[i] $\neq$ 0 *e* $i \neq$ pid)
            lib.ind $\leftarrow$ VP[i]
            envia liberação (lib) para p_i
        $\forall i$: respostas[i] $\leftarrow$ 0

Recepção da liberação (lib) de p_k:
  se (bloqueado)
    se (lib.ind = VD[pid])
      transformaCkpt()

salvaCkpt(tipo)
  armazena um checkpoint de acordo com tipo
  VD[pid] $\leftarrow$ VD[pid] + 1

transformaCkpt()
  transforma o *checkpoint* provisório
    em permanente
  perm_VD $\leftarrow$ VD
  bloqueado $\leftarrow$ *falso*

verificaVP()
  $\forall i$: se (VD[i] > perm_VD[i])
      VP[i] $\leftarrow$ VD[i]
    senão
      VP[i] $\leftarrow$ 0

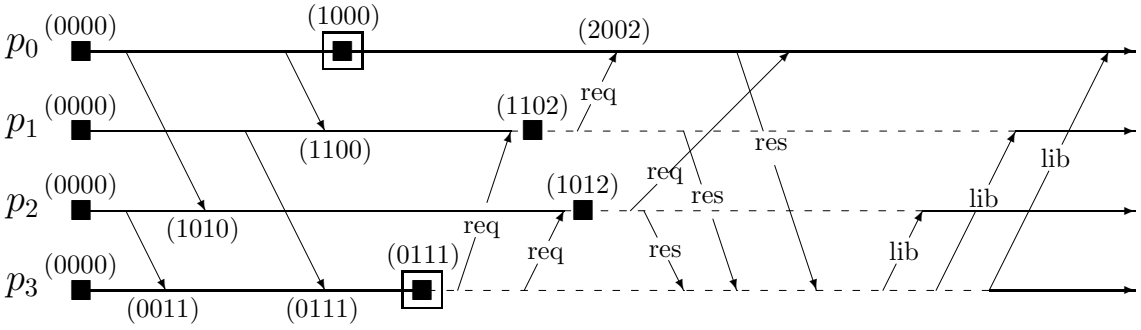


Figura 4.15: Exemplo de padrão gerado pelo protocolo VD-minimal

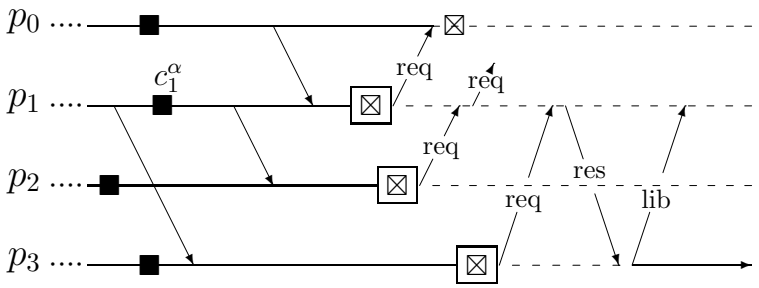


Figura 4.16: Protocolo VD-minimal na presença de iniciadores concorrentes

Prova de Correção

Lema 3 *Se não existe nenhuma construção consistente na fase de liberações, então o conjunto dos checkpoints permanentes dos processos forma um checkpoint global consistente.*

Prova: Sabemos que o *checkpoint* inicial é um *checkpoint* permanente. Pelo protocolo VD-minimal, um *checkpoint* permanente é removido somente quando um novo *checkpoint* permanente é salvo em memória estável, o que implica que todo processo possui um *checkpoint* permanente. Vamos supor que c_a^α é o *checkpoint* permanente de p_a e c_b^β é o *checkpoint* permanente de p_b e $c_a^\alpha \rightarrow c_b^\beta$. Sabemos também que p_b salvou o *checkpoint* c_b^β durante uma construção consistente $\mathcal{C}$. Portanto, p_b é participante de $\mathcal{C}$ e quando salvou o *checkpoint* c_b^β como provisório, propagou uma mensagem de requisição para p_a, pois p_b capturou informações sobre c_a^α no seu último intervalo de *checkpoints*. Portanto, p_a também é um processo participante de $\mathcal{C}$. Mas como não há construção consistente na fase de liberações, $\mathcal{C}$ está encerrada e portanto, p_a deveria ter salvo um *checkpoint* $c_a^{\alpha+1}$

posterior a c_a^{α} durante $\mathcal{C}$ e $c_a^{\alpha+1}$ deve ser um *checkpoint* consistente com c_b^{β}. □

Teorema 6 *O protocolo VD-minimal garante que, a qualquer momento, existe um checkpoint global consistente formado por checkpoints estáveis.*

Prova: Pelo Lema 3, sabemos que se nenhuma construção consistente está em andamento, então os *checkpoints* permanentes dos processos formam um *checkpoint* global consistente. Vamos então supor que existe pelo menos uma liberação pendente, ou seja, o processo p_a não recebeu a mensagem de liberação e possui um *checkpoint* permanente c_a^{α} inconsistente com os *checkpoints* permanentes dos demais processos. Mas se p_a é participante de uma construção consistente $\mathcal{C}$ em fase de liberações, então p_a possui um *checkpoint* provisório consistente com os *checkpoints* salvos durante $\mathcal{C}$. Portanto, para todo processo que ainda não recebeu mensagem de liberação, basta trocar o *checkpoint* permanente pelo provisório correspondente. □

Teorema 7 *Na presença de um único iniciador a cada instante de tempo, o protocolo VD-minimal é minimal.*

Prova: Seja c_{ini}^{ι} o *checkpoint* salvo pelo iniciador p_{ini} e seja c_a^{α} o último *checkpoint* de p_a armazenado antes da construção consistente $\mathcal{C}$ de p_{ini}. Pela Definição 12, um protocolo minimal induz um processo p_a a salvar o *checkpoint* $c_a^{\alpha+1}$ se, e somente se, existe uma *zigzag path* de c_a^{α} a c_{ini}^{ι} ($c_a^{\alpha} \rightsquigarrow c_{ini}^{\iota}$). Pelo Lema 3, sabemos que se existe uma *zigzag path* entre o último *checkpoint* de p_a c_a^{α} a c_{ini}^{ι}, então p_a armazena o *checkpoint* $c_a^{\alpha+1}$ durante $\mathcal{C}$, pois quando $\mathcal{C}$ encerra, os *checkpoints* salvos durante $\mathcal{C}$ são consistentes. Para provar que o protocolo VD-minimal é minimal, basta provar que apenas os processos cujo último *checkpoint* forma uma *zigzag path* com o *checkpoint* do iniciador armazenam *checkpoint* durante $\mathcal{C}$. Por contradição, vamos supor que p_b salva um *checkpoint* $c_b^{\beta+1}$ durante $\mathcal{C}$, mas $c_b^{\beta} \not\rightsquigarrow c_{ini}^{\iota}$. Vamos supor também que p_a armazena $c_a^{\alpha+1}$ durante $\mathcal{C}$. Se um processo p_b é incluído como participante quando p_a salva $c_a^{\alpha+1}$, então uma mensagem m foi enviada por p_b e recebida por p_a antes de $c_a^{\alpha+1}$. Se m foi enviada por p_b após o armazenamento de c_b^{β}, então $c_b^{\beta} \rightsquigarrow c_{ini}^{\iota}$. Caso contrário, a informação conhecida por p_a sobre p_b é menor que β e portanto, p_b não salva um *checkpoint* durante $\mathcal{C}$. □

O protocolo é encerrado quando todos os participantes da construção consistente recebem uma mensagem de liberação do iniciador. O iniciador por sua vez, só envia mensagens de liberação quando todos os participantes enviam-no mensagens de resposta. O iniciador detecta que a construção consistente pode ser encerrada quando seu vetor de respostas possui índices maiores ou iguais aos conhecidos pelos participantes. Sabemos que todo processo participante envia uma mensagem de resposta ao iniciador. Para provar que o protocolo termina corretamente basta provar que o iniciador conhece todos os participantes de sua construção consistente antes de enviar as mensagens de liberação.

Teorema 8 *O protocolo VD-minimal termina corretamente uma construção consistente.*

Prova: Suponha que o iniciador p_{ini} encerrou sua construção consistente antes do participante p_a salvar um *checkpoint* consistente com o *checkpoint* de p_{ini}. O processo p_a portanto recebeu uma mensagem de requisição de um processo p_b que é participante conhecido por p_{ini}. Então, p_{ini} recebe uma mensagem de resposta de p_b antes de encerrar a construção consistente. Mas a recepção da mensagem de resposta de p_b faz com que p_{ini} inclua no seu vetor de participantes o índice do *checkpoint* de p_a conhecido por p_b. Neste caso, p_{ini} não passa para a fase de liberações enquanto não recebe uma mensagem de resposta de p_a com a informação de um índice de *checkpoint* maior ou igual ao índice conhecido por p_b. □

4.3 Sumário

Nos protocolos síncronos, o armazenamento de um *checkpoint* requisitado pela aplicação implica na construção de um novo *checkpoint* global consistente mais recente até aquele momento formado por *checkpoints* estáveis. Em aplicações com um número excessivo de troca de mensagens, pode haver comunicação de todos os processos para todos os processos durante um intervalo de *checkpoints*. Neste caso, todos os processos deverão salvar *checkpoints* durante uma construção consistente. No entanto, se só um subgrupo de processos se comunicou com o processo iniciador do protocolo síncrono no seu último intervalo de *checkpoints*, o novo *checkpoint* global consistente pode ser construído mesmo que apenas os processos desse subgrupo armazenem *checkpoints* durante a construção consistente, pois os outros processos já possuem *checkpoints* estáveis consistentes com o *checkpoint* salvo pelo iniciador. Quando somente um número minimal de processos é induzido a armazenar *checkpoints* durante uma construção consistente, o protocolo é dito síncrono minimal.

Os protocolos minimais reduzem o custo de armazenamento de *checkpoints*, porém precisam ser bloqueantes, ou seja, os processos envolvidos devem permanecer bloqueados durante uma construção consistente. Na literatura, existem duas abordagens para a construção de protocolos síncronos minimais: (i) em níveis e; (ii) *broadcast*. Na abordagem em níveis, o iniciador envia mensagens de requisição a um subgrupo de participantes que recursivamente propagam mensagens de requisição para outros processos até que todos os participantes da construção consistente recebam a mensagem de requisição. Notamos na literatura, um esforço em tentar reduzir o número de mensagens de controle trocadas durante uma construção consistente nos protocolos com a abordagem em níveis [20, 22, 35]. Na abordagem *broadcast*, todos os processos são bloqueados num primeiro momento até que o iniciador decida quais os processos são participantes de sua construção global. Esta

abordagem tem como objetivo a redução do tempo de bloqueio dos processos durante uma construção consistente [8].

Os protocolos minimais requerem a propagação de informação de controle com as mensagens da aplicação e mensagens de controle adicionais durante uma construção consistente. Para reduzir o custo nas mensagens da aplicação, alguns protocolos propostos na literatura anotam a recepção de mensagens, mas não o intervalo de origem dessas mensagens. Porém, provamos que estes protocolos não possuem informação suficiente para serem minimais [35, 36]. Notamos que a redução no número de mensagens de controle trocadas durante uma construção consistente depende da maneira como as informações de dependência entre *checkpoints* são capturadas (afeta diretamente na maneira como as mensagens de requisição são propagadas) e do mecanismo de detecção de terminação do escolhido (determina para quais processos as mensagens de resposta e liberação devem ser enviadas).

Neste capítulo, propomos e descrevemos dois protocolos síncronos minimais: (i) Broad-minimal, baseado na abordagem *broadcast* e; (ii) VD-minimal, baseado na abordagem em níveis. Em nosso conhecimento, o único protocolo síncrono minimal com abordagem *broadcast* é o Broad-minimal. A idéia do *broadcast* na construção de protocolos síncronos minimais foi proposta originalmente por Cao e Singhal [8]. Porém, o protocolo proposto por eles, apesar de construir *checkpoints* globais consistentes, não é minimal, conforme demonstrado neste capítulo.

O protocolo VD-minimal utiliza vetores de dependências para capturar as dependências entre *checkpoints* e garantir a minimalidade no número de *checkpoint* de uma construção consistente. Além disso, propomos um novo mecanismo de detecção de terminação que necessita propagar um vetor de inteiros, mas reduz o número de mensagens de respostas para $\mathcal{O}(\alpha.n)$, onde α representa o número de construções consistentes. Este protocolo permite a presença de iniciadores concorrentes.

Capítulo 5

Protocolos de *Checkpointing* Síncronos Não-Bloqueantes

O problema de baixo desempenho dos protocolos minimais causado pelo bloqueio da aplicação foi amenizado pelos protocolos de *checkpointing* síncronos não-bloqueantes que permitem o progresso da computação durante uma construção consistente. O relaxamento da condição de bloqueio impede a construção de protocolos minimais [10]. Uma solução para este problema é o uso de *checkpoints* mutáveis que podem ser salvos em memória não-estável com o objetivo de garantir a minimalidade no número de *checkpoints* em memória estável [8]. Notamos porém que esta minimalidade está diretamente relacionada ao momento em que os *checkpoints* mutáveis são armazenados e mostramos que, mesmo na presença de um único iniciador a cada instante de tempo, é impossível garantir o número mínimo de *checkpoints* estáveis durante toda a execução da aplicação. Além disso, provamos que é impossível desenvolver um protocolo síncrono não-bloqueante que salva um número minimal de *checkpoints* estáveis na presença de iniciadores concorrentes.

Com base na análise realizada, propomos dois novos protocolos síncronos não-bloqueantes baseados em protocolos quase-síncronos de classes distintas: RDT-NBS baseado no protocolo FDAS (Seção 3.2.2) e BCS-NBS baseado no protocolo BCS (Seção 3.2.3). Esses protocolos permitem a presença de iniciadores concorrentes, reduzem o número de *checkpoints* estáveis a cada construção consistente, mas diferem no mecanismo de armazenamento de *checkpoints* mutáveis. Um experimento inicial de simulação foi realizado por meio da implementação desses protocolos com o uso do ChkSim [43].

As próximas seções estão divididas como a seguir. A Seção 5.1 descreve as características de um protocolo síncrono não-bloqueante. A Seção 5.2 discute a questão da minimalidade no número de *checkpoints* para esta classe de protocolos. Em seguida, descrevemos o protocolo RDT-NBS na Seção 5.3, o BCS-NBS na Seção 5.4 e os resultados de simulação são apresentados na Seção 5.5.

5.1 Características

Um *checkpoint* básico é requisitado no momento de interesse da aplicação. Se um protocolo de *checkpointing* síncrono permite apenas um único iniciador a cada instante, um processo poderá não ter permissão para salvar o seu *checkpoint* no momento que necessita por existir uma outra construção consistente em andamento. Além disso, todo protocolo que permite apenas um único iniciador deve implementar um mecanismo para controlar qual processo pode iniciar uma construção consistente e determinar quando todas as mensagens de controle enviadas durante essa execução foram entregues. Nesta seção, analisamos as características dos protocolos síncronos não-bloqueantes que reduzem o número de *checkpoints* salvos durante uma construção consistente na presença de um único iniciador a cada instante de tempo e de iniciadores concorrentes.

5.1.1 Único Iniciador

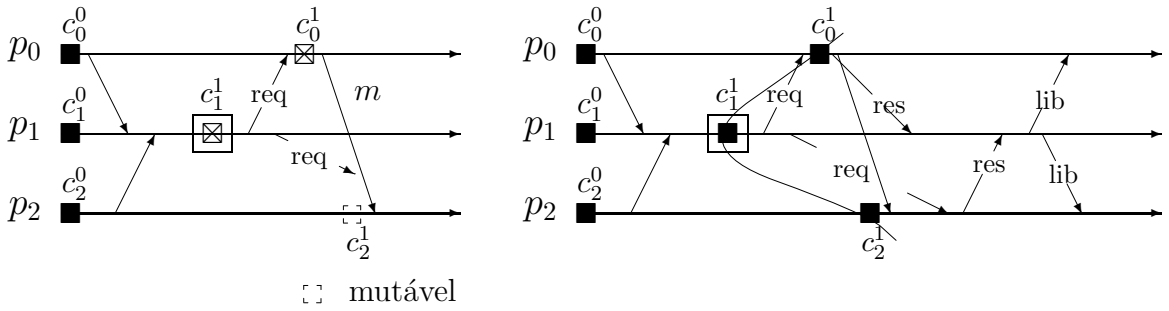
Os principais aspectos que devem ser considerados ao se desenvolver protocolos síncronos não-bloqueantes com um único iniciador a cada instante são descritos a seguir. É importante ressaltar que estamos considerando protocolos que têm como objetivo induzir o menor número de *checkpoints* durante uma construção consistente.

Propagação das Mensagens de Requisição

Como visto na Seção 3.3.2, os protocolos de *checkpointing* síncronos não-bloqueantes propagam informação de controle com as mensagens da aplicação para que, quando necessário, um *checkpoint* seja induzido imediatamente antes de receber uma delas evitando inconsistências. Esses *checkpoints* poderão vir a fazer parte de um *checkpoint* global consistente formado por uma construção consistente. Portanto, quando um processo, recebe uma mensagem de requisição, deve primeiro verificar se já possui um *checkpoint* consistente para esta construção consistente.

Para reduzir o custo de armazenamento em memória estável, os *checkpoints* induzidos por mensagens da aplicação podem ser salvos como *checkpoints* mutáveis. Um *checkpoint* mutável é armazenado em memória não-estável e é transferido para memória estável somente se faz parte de um *checkpoint* global consistente construído durante uma construção consistente. Quando o processo que possui um *checkpoint* mutável não recebe mensagens de requisição e portanto, não necessita do *checkpoint* mutável para nenhuma construção consistente, então este é descartado. Na Figura 5.1 (a), p_0 salva um *checkpoint* provisório ao receber a mensagem de requisição de p_1 e envia uma mensagem da aplicação m para p_2. O processo p_2 salva um *checkpoint* mutável antes de receber m. Na Figura 5.1 (b), quando p_2 recebe a requisição de p_1, p_2 transfere c_2^1 para memória estável e este *checkpoint*

é usado para formar o *checkpoint* global consistente da construção consistente iniciada por p_1.



(a) *checkpoint* mutável (b) construção consistente

Figura 5.1: *Checkpointing* síncrono com *checkpoints* mutáveis

Efeito Avalanche

O efeito avalanche ocorre quando um *checkpoint* induz outro *checkpoint* que induz um terceiro *checkpoint* e assim sucessivamente, podendo não ter fim. Quando as regras do protocolo de *checkpointing* permitem que um *checkpoint* mutável induza outro *checkpoint* mutável, então pode ocorrer o efeito avalanche de *checkpoints* mutáveis (Figura 5.2). Cao e Singhal eliminam esta possibilidade permitindo a cada processo salvar no máximo um *checkpoint* mutável durante uma construção consistente e nenhum *checkpoint* mutável é induzido se não existe uma construção consistente em andamento [8].

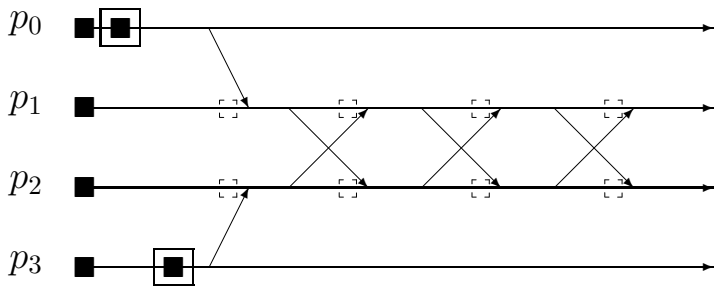


Figura 5.2: Efeito avalanche de *checkpoints* mutáveis

Coleta de Lixo

Como visto na Seção 3.5, a coleta de lixo ótima em protocolos síncronos é bem simples bastando remover os *checkpoints* salvos anteriormente aos *checkpoints* pertencentes a construção consistente encerrada. Nos protocolos síncronos não-bloqueantes com um único iniciador, a coleta de lixo deve remover, não somente os *checkpoints* dos participantes de uma construção consistente, mas também os *checkpoints* mutáveis que foram induzidos e não foram aproveitados para a formação do *checkpoint* global consistente construído. A Figura 5.3 apresenta uma construção consistente iniciada por p_1 que, após encerrar, elimina os *checkpoints* c_1^0 e c_2^0 pois p_1 e p_2 salvaram novos *checkpoints* que fazem parte desta construção consistente e p_0 removeu c_0^1 pois este foi induzido, mas não foi utilizado. Note que c_0^1 poderia ter sido utilizado para a formação do *checkpoint* global consistente que inclui o *checkpoint* iniciador p_1, mas como temos como objetivo salvar o menor número possível de *checkpoints* estáveis durante uma construção consistente, o *checkpoint* c_0^1 é ignorado. Na última fase de execução do protocolo de Cao e Singhal, a mensagem de liberação é enviada para todos os processos para que os processos não participantes da execução eliminem, caso necessário, seus *checkpoints* mutáveis [8].

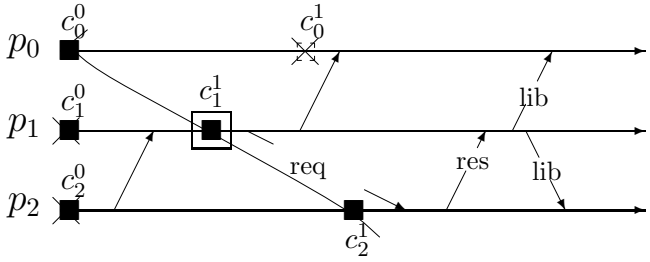


Figura 5.3: Coleta de lixo

5.1.2 Iniciadores Concorrentes

Os principais aspectos que devem ser considerados ao se desenvolver protocolos síncronos não-bloqueantes que permitem iniciadores concorrentes são descritos a seguir. É importante ressaltar que estamos considerando protocolos que têm como objetivo reduzir o número de *checkpoints* durante uma construção consistente.

Propagação das Mensagens de Requisição

Em um protocolo síncrono não-bloqueante com iniciações concorrentes é natural que, em um determinado instante, dois processos iniciadores estejam com suas construções

consistentes em andamento. Um processo, ao receber duas mensagens de requisição de construções consistentes distintas, deve definir para cada uma delas, o conjunto de processos para os quais deve propagar mensagens de requisição. Nenhuma das construções consistentes deve ser encerrada antes que todos os processos participantes armazenem seus *checkpoints*. Por exemplo, na Figura 5.4, p_2, ao receber uma mensagem de requisição de p_3, salva um *checkpoint* provisório e envia uma mensagem de requisição para p_1. Quando p_2 recebe a mensagem de requisição de p_0, p_2 já possui um *checkpoint* estável consistente com o *checkpoint* de p_0, portanto não é necessário p_2 armazenar outro *checkpoint* provisório. Neste momento, se p_2 simplesmente envia uma mensagem de resposta para p_0, do ponto de vista de p_0, o único processo participante em sua execução é o processo p_2 e portanto p_0 passa para a fase de liberações. Apesar de p_0 concluir sua construção consistente, não existe um *checkpoint* global consistente que inclui o *checkpoint* salvo por p_0 pois p_1 ainda não recebeu a mensagem de requisição enviada por p_2. Na Figura 5.4, o conjunto Σ representa um *checkpoint* global inconsistente construído pelo iniciador p_0.

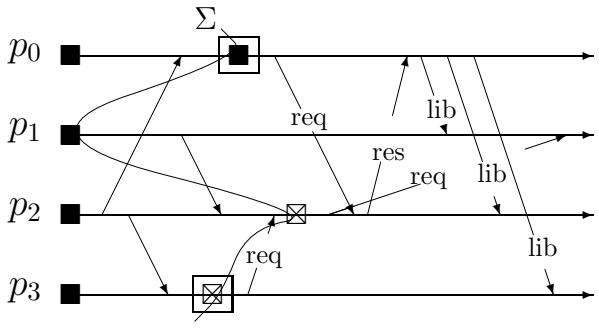


Figura 5.4: Problemas na recepção concorrente de requisições distintas

Vamos supor que p_a salvou o *checkpoint* c_a durante uma construção consistente $\mathcal{C}$. Quando p_a recebe uma mensagem de liberação correspondente à $\mathcal{C}$, todos os processos para os quais p_a enviou mensagens de requisição devem possuir um *checkpoint* estável consistente com o c_a. Uma maneira de prevenir que o protocolo encerre uma construção consistente precipitadamente na presença de múltiplos iniciadores é adiar o envio de uma mensagem de resposta ao iniciador se o processo já estiver aguardando uma mensagem de liberação de uma outra construção consistente em andamento. Por exemplo, na Figura 5.4, se o processo p_2 aguardasse a mensagem de liberação de p_3 para então enviar a mensagem de resposta para p_0, tanto o iniciador p_3 quanto o iniciador p_0 encerrariam suas construções consistentes corretamente.

Esta solução porém, não é recomendada pois pode levar o sistema a um estado de *deadlock*. Na Figura 5.5, p_0 envia uma mensagem de requisição para p_1 que salva um

checkpoint provisório e envia uma mensagem de requisição para p_2. Já o processo p_2 recebe uma mensagem de requisição do iniciador p_3, salva um *checkpoint* provisório e envia uma mensagem de requisição para p_1. Quando p_1 recebe a mensagem de requisição de p_2, p_1 espera pelo fim da construção consistente de p_0 para enviar uma mensagem de resposta ao iniciador p_3. Da mesma forma, p_2 espera pelo fim da construção consistente de p_3 para enviar resposta para p_1. Porém, o iniciador p_0 não pode concluir sua construção consistente enquanto não receber uma mensagem de resposta de p_2 e, p_3 fica aguardando uma mensagem de resposta de p_1. Neste caso, p_0 aguarda uma mensagem de resposta de p_2 que por sua vez, aguarda uma mensagem de liberação de p_3 que aguarda uma mensagem de resposta de p_1 que aguarda uma mensagem de liberação de p_0, o que caracteriza uma situação de *deadlock*.

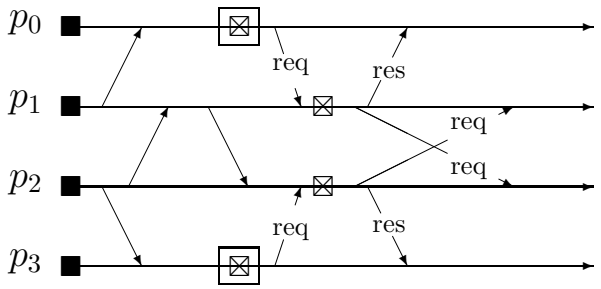


Figura 5.5: *Deadlock* na espera de liberação

Para que não ocorra *deadlock* no sistema, sugerimos a repropagação das mensagens de requisição. Isto é, um processo p_a, ao receber uma mensagem de requisição, verifica se deve salvar ou salvou um *checkpoint* provisório para o iniciador da mensagem de requisição. A partir do *checkpoint* provisório, p_a deve (re) propagar mensagens de requisição para todos os processos que possuem *checkpoints* que precedem o seu *checkpoint* provisório desde o seu último *checkpoint* permanente. Para o cenário da Figura 5.4, o processo p_2, ao receber a mensagem de requisição de p_0, deve repropagar a mensagem de requisição ao processo p_1 e então enviar uma mensagem de resposta para o processo p_0. Quando o processo p_0 recebe a mensagem de resposta de p_2, p_0 não conclui sua construção consistente pois sabe que existe um outro processo participante do qual ainda não obteve resposta. Assim, p_1 salva um *checkpoint* ao receber a primeira mensagem de requisição e envia uma mensagem de resposta para cada mensagem de requisição recebida e os iniciadores p_0 e p_3 encerram corretamente suas construções consistentes.

Efeito Avalanche

O efeito avalanche de *checkpoints* mutáveis pode ocorrer independentemente se o protocolo permite ou não iniciadores concorrentes. Note que o armazenamento de *checkpoints* mutáveis depende da regra adotada na recepção das mensagens da aplicação. Se esta regra é conhecida (por exemplo, baseada em um protocolo quase-síncrono) é possível prever o seu comportamento.

Lista de *Checkpoints* Estáveis

Na presença de um único iniciador a cada instante, cada processo necessita manter um *checkpoint* permanente e no máximo mais um *checkpoint* estável quando uma construção consistente está em andamento. Porém, quando mais de uma iniciação concorrente é permitida, os processos necessitam armazenar uma lista de *checkpoints* estáveis que conterá no máximo, um *checkpoint* provisório por construção consistente. Portanto, se considerarmos que um processo só inicia uma construção consistente após o término de sua construção consistente anterior, então o número máximo de *checkpoints* provisórios no sistema é $\mathcal{O}(n^2)$. Para cada processo, é necessário manter apenas um *checkpoint* permanente, pois quando um novo permanente é salvo, o anterior pode ser removido. Na Figura 5.6, o processo p_1 salva o seu primeiro *checkpoint* provisório para a construção consistente iniciada por p_0. Quando p_1 recebe a mensagem de requisição de p_2, p_1 ainda não recebeu uma mensagem de liberação de p_0 e portanto não pode transformar o seu *checkpoint* provisório em permanente, mas necessita armazenar outro *checkpoint* provisório para a construção consistente de p_2.

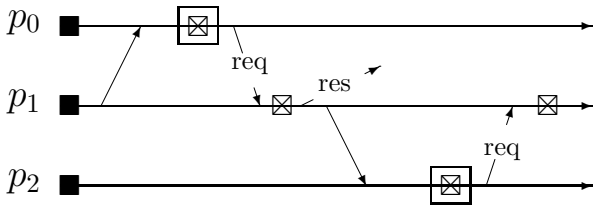


Figura 5.6: Lista de *checkpoints* estáveis

Coleta de Lixo

No momento em que um processo p_a recebe uma mensagem de liberação do iniciador p_{ini}, p_a verifica se existe um *checkpoint* provisório c_a^α consistente com o *checkpoint* de p_{ini}. Ao salvar c_a^α como permanente, p_a possui um *checkpoint* que faz parte de um *checkpoint*

global consistente construído por p_{ini}. Portanto, a idéia natural para a coleta de lixo é remover todos os *checkpoints* salvos anteriormente a c_a^α. Esse mecanismo é válido, porém, o processo envolvido deve ter informação suficiente para ignorar mensagens de requisição atrasadas. Para ilustrar possíveis problemas, na Figura 5.7, p_3 salva um *checkpoint* provisório para a construção consistente de p_0. Quando p_3 recebe a mensagem de requisição de p_2, que também é uma requisição da construção consistente de p_0, p_3 não necessita armazenar outro *checkpoint* provisório. Mas se p_3 salvou o seu *checkpoint* permanente para a construção consistente de p_4 e não possui informações suficientes para ignorar a mensagem de requisição enviada por p_2, então temos dois resultados não adequados: (i) p_3 estará armazenando mais que um provisório para a mesma construção consistente, mas um dos objetivos do protocolo é reduzir o número de *checkpoints* estáveis; (ii) se p_3 salva um novo *checkpoint* provisório, p_3 deve propagar a mensagem de requisição para p_4, mas p_3 já enviou uma mensagem de resposta para p_0 quando armazenou o *checkpoint* provisório anterior e portanto, p_0 pode concluir sua construção consistente antes da formação de um *checkpoint* global consistente.

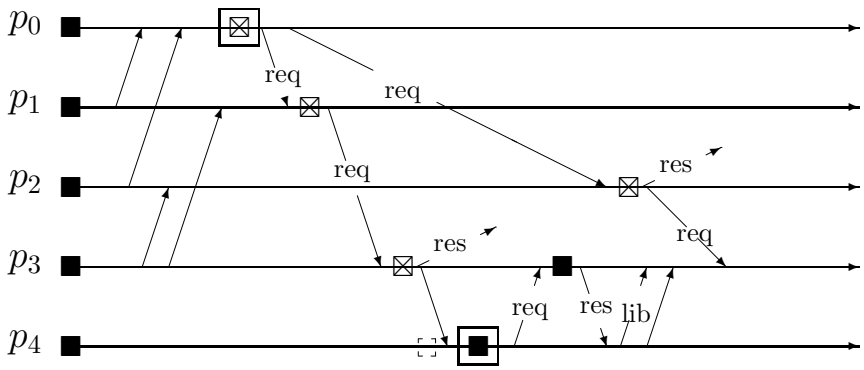


Figura 5.7: Recepção de uma mensagem de requisição atrasada

5.2 Minimalidade no Número de *Checkpoints*

Cao e Singhal provaram que é impossível desenvolver um protocolo síncrono não-bloqueante que armazena um número minimal de *checkpoints* para cada construção consistente [9]. Eles introduzem o conceito de z-dependência para descrever essa prova e afirmam que esta não pode ser descrita utilizando somente o conceito de *zigzag path* introduzido por Netzer e Xu [27]. Na Seção 2.4.3, mostramos que esses dois conceitos estão relacionados. Nesta Seção, descrevemos a prova da inexistência de um protocolo síncrono não-bloqueante e

minimal utilizando apenas o conceito de *zigzag paths*. Além disso, provamos também que não é possível garantir um número mínimo de *checkpoints* durante toda a execução da aplicação.

A participação de um processo em uma construção consistente se dá por meio do envio e recepção de mensagens de controle e/ou o armazenamento de *checkpoints*. Chamamos de processo participante, o processo que envia e recebe mensagens de controle.

Lema 4 *Seja p_{ini} o iniciador de um protocolo não-bloqueante minimal e p_a e p_b dois processos que salvam novos checkpoints durante a construção consistente $\mathcal{C}$ iniciada por p_{ini}. Se p_a enviou uma mensagem m após ter salvo o checkpoint durante $\mathcal{C}$ e p_b recebeu m e ainda não armazenou um checkpoint durante $\mathcal{C}$, então p_b deve armazenar um checkpoint imediatamente antes da recepção de m.*

Prova: Por contradição, vamos supor que p_b salva um *checkpoint* durante a construção consistente $\mathcal{C}$ iniciada por p_{ini}, após receber a mensagem m de p_a. Sabemos que em um protocolo minimal, p_b salva um *checkpoint* durante uma construção consistente se seu último *checkpoint* z-precede o *checkpoint* de p_{ini} (Definição 12). Portanto, se p_b salva seu *checkpoint* após processar m, nem o seu *checkpoint* anterior, nem o *checkpoint* salvo durante $\mathcal{C}$ fazem parte do *checkpoint* global consistente que contém o *checkpoint* do iniciador. A Figura 5.8 ilustra um cenário em que p_a e p_b salvam *checkpoints* durante $\mathcal{C}$ e p_b não salva um *checkpoint* antes de receber a mensagem de p_a. Note que, neste caso, p_b não possui *checkpoint* consistente com o *checkpoint* salvo pelo iniciador. □

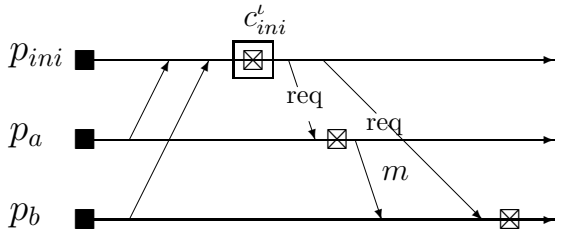
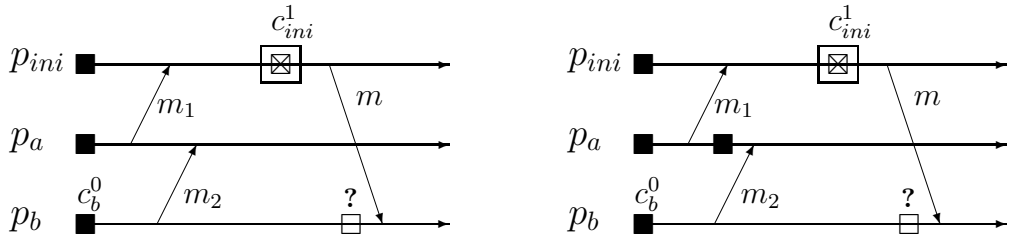


Figura 5.8: O processo p_b não possui *checkpoint* consistente com c^{ι}_{ini}

Lema 5 *Em um protocolo não-bloqueante e minimal, não há informação suficiente na recepção de uma mensagem m, enviada por um participante durante uma construção consistente $\mathcal{C}$, para decidir se existe uma zigzag path entre o último checkpoint do processo que recebeu m e o checkpoint salvo pelo iniciador de $\mathcal{C}$.*

Prova: Em um protocolo não-bloqueante, os processos não suspendem suas atividades durante uma construção consistente. Suponha que o processo p_a, participante de uma construção consistente $\mathcal{C}$, após armazenar seu *checkpoint* durante $\mathcal{C}$, envia uma mensagem m para o processo p_b. Sabemos que, em um protocolo minimal, o processo p_b deve salvar um *checkpoint* imediatamente antes de receber m somente se o último *checkpoint* de p_b z-precede o *checkpoint* do iniciador de $\mathcal{C}$ (Lema 4). Vamos provar que é impossível para p_b decidir, no momento da recepção de m, se deve ou não salvar um *checkpoint* para garantir minimalidade no número de *checkpoints*. Vamos analisar o cenário da Figura 5.9, onde o iniciador p_{ini} envia uma mensagem m para p_b antes de encerrar sua construção consistente. Na Figura 5.9 (a) temos que $c_b^0 \rightsquigarrow c_{ini}^1$ e portanto p_b deve armazenar um *checkpoint* imediatamente antes da recepção de m. Já no caso do cenário da Figura 5.9 (b), p_b não é participante da construção consistente de p_{ini} pois p_a possui um *checkpoint* consistente com o *checkpoint* do iniciador que foi salvo antes da recepção de m_2. Portanto, p_b não deve salvar um *checkpoint* na recepção da mensagem m. Porém, p_b pode receber informações sobre os eventos p_a até o envio da mensagem m_1 por informações de controle adicionados às mensagens da aplicação. O que ocorre após o envio de m_1 só seria conhecido por outros processos, inclusive por p_b, se p_a tivesse enviado uma mensagem com os eventos ocorridos após m_1. Mas no momento da recepção de m é impossível para p_b saber se existe ou não um *checkpoint* armazenado por p_a após o envio de m_1 e antes da recepção de m_2. □



(a) $c_b^0 \rightsquigarrow c_{ini}^1$ (b) $c_b^0 \not\rightsquigarrow c_{ini}^1$

Figura 5.9: Inexistência de um protocolo não-bloqueante minimal

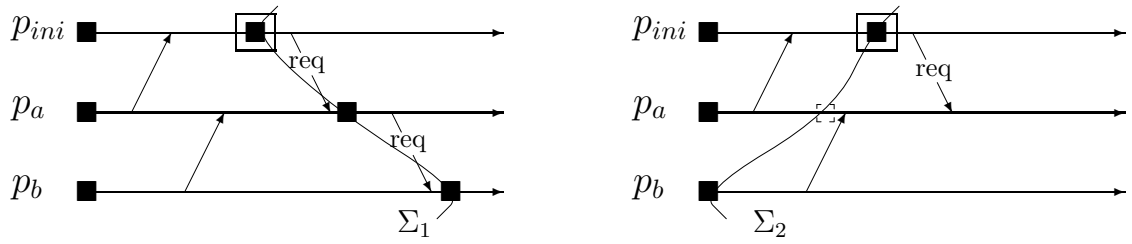
Teorema 9 *Um protocolo não pode ser não-bloqueante e minimal.*

Prova: Pelo Lema 4, sabemos que um processo, ao receber uma mensagem durante uma construção consistente, precisaria saber se existe uma *zigzag path* a partir do seu último *checkpoint* até o *checkpoint* salvo pelo iniciador. Porém, pelo Lema 5 é impossível obter

essa informação no momento da recepção da mensagem. Portanto, um protocolo não pode ser não-bloqueante e minimal. □

Os *checkpoints* mutáveis foram introduzidos para garantir a minimalidade no número de *checkpoints* estáveis em protocolos de *checkpointing* síncronos não-bloqueantes [8]. *Checkpoints* mutáveis são *checkpoints* que podem ser salvos em memória não-estável e reduzem o custo de transferência do estado para armazenamento estável (Seção 3.3.2).

Notamos que a minimalidade no número de *checkpoints* estáveis nos protocolos de *checkpointing* síncronos não-bloqueantes depende do momento em que os *checkpoints* mutáveis são salvos. Por exemplo, na Figura 5.10 (a), nenhum *checkpoint* mutável foi salvo e portanto, quando p_{ini} iniciou uma construção consistente, o protocolo minimal induziu novos *checkpoints* em p_a e p_b para formar um *checkpoint* global consistente. Na Figura 5.10 (b), todas as *zigzag paths* não-causais são evitadas por meio da indução de *checkpoints* mutáveis e, portanto, p_a salvou um *checkpoint* mutável antes de receber uma mensagem de p_b. Quando p_a recebe uma mensagem de requisição de p_{ini}, p_a pode transformar esse mutável em estável e o protocolo minimal não induzirá um novo *checkpoint* em p_b.



(a) existência de *zigzag path* não-causal (b) inexistência de *zigzag path* não-causal

Figura 5.10: Minimalidade no número de *checkpoints* estáveis

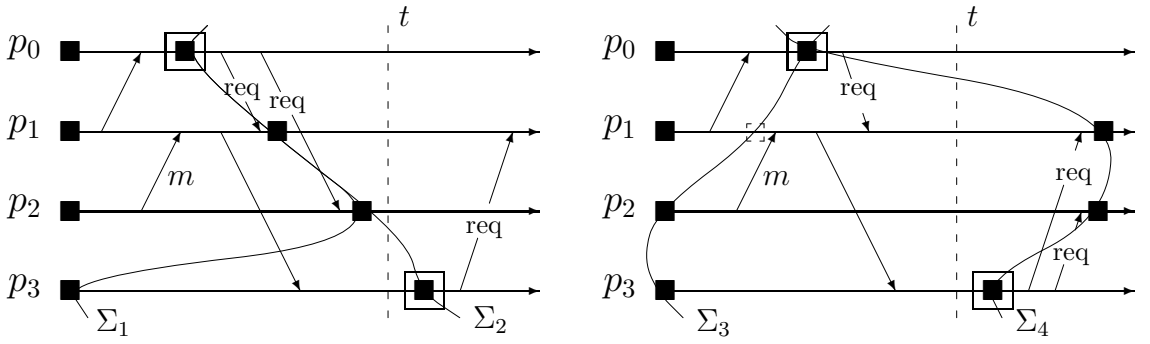
Se todas as *zigzag paths* não-causais são evitadas, então cada execução de um protocolo de *checkpointing* síncrono não-bloqueante constrói um *checkpoint* global consistente mais à esquerda. Porém, o armazenamento de *checkpoint* em um determinado instante de tempo pode reduzir a necessidade de *checkpoints* no futuro. Assim, concluímos que a construção de um *checkpoint* global consistente por um protocolo síncrono não-bloqueante depende do momento em que os *checkpoints* mutáveis são salvos. A seguir, provamos que não existe um protocolo síncrono não-bloqueante que garante um número mínimo de *checkpoints* durante toda a execução da computação distribuída. A descrição desta prova utiliza uma abordagem semelhante ao proposto por Tsai e outros para os protocolos quase-síncronos RDT [41].

Teorema 10 *Nenhum protocolo de checkpointing síncrono não-bloqueante garante o número mínimo de checkpoints estáveis para todos os possíveis padrões de checkpoints e mensagens.*

Prova: Esta prova é baseada no padrão de *checkpoints* e mensagens da Figura 5.11. A Figura 5.11 (a) armazena *checkpoints* mutáveis somente se uma construção consistente está em andamento, conforme proposto por Cao e Singhal [8]. A Figura 5.11 (b) armazena *checkpoints* mutáveis para garantir a propriedade RDT, evitando todas as *zigzag paths* não-causais não duplicadas causalmente. Devemos observar também que para cada construção consistente apenas um número minimal de *checkpoints* estáveis deve ser salvo.

Vamos considerar o cenário da Figura 5.11 até o tempo t, ou seja, vamos analisar apenas a construção consistente iniciada por p_0. Na Figura 5.11 (a), p_1 e p_2 devem salvar *checkpoints* estáveis para construir o *checkpoint* global consistente representado por Σ_1, enquanto que apenas p_1 deve transformar seu *checkpoint* mutável em estável para construir o *checkpoint* global consistente Σ_3, como indica a Figura 5.11 (b). Assim, o armazenamento de um *checkpoint* mutável salvo, mesmo enquanto não havia nenhuma construção consistente em andamento, possibilitou o número mínimo de *checkpoints*.

Vamos agora analisar a execução após as construções consistentes iniciadas por p_0 e por p_3. Note que p_3 inicia sua construção consistente após o término da construção consistente de p_0, ou seja, as iniciações não são concorrentes. A Figura 5.11 (a) mostra que os *checkpoints* salvos durante a construção consistente iniciada por p_0 também são consistentes com o *checkpoint* salvo por p_3. Porém, para o cenário da Figura 5.11 (b), p_1 salvou o *checkpoint* antes da recepção de m e portanto, p_1 e p_2 devem salvar novos *checkpoints* durante a construção consistente iniciada por p_3 para construir um *checkpoint* global consistente. Neste caso, temos o cenário da Figura 5.11 (a) com menos *checkpoints* estáveis que o cenário da Figura 5.11 (b). □



(a) Cao e Singhal [8] (b) RDT-NBS

Figura 5.11: Impossibilidade do número mínimo de *checkpoints* estáveis

Um protocolo síncrono salva um número minimal de *checkpoints* durante uma construção consistente $\mathcal{C}$ se, para cada *checkpoint* estável salvo durante $\mathcal{C}$ não existe um *checkpoint* estável anterior que possa fazer parte do *checkpoint* global consistente construído pela construção consistente. Um protocolo síncrono não-bloqueante que permite iniciadores concorrentes e que tem como objetivo armazenar um número minimal de *checkpoints* estáveis deve considerar os *checkpoints* estáveis salvos pelos iniciadores concorrentes e os *checkpoints* salvos durante as construções consistentes. Isto é, pode-se formar um único *checkpoint* global consistente que inclui os *checkpoints* salvos pelos iniciadores das construções consistentes. Portanto, para um mesmo padrão de *checkpoints* e mensagens, *checkpoints* globais consistentes diferentes podem ser construídos pelo protocolo na presença de um único iniciador e com iniciadores concorrentes. A Figura 5.12 mostra um cenário com os iniciadores p_0 e p_3, mas p_0 inicia sua construção consistente somente após o término da construção consistente de p_3. Quando p_1 recebe uma mensagem de liberação de p_3, p_1 pode remover o seu *checkpoint* mutável já que este não foi necessário para a construção consistente de p_3. Neste caso, p_1 e p_2 precisam salvar novos *checkpoints* para formar um *checkpoint* global consistente que inclui o *checkpoint* salvo pelo iniciador p_0.

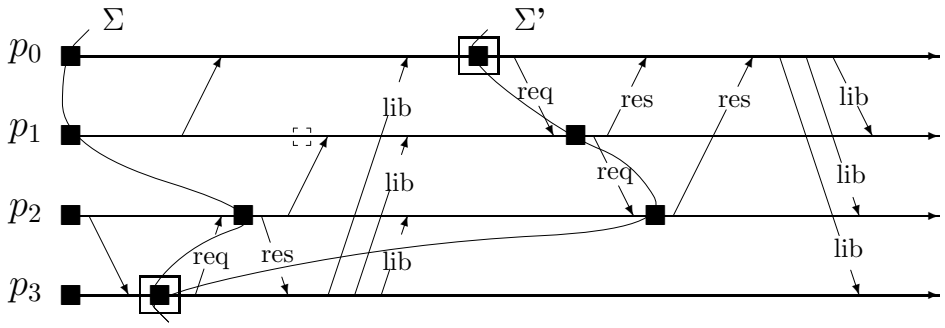


Figura 5.12: Minimalidade e um iniciador a cada instante

Um cenário semelhante em que os iniciadores p_0 e p_3 iniciam suas construções consistentes concorrentemente, armazena um número minimal de *checkpoints* transformando o *checkpoint* mutável salvo por p_1 em *checkpoint* provisório (Figura 5.13). Note que o processo p_1 salvou um *checkpoint* mutável para a construção consistente de p_3 e recebe uma mensagem de requisição do iniciador p_0. Neste caso, p_1 não necessita propagar uma mensagem de requisição para p_2. Nesta figura, Σ representa o *checkpoint* global consistente inclui os *checkpoints* salvos pelos iniciadores e induz um número minimal de *checkpoints*. Neste cenário, quando p_1 recebe uma mensagem de requisição de p_0, p_1 deve decidir apenas com informação local entre armazenar um novo *checkpoint* provisório com as informações do iniciador p_0 ou aproveitar um mutável salvo com informações de

uma outra construção consistente em andamento. Porém, concluímos que um processo não tem informação suficiente para tomar essa decisão e garantir um número minimal de *checkpoints* estáveis salvos para as construções consistentes concorrentes.

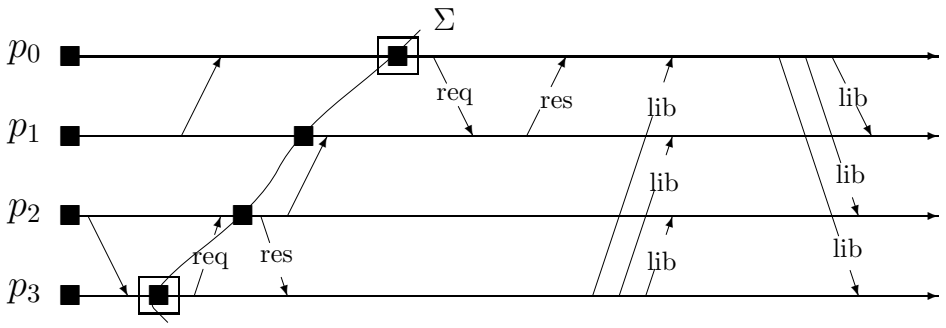


Figura 5.13: Minimalidade e iniciadores concorrentes

Teorema 11 *Não existe um protocolo síncrono não-bloqueante que armazena um número minimal de checkpoints estáveis na presença de iniciadores concorrentes.*

Prova: Esta prova é baseada no cenário em que um processo p_x possui dois *checkpoints* mutáveis e não tem informação suficiente para decidir qual *checkpoint* mutável deve transformar como estável para garantir um número minimal de *checkpoints* estáveis. Na Figura 5.14, p_x salvou o *checkpoint* c_x^a antes de receber uma mensagem da aplicação com informações sobre a construção consistente de p_a. Da mesma forma, c_x^b foi salvo ao receber informações sobre a construção consistente de p_b.

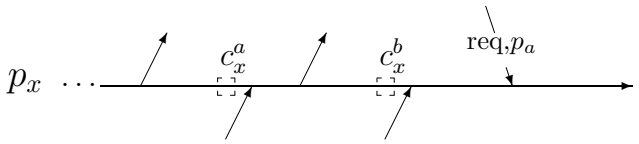
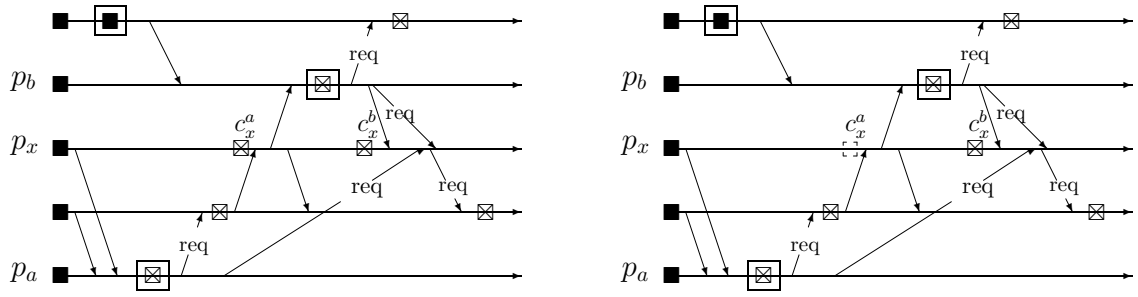


Figura 5.14: Cenário para a prova do Teorema 11

Quando p_x recebe a mensagem de requisição referente à construção consistente de p_a, se p_x opta em transferir para memória estável o *checkpoint*:

- c_x^a – p_x pode receber posteriormente uma mensagem de requisição da construção consistente de p_b e, neste momento, deverá transformar o *checkpoint* c_x^b em estável

(Figura 5.15 (a)). Neste caso, seria melhor salvar c_x^b como estável na recepção da mensagem de requisição de p_a e descartar c_x^a (Figura 5.15 (b)). Portanto, o protocolo não pode ser minimal transformando sempre c_x^a em estável.

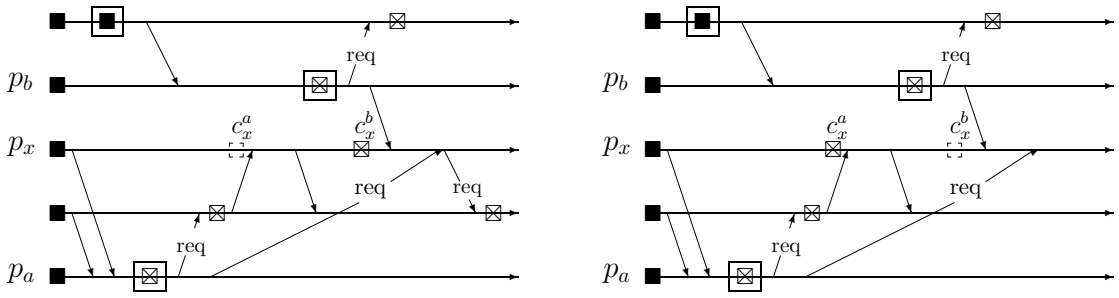


(a) c_x^a e c_x^b são transformados em estáveis (b) c_x^b é transformado em estável

Figura 5.15: p_x recebe uma mensagem de requisição de p_b

- c_x^b – c_x^b é transferido para memória estável e c_x^a é descartado. O processo p_x pode não receber uma mensagem de requisição da construção consistente de p_b (Figura 5.16 (a)). Quando p_x transforma c_x^b em *checkpoint* estável, p_x envia uma mensagem de requisição para todos os processos que possuem um *checkpoint* que precede c_x^b induzindo novos *checkpoints*. Porém, c_x^b não z-precede o *checkpoint* salvo pelo iniciador p_b durante sua construção consistente. Portanto, para este caso, um *checkpoint* global consistente com um número minimal de *checkpoints* é formado transformando-se c_x^a em provisório e descartando c_x^b como apresenta a Figura 5.16 (b). Assim, o protocolo também não pode ser minimal ao transformar sempre c_x^b como estável.

No momento em que p_x recebe a mensagem de requisição de p_a, p_x não tem como saber se c_x^b será necessário para a construção consistente iniciada por p_b. Então, p_x não tem como decidir qual dos *checkpoints* (c_x^a ou c_x^b) salvar como provisório para garantir o número minimal de *checkpoints* para as construções consistentes concorrentes. Portanto, não existe um protocolo síncrono não-bloqueante com um número minimal de *checkpoints* estáveis na presença de iniciadores concorrentes. □



(a) c_x^b induz um *checkpoint* adicional (b) nenhum *checkpoint* adicional

Figura 5.16: p_x não recebe uma mensagem de requisição de p_b

5.3 Protocolo RDT-NBS

A idéia do protocolo RDT-NBS, proposto em [33], é adicionar mensagens de controle ao protocolo de *checkpointing* quase-síncrono FDAS (*Fixed Dependency After Send*) [45] com o objetivo de desenvolver um protocolo de *checkpointing* síncrono não-bloqueante que utiliza *checkpoints* mutáveis. No protocolo quase-síncrono FDAS, um processo não modifica seu vetor de dependências (`VD`) após enviar uma mensagem da aplicação garantindo assim a presença da propriedade RDT [45]. O protocolo RDT-NBS induz *checkpoints* mutáveis da mesma maneira que o FDAS induz *checkpoints* forçados, prevenindo a existência de *zigzag paths* não-causais entre *checkpoints*. Além disso, o uso de `VD`s permite o rastreamento on-line das dependências entre *checkpoints*. Escolhemos o protocolo quase-síncrono RDT chamado FDAS por sua simplicidade, porém RDT-NBS pode ser facilmente reescrito usando qualquer outro protocolo RDT [4, 18]. Por exemplo, nas Seções C.4 e C.3 encontram-se os protocolos síncronos não-bloqueantes RDT-Minimal-NBS e RDT-Partner-NBS baseados nos protocolos quase-síncronos RDT-Minimal e RDT-Partner.

5.3.1 Descrição do Protocolo

O protocolo FDAS é um protocolo quase-síncrono que possui a propriedade RDT. Neste protocolo, um processo não altera o seu vetor de dependências após o envio de uma mensagem da aplicação durante um intervalo de *checkpoints* [45]. Uma otimização na condição de indução de *checkpoints* foi proposta em [17] e sua descrição é apresentada na Seção 3.2.2.

Em um padrão RDT, o *checkpoint* global consistente mínimo que contém um *checkpoint* c pode ser calculado a partir do vetor de dependências de c. Quando um iniciador p_{ini} salva um *checkpoint* provisório, p_{ini} envia uma mensagem de requisição para todos os processos participantes. Um processo p_i é participante de p_{ini} se $\texttt{VD}_{ini}\texttt{[}i\texttt{]}$ foi alterado desde o último *checkpoint* permanente de p_{ini}, ou seja, existe uma precedência causal entre um *checkpoint* permanente de p_i ao *checkpoint* provisório salvo por p_{ini} (Seção 2.3.2). Todo processo que recebe uma mensagem de requisição atende uma de três condições: já possui um *checkpoint* estável consistente com o *checkpoint* do iniciador; ou transforma um *checkpoint* mutável em provisório; ou salva um novo *checkpoint* provisório. Em qualquer caso, uma mensagem de resposta deve ser enviada para p_{ini}. Quando p_{ini} recebe uma mensagem de resposta de cada um dos seus participantes, p_{ini} envia mensagens de liberação aos processos participantes finalizando sua construção consistente.

O protocolo RDT-NBS permite a presença de iniciadores concorrentes. Neste protocolo, as mensagens de controle são enviadas do iniciador aos participantes e dos participantes ao iniciador, de forma direta, isto é, apenas o iniciador propaga mensagens de requisição e liberação para no máximo, $n-1$ processos participantes e cada processo participante envia exatamente uma mensagem de resposta ao iniciador. Portanto, este protocolo necessita de no máximo $\mathcal{O}(n)$ mensagens de controle em cada uma das fases de *checkpointing*. Além disso, o problema da repropagação das mensagens de requisição foi facilmente resolvido neste protocolo pois cada iniciador conhece todos os participantes de sua construção consistente. É importante observar que na fase de liberações não é feito um *broadcast* da mensagem de liberação.

Variáveis do processo

As variáveis utilizadas por este protocolo são descritas a seguir:

- `enviou` – indica se o processo enviou pelo menos uma mensagem durante o intervalo de *checkpoints* corrente.
- $\texttt{VD}_i$ – vetor de dependências mantido pelo processo p_i para capturar as precedências causais entre *checkpoints*.
- $\texttt{VP}_{ini}$ – vetor com os índices dos processos participantes atualizado pelo iniciador p_{ini} a cada início de uma construção consistente. Um processo p_i é inserido como participante da construção consistente pelo iniciador p_{ini} se $\texttt{VD}\texttt{[}i\texttt{]}$ foi alterado desde o primeiro *checkpoint* permanente mantido por p_{ini}. Seja `p` o primeiro *checkpoint* permanente mantido p_{ini}. Então, $\texttt{VP}_{ini}\texttt{[}i\texttt{]}$ é atualizado com o valor de $\texttt{VD}_{ini}\texttt{[}i\texttt{]}$ se $\texttt{VD}_{ini}\texttt{[}i\texttt{]} > \texttt{p.VD[}i\texttt{]}$.
- `respostas` – vetor mantido pelo iniciador para detecção da terminação da construção consistente.

Início da construção consistente

Para iniciar uma construção consistente, um processo p_{ini} armazena um *checkpoint* provisório e envia mensagens de requisição para cada um dos seus participantes. O conteúdo de $\texttt{VD}_{ini}\texttt{[}i\texttt{]}$ é propagado com a mensagem de requisição enviada para p_i (`req.ind`).

Recepção da mensagem de requisição

Quando um processo p_i recebe uma mensagem de requisição de p_{ini}, p_i compara o seu índice atual com o índice da requisição. Se $\texttt{req.ind} \geq \texttt{VD}_i\texttt{[i]}$, então p_i salva um *checkpoint* provisório. Senão, p_i procura pelo primeiro *checkpoint* com índice maior que `req.ind` e, caso esse *checkpoint* seja um *checkpoint* mutável, ele é transformado em provisório. Em qualquer um dos casos, p_i envia uma mensagem de resposta para p_{ini}.

Recepção da mensagem de resposta

Um processo p_{ini} processa uma mensagem de resposta apenas se a construção consistente que iniciou não foi encerrada, ou seja, se possui um *checkpoint* provisório com o índice da construção consistente. O iniciador p_{ini}, após receber uma mensagem de resposta de cada um dos seus participantes, transforma o seu *checkpoint* provisório em permanente, remove os *checkpoints* anteriormente salvos e envia uma mensagem de liberação para todos os participantes de sua construção consistente.

Recepção da mensagem de liberação

Um processo, ao receber uma mensagem de liberação, transforma o *checkpoint* correspondente em permanente e remove todos os *checkpoints* anteriores a ele.

A descrição do protocolo RDT-NBS é apresentada pelo Protocolo 5.1. Consideramos que todo procedimento deste protocolo é executado de forma atômica.

Protocolo 5.1 RDT-NBS (declarações)

Variáveis do processo:
- VD ≡ vetor$[0 \ldots n-1]$ de inteiros
- enviou ≡ booleano
- VP ≡ vetor$[0 \ldots n-1]$ de inteiros
- respostas ≡ vetor$[0 \ldots n-1]$ de inteiros
- pid ≡ inteiro

Tipos de Mensagem:
- aplicação:
  - VD ≡ vetor$[0 \ldots n-1]$ de inteiros
- requisição:
  - ipid ≡ inteiro
  - ind ≡ inteiro
- resposta:
  - ind ≡ inteiro
- liberação:
  - ind ≡ inteiro

Protocolo 5.1 RDT-NBS

Início:
  $\forall i$: VD[i] $\leftarrow$ 0
  salvaCkpt (permanente)

Envio da mensagen da aplic. (m) para p_k
  enviou $\leftarrow$ *verdadeiro*
  m.VD $\leftarrow$ VD
  envia mensagem da aplicação (m)

Recepção da mensagem da aplic. (m) de p_k:
  se (enviou *e* m.VD[k] > VD[k])
    salvaCkpt (mutável)
  $\forall i$: VD[i] $\leftarrow$ max(VD[i], m.VD[i])
  processa a mensagem da aplicação (m)

Início da construção consistente:
  c $\leftarrow$ salvaCkpt (provisório)
  verificaVP(c)
  se ($\exists i \neq$ pid: VP[i] > 0)
    $\forall i$: se (VP[i] $\neq$ 0)
        req.ipid $\leftarrow$ pid
        req.ind $\leftarrow$ VP[i]
        envia requisição (req) para p_i
    $\forall i$: respostas[i] $\leftarrow$ 0
  senão
    transformaCkpt(c, permanente)

Recepção da requisição (req) de p_{ini}**:**
  se (req.ind $\geq$ VD[pid])
    res.ind $\leftarrow$ VD[pid]
    salvaCkpt (provisório)
  senão
    $c \leftarrow$ procuraCkpt(req.ind)
    se (c é mutável)
      transformaCkpt(c, provisório)
    res.ind $\leftarrow$ c.VD[pid]
  envia resposta (res) para p_{ini}

Recepção da resposta (res) de p_k**:**
  c $\leftarrow$ procuraCkpt(VP[pid])
  se (c *não* é *nulo e* c é provisório)
    respostas[k] $\leftarrow$ res.ind
    se ($\nexists i \neq$ pid: VP[i] $\neq$ 0 *e*
      VP[i] > respostas[i])
      $\forall i \neq$ pid: se (VP[i] $\neq$ 0)
          lib.ind $\leftarrow$ respostas[i]
          envia liberação (lib) para p_i
      transformaCkpt(c, permanente)

Recepção da liberação (lib) de p_k**:**
  $c \leftarrow$ procuraCkpt(lib.ind)
  transformaCkpt(c, permanente)

salvaCkpt(tipo)
  salva um checkpoint de acordo com o tipo
  VD[pid] $\leftarrow$ VD[pid] + 1
  enviou $\leftarrow$ *falso*
  retorna o ckpt salvo

verificaVP(c)
  p $\leftarrow$ primeiro checkpoint permanente
  $\forall i$: se (c.VD[i] > p.VD[i])
      VP[i] $\leftarrow$ VD[i]
    senão
      VP[i] $\leftarrow$ 0

transformaCkpt(c, tipo)
  transforma c de acordo com tipo
  se (tipo = permanente)
    remove os ckpts salvos antes de c

procuraCkpt(ind)
  retorna o primeiro checkpoint c com
    c.VD[pid] $\geq$ ind

5.3.2 Exemplo

Um exemplo de execução do protocolo RDT-NBS é ilustrado pela Figura 5.17. Neste cenário, o iniciador p_2 envia uma mensagem de requisição ao processo p_3 e uma mensagem da aplicação para p_1. O processo p_1 salva um *checkpoint* mutável imediatamente antes de processar a mensagem e então envia uma mensagem para p_3. O processo p_3 recebe a mensagem de p_1 e salva um *checkpoint* mutável antes de processar a mensagem. Quando

p_3 recebe a mensagem de requisição de p_2, p_3 já possui um *checkpoint* mutável com índice igual ao índice conhecido por p_2 e portanto, p_3 transforma esse *checkpoint* mutável em provisório. No final da construção consistente de p_2, os *checkpoints* provisórios de p_2 e p_3 são transformados em permanentes e formam um *checkpoint* global consistente com os *checkpoints* iniciais de p_0 e p_1. Na construção consistente de p_0, os *checkpoints* mutáveis de p_1 e p_2 são transformados em provisórios e posteriormente em permanentes. Note que o primeiro *checkpoint* mutável de p_1 só é descartado quando o segundo é transformado em permanente. A linha Σ representa o *checkpoint* global consistente construído por p_0.

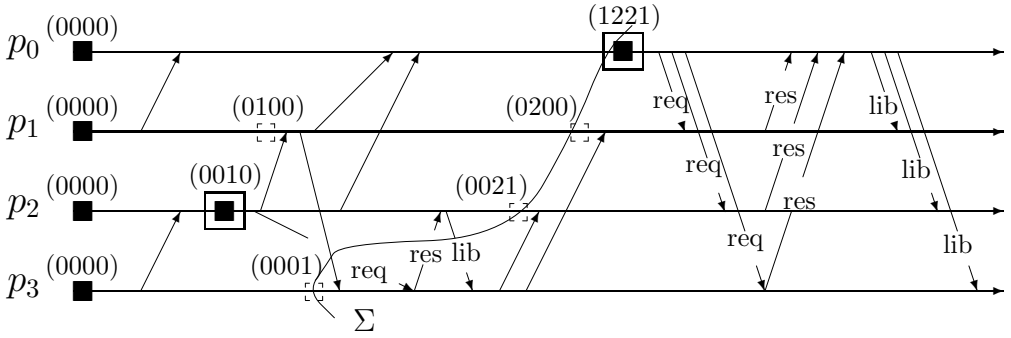


Figura 5.17: Exemplo de padrão gerado pelo protocolo RDT-NBS

5.3.3 Prova de Correção

Nesta seção, analisamos o comportamento do protocolo RDT-NBS na presença de um único e múltiplos iniciadores concorrentes. Provaremos que este protocolo salva um número minimal de *checkpoints* estáveis na presença de um único iniciador e que mantém um *checkpoint* global consistente a qualquer instante de sua execução.

Teorema 12 *Na presença de um único iniciador a cada instante de tempo, o protocolo RDT-NBS salva um número minimal de checkpoints estáveis durante uma construção consistente.*

Prova: Por contradição, suponha que o processo p_a salva o *checkpoint* estável c_a^{α} durante a construção consistente $\mathcal{C}$ iniciada por p_{ini} e $c_a^{\alpha-1} \not\rightarrow c_{ini}^{\iota}$, onde c_{ini}^{ι} é o *checkpoint* salvo pelo iniciador p_{ini} para $\mathcal{C}$. Se p_a salva c_a^{α} como estável durante $\mathcal{C}$, então p_a recebeu uma mensagem de requisição de p_{ini}. Então, p_{ini} modificou a entrada $\mathtt{VD}(c_{ini}^{\iota})[a]$ do seu vetor durante seu último intervalo de *checkpoints* que implica que o *checkpoint* salvo antes de c_a^{α} precede causalmente c_{ini}^{ι} ($c_a^{\alpha-1} \rightarrow c_{ini}^{\iota}$). □

Lema 6 *Seja c_a^α o checkpoint mais à esquerda de p_a. Se $\mathtt{VD}(c_a^\alpha)[b] = \beta$, então c_b^β é um checkpoint estável.*

Prova: Sabemos pela política de coleta de lixo que o primeiro *checkpoint* mantido por um processo é permanente e portanto, c_a^α é um *checkpoint* permanente. Por contradição, suponha que c_b^β é um *checkpoint* mutável ou c_b^β não foi armazenado por p_b. Sabemos que:

- se $\mathtt{VD}(c_a^\alpha)[b] = \beta$, então $c_b^{\beta-1} \rightarrow c_a^\alpha$;
- se c_a^α é o *checkpoint* mais à esquerda de p_a, então p_a participou de uma construção consistente $\mathcal{C}$ que já foi finalizada;
- o uso de vetores de dependências permite o rastreamento das dependências causais;
- o protocolo RDT-NBS garante a ausência de *zigzag paths* não duplicadas causalmente por meio da indução de *checkpoints* mutáveis.

Então, $c_b^{\beta-1}$ precede causalmente o *checkpoint* do iniciador da construção consistente $\mathcal{C}$ e portanto, p_b é participante de $\mathcal{C}$. Assim, no momento em que p_b recebe uma mensagem de requisição, temos duas possibilidades: p_b salva c_b^β provisório ou p_b já possui o *checkpoint* c_b^β estável salvo antes de $\mathcal{C}$. □

Teorema 13 *O protocolo RDT-NBS garante que, a qualquer momento, existe um checkpoint global consistente formado por checkpoints estáveis.*

Prova: Sabemos que o vetor de dependências salvo com um *checkpoint* c indica o índice dos *checkpoints* que forma um *checkpoint* global consistente que inclui c [45]. Sabemos também que a união de *checkpoints* globais consistentes tem como resultado um *checkpoint* global consistente. Seja $c_a^{\iota_a}$ o *checkpoint* mais à esquerda de p_a. Seja α_i o máximo de $\mathtt{VD}(c_j^{\iota_j})[i]$, $\forall j$. Então, um *checkpoint* global consistente Σ pode ser formado por:

$\Sigma = \{c_0^{\alpha_0}, \ldots, c_{n-1}^{\alpha_{n-1}}\}$, tal que $\alpha_i = \max(\mathtt{VD}(c_j^{\iota_j})[i])$, $\forall j$ e $c_j^{\iota_j}$ é o *checkpoint* mais à esquerda de p_j.

Pelo Lema 6, $c_a^{\alpha_a}$ é um *checkpoint* estável. Então, Σ é consistente e é formado apenas por *checkpoints* estáveis. □

5.4 Protocolo BCS-NBS

O protocolo BCS-NBS é baseado no protocolo quase-síncrono da classe ZCF, chamado BCS [6] acrescido do mecanismo semelhante ao FDAS que impede a indução de *checkpoints* se nenhuma mensagem foi enviada no mesmo intervalo de *checkpoints* [42]. Este

protocolo requer a propagação de apenas um inteiro com as mensagens da aplicação. Além disso, este protocolo não sofre do efeito avalanche de *checkpoints* mutáveis e permite a presença de iniciadores concorrentes. Uma versão mais simples deste protocolo que não utiliza *checkpoints* mutáveis foi proposta em [34].

5.4.1 Descrição do Protocolo

Este protocolo utiliza relógios lógicos semelhantes ao mecanismo proposto por Lamport [21], mas ao invés de enumerar todos os eventos dos processos, apenas os *checkpoints* recebem um índice que é incrementado a cada novo armazenamento e pode ser atualizado no momento da recepção de mensagens. Quando um processo p_i recebe uma mensagem com índice maior que o índice mantido por p_i e p_i enviou uma mensagem no intervalo atual, um *checkpoint* mutável é induzido imediatamente antes de processar a mensagem e p_i atualiza o seu índice como sendo igual ao índice recebido na mensagem.

Quando uma nova construção consistente é iniciada por p_{ini}, p_{ini} salva um *checkpoint* provisório, e envia uma mensagem de requisição para p_k se p_{ini} recebeu uma mensagem de p_k com novo índice durante o seu último intervalo de *checkpoints*. O índice do *checkpoint* provisório salvo por p_{ini} é o índice da construção consistente. Um processo p_i, ao receber uma mensagem de requisição de p_k, verifica se é necessário armazenar um *checkpoint* provisório (novo ou pela transformação de um *checkpoint* mutável) e propaga mensagens de maneira semelhante ao iniciador, porém, não necessita enviar mensagens de requisição para os processos que já são participantes da construção consistente. Além disso, se o índice atual de p_i é menor que o índice da construção consistente e p_i enviou uma mensagem no intervalo, p_i salva um novo *checkpoint* mutável para atualizar o seu índice. No final da construção consistente, um novo *checkpoint* global consistente é construído e todos os processos envolvidos estão com o índice maior ou igual ao do iniciador p_{ini}.

Variáveis do processo

As variáveis utilizadas por este protocolo são descritas a seguir:

- `enviou` – indica se o processo enviou pelo menos uma mensagem durante o intervalo de *checkpoints* corrente.
- $\texttt{VI}_i$ – vetor de índices mantido pelo processo p_i para capturar as precedências causais diretas entre *checkpoints*. A entrada $\texttt{VI}_i\texttt{[}i\texttt{]}$ indica o índice atual de p_i. Quando p_i recebe uma mensagem de p_k, p_i atualiza em seu vetor o índice propagado por p_k ($\texttt{VI}_i\texttt{[}k\texttt{] = max(m.ind, VI}_i\texttt{[}k\texttt{])}$).
- $\texttt{VP}_i$ – vetor com os índices dos participantes de uma construção consistente. Um processo p_j é inserido como participante da construção consistente por p_i se p_i

recebeu uma mensagem com novo índice de p_j desde o seu primeiro *checkpoint* permanente. Seja `p` o primeiro *checkpoint* permanente mantido p_i. Então, $\texttt{VP}_i\texttt{[}j\texttt{]}$ é atualizado com o valor de $\texttt{VI}_i\texttt{[}j\texttt{]}$ se $\texttt{VI}_i\texttt{[}j\texttt{]} > \texttt{p.VI[}j\texttt{]}$.

- `respostas` – vetor mantido pelo iniciador para detecção da terminação da construção consistente.

Início da construção consistente

Uma construção consistente é iniciada por um processo p_{ini} quando este armazena um *checkpoint* provisório e envia mensagens de requisição para os processos selecionados em `VP`. A mensagem de requisição propaga o índice do *checkpoint* salvo por p_{ini} (índice da construção consistente) e seu `VP`.

Recepção da mensagem de requisição

Quando um processo p_i recebe uma mensagem de requisição de p_k, p_i procura por um *checkpoint* c consistente com o último *checkpoint* salvo por p_k, isto é, o *checkpoint* com índice maior ou igual ao índice que p_k conhece de p_i. Se p_i:

- não possui c, então um novo *checkpoint* provisório é salvo.
- possui c e c é um *checkpoint* mutável, então c é transformado em provisório.

Após salvar um *checkpoint* provisório, o processo p_i calcula o conjunto de participantes $\texttt{VP}_i$. Para reduzir o número de mensagens de requisição, p_i envia uma mensagem de requisição para p_j se p_j faz parte de $\texttt{VP}_i$ e $\texttt{req.VP[}j\texttt{]} < \texttt{VP}_i\texttt{[}j\texttt{]}$, ou seja, p_j ainda não faz parte do conjunto de participantes da construção consistente com o índice conhecido por p_i. Note que se c é permanente, $\texttt{VP}_i$ não inclui nenhum processo como participante e não é necessário propagar requisição.

Todo processo que recebe uma mensagem de requisição, envia uma mensagem de resposta ao iniciador com a informação de quais processos foram adicionados como seus participantes e o índice do seu *checkpoint* provisório.

Recepção da mensagem de resposta

Um processo p_{ini} processa uma mensagem de resposta apenas se a construção consistente que iniciou não foi encerrada, ou seja, se possui um *checkpoint* provisório com o índice da construção consistente. Ao receber uma mensagem de resposta de p_k, p_{ini} atualiza o seu vetor `respostas` com o índice recebido por p_k e atualiza o seu vetor `VP` fazendo a união de seus participantes com os participantes conhecidos por p_k. Enquanto um processo participante p_j não enviar mensagem de resposta ao iniciador com o índice esperado a construção consistente não é encerrada. Assim, quando p_{ini} recebe mensagens

de todos os participantes de sua construção consistente, p_{ini} transforma seu *checkpoint* provisório em permanente, propaga mensagens de liberação e remove todos os *checkpoint* salvos antes do novo *checkpoint* permanente.

Recepção da mensagem de liberação

Um processo p_i, ao receber uma mensagem de liberação, transforma o seu *checkpoint* provisório em permanente e remove os *checkpoints* salvos antes dele.

A descrição do protocolo BCS-NBS é apresentada pelo Protocolo 5.2.

Protocolo 5.2 BCS-NBS (declarações)

Variáveis do processo:
- enviou ≡ booleano
- VI ≡ vetor[$0 \ldots n-1$] de inteiros
- VP ≡ vetor[$0 \ldots n-1$] de inteiros
- respostas ≡ vetor[$0 \ldots n-1$] de inteiros
- pid ≡ inteiro

Tipos de Mensagem:
- aplicação:
  - ind ≡ inteiro
- requisição:
  - iind ≡ inteiro
  - ipid ≡ inteiro
  - VP ≡ vetor[$0 \ldots n-1$] de inteiros
- resposta:
  - ind ≡ inteiro
  - VP ≡ vetor[$0 \ldots n-1$] de inteiros
- liberação:
  - ind ≡ inteiro

5.4.2 Exemplo

O protocolo BCS-NBS utiliza um mecanismo semelhante a relógios lógicos para sincronizar as atividades de *checkpointing* e garantir a construção de *checkpoints* globais consistentes. A Figura 5.18 ilustra um exemplo de execução do protocolo BCS-NBS. O processo p_3 inicia uma construção consistente e salva o seu *checkpoint* como permanente, pois não necessita propagar mensagens de requisição. Quando p_1 recebe a mensagem da aplicação de p_3, p_1 induz um *checkpoint* mutável utilizando a mesma regra que protocolo quase-síncrono BCS-Aftersend usa para salvar um *checkpoint* forçado. Da mesma maneira, p_0 salva um *checkpoint* mutável. Quando p_0 inicia uma construção consistente, salva um *checkpoint* provisório e envia mensagens de requisição para p_1 e p_2. Tanto o processo p_1 quanto o processo p_2 salvam um *checkpoint* provisório e enviam uma mensagem de requisição para p_3. O processo p_3 recebe a mensagem de requisição de p_2 e como o último *checkpoint* permanente de p_3 não é conhecido por p_2, p_3 não salva um novo *checkpoint* provisório. Porém, como o índice do iniciador é maior que índice de p_3, p_3 salva um *checkpoint* mutável e atualiza o seu índice com o índice do iniciador. Quando p_3 recebe a mensagem

Protocolo 5.2 BCS-NBS

Início:
  $\forall i$: VI[i] ← 0
  salvaCkpt(permanente, 1)

Envio da mensagen da aplic. (m) para p_k
  enviou ← *verdadeiro*
  m.ind ← VI[pid]
  envia mensagem da aplicação (m)

Recepção da mensagem da aplic. (m) de p_k:
  se (m.ind > VI[pid])
    se (enviou)
      salvaCkpt(mutável, m.ind)
    VI[pid] ← m.ind
  VI[k] ← max(m.ind, VI[k])
  processa a mensagem da aplicação (m)

Início da construção consistente:
  c ← salvaCkpt(provisório, VI[pid] + 1)
  VP ← verificaVP(c)
  se ($\exists i \neq$ pid: VP[i] > 0)
    $\forall i$: se (VP[i] > 0)
        req.VP ← VP
        req.ipid ← pid
        req.iind ← VI[pid]
        envia requisição (req) para p_i
    $\forall i$: respostas[i] ← 0
  senão
    transformaCkpt(c, permanente)

Recepção da requisição (req) de p_k:
  c ← procuraCkpt(req.VP[pid])
  se (c é nulo)
    c ← salvaCkpt(provisório, req.iind)
  se (c é mutável)
    transformaCkpt(c, provisório)
  res.ind ← c.VI[pid]
  res.VP ← verificaVP(c)
  propReq(res.VP, req.VP, req.ipid, req.iind)
  se (VI[pid] < req.iind)
    se (enviou)
      salvaCkpt(mutável, req.iind)
    VI[pid] ← req.iind
  envia resposta (res) para $p_{req.ipid}$

Recepção da resposta (res) de p_k:
  c ← procuraCkpt(VP[pid])
  se (c *não* é *nulo* *e* c é provisório)
    VP ← max (VP, res.VP)
    respostas[k] ← max (respostas[k], res.VP[k])
    se ($\nexists i \neq$ pid: VP[i] > 0 *e*
        VP[i] > respostas[i])
      $\forall i$: se (VP[i] > 0 *e* i ≠ pid)
          rel.ind ← respostas[i]
          envia liberação (lib) para p_i
      transformaCkpt(c, permanente)

Recepção da liberação (lib) de p_k:
  c ← procuraCkpt(lib.ind)
  se (c.VI[pid] = lib.ind *e* c é provisório)
    transformaCkpt(c, permanente)

salvaCkpt(tipo, num)
  armazena um checkpoint de acordo com tipo
  VI[pid] ← num
  enviou ← *falso*
  retorna ckpt salvo

transformaCkpt(c, tipo)
  transforma c de acordo com tipo
  se (tipo = permanente)
    remove os ckpts salvos antes de c

procuraCkpt(ind)
  retorna primeiro checkpoint c com
    c.VI[pid] ≥ ind

verificaVP(c)
  p ← primeiro checkpoint permanente
  $\forall i$: auxVP[i] ← 0
  $\forall i$: se (c.VI[i] > p.VI[i])
    auxVP[i] ← c.VI[i]
  retorna auxVP

propReq(pInd, reqInd, ini, iniInd)
  $\forall i$: se (i ≠ pid *e* pInd[i] > reqInd[i])
      req.VP ← max(pInd, reqInd)
      req.ipid ← ini
      req.iind ← iniInd
      envia mensagem de requisição (req) para p_i

de requisição de p_1, p_3 transforma o seu *checkpoint* mutável em provisório e envia uma mensagem de resposta ao iniciador. O iniciador p_0 encerra sua construção consistente após receber as mensagens de resposta de p_1, p_2 e p_3 (com o índice do novo *checkpoint* provisório). O *checkpoint* global consistente formado pela construção consistente iniciada por p_0 é representada pela linha Σ.

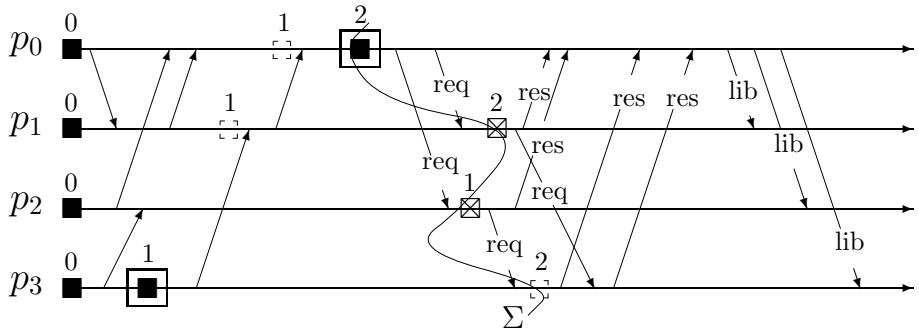


Figura 5.18: Exemplo de padrão gerado pelo protocolo BCS-NBS

5.4.3 Prova de Correção

Teorema 14 *O protocolo BCS-NBS garante que, a qualquer momento, existe um checkpoint global consistente formado por checkpoints estáveis.*

Prova: Por contradição, vamos supor que, em um determinado instante, não existe um *checkpoint* global consistente formado por *checkpoints* estáveis. Então, existem c_a^α e c_b^β tais que c_a^α é o último *checkpoint* estável de p_a, c_b^β é o primeiro *checkpoint* estável de p_b e $c_a^\alpha \rightarrow c_b^\beta$ devido a uma mensagem m (Figure 5.19).

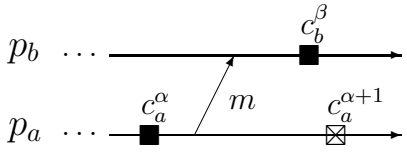


Figura 5.19: $c_a^\alpha \rightarrow c_b^\beta$

Sabemos pela política de coleta de lixo que o primeiro *checkpoint* mantido por um processo é permanente e portanto, c_b^β é permanente. Então c_b^β foi salvo durante uma construção consistente $\mathcal{C}$. Quando c_b^β foi salvo por p_b, p_b propaga uma mensagem de

requisição para p_a ou um outro processo participante de $\mathcal{C}$ que conhece c_a^α já enviou uma mensagem de requisição para p_a. Portanto, p_a é um participante de $\mathcal{C}$ e como c_b^β é um *checkpoint* permanente, p_a já recebeu uma mensagem de requisição após o envio de m. Pela Propriedade 1, se $c_a^\alpha \rightarrow c_b^\beta$ então o índice do *checkpoint* c_b^β é maior que o índice de c_a^α. Portanto, quando p_a recebe a mensagem de requisição durante $\mathcal{C}$, p_a salva um novo *checkpoint* provisório ou possui um *checkpoint* provisório salvo após o envio de m. Podemos concluir então que c_a^α não é o último *checkpoint* estável de p_a, o que contradiz a hipótese. □

5.5 Resultados de Simulação

O simulador de protocolos de *checkpointing* ChkSim foi utilizado como experimento inicial para comparar os protocolos propostos nesta tese e os existentes na literatura [43]. Este simulador foi originalmente proposto para comparar o desempenho dos protocolos quase-síncronos. O simulador gera uma seqüencia de eventos de comunicação (envio e recepção de mensagens) e *checkpoints* da aplicação. Adicionando-se mensagens de controle, foi possível simular os protocolos síncronos não-bloqueantes. O resultado foi obtido simulando-se um sistema com 16 nós interligados por meio de um grafo de comunicação não completo, como ilustra a Figura 5.20, após a execução de 12.000 eventos por processo. As amostras foram obtidas com intervalos de *checkpoints* da aplicação a cada 4 a 112 eventos de comunicação e cada protocolo foi executado 10 vezes.

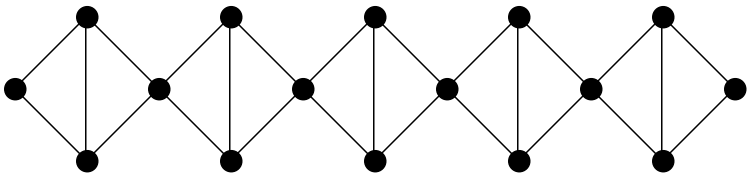


Figura 5.20: Topologia utilizada na simulação

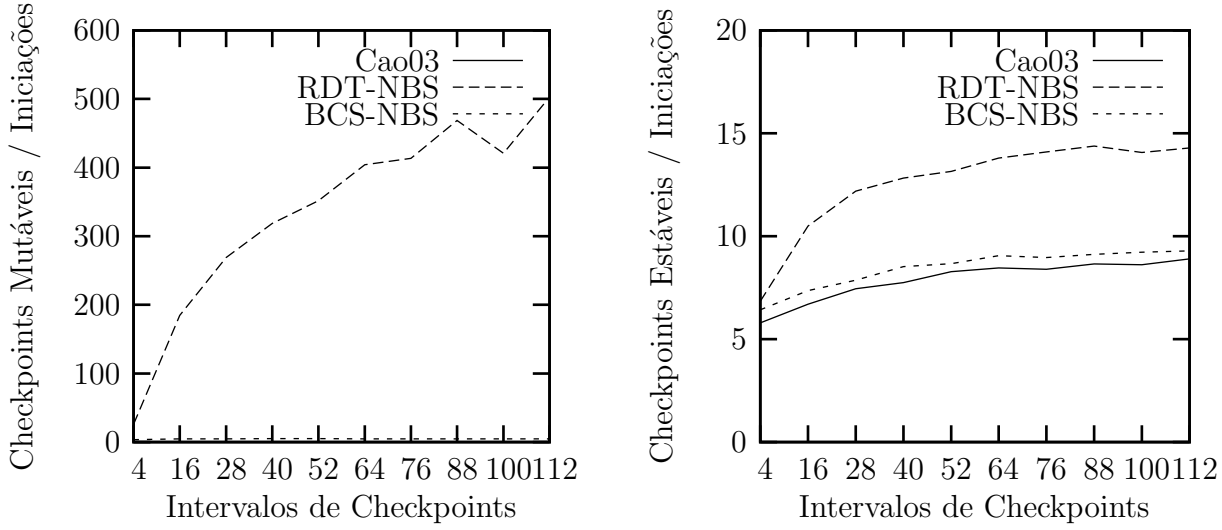
Dentre as várias topologias testadas, esta foi escolhida pois simula o comportamento de uma rede em que a comunicação ocorre entre grupos de processos. Redes em que existe troca de mensagens frequentes entre todos os processos da aplicação levariam a adoção de um protocolo global e não minimal.

5.5.1 Único Iniciador

O único protocolo síncrono não-bloqueante que utiliza *checkpoints* mutáveis existente na literatura, chamado nesta Seção de Cao03, não permite a presença de iniciadores

concorrentes [8]. A comparação do protocolo Cao03 com os protocolos não-bloqueantes RDT-NBS e BCS-NBS propostos nesta tese foi realizada implementando-se o mecanismo de passagem de ficha para garantir que apenas o processo que possui a ficha possa iniciar uma construção consistente (uma única iniciação a cada instante). Porém, notamos que os diferentes mecanismos na propagação das mensagens de controle influenciou os pontos de armazenamento dos *checkpoints* dos iniciadores. Portanto, não conseguimos comparar estes protocolos utilizando o mesmo padrão de *checkpoints* e mensagens. Assim, todos os valores mostrados a seguir são proporcionais ao número de construções consistentes executados a cada protocolo.

A Figura 5.21 mostra o número de *checkpoints* salvos pelos protocolos. O protocolo BCS-NBS apresentou um número menor de *checkpoints* mutáveis em relação ao protocolo RDT-NBS (Figura 5.21 (a)). Este resultado é esperado já que simulações dos protocolos quase-síncronos mostram que o protocolo BCS-Aftersend salva menos *checkpoints* forçados que o protocolo FDAS [44, 43]. Já o protocolo Cao03 induziu um baixo número de *checkpoints* mutáveis pois este protocolo salva *checkpoints* mutáveis apenas quando uma construção consistente está em andamento e uma mensagem da aplicação com informação sobre construção consistente chega antes da mensagem de requisição. A Figura 5.21 (b) mostra que o protocolo Cao03 salva um número menor de *checkpoints* estáveis seguido pelo BCS-NBS e então pelo RDT-NBS.



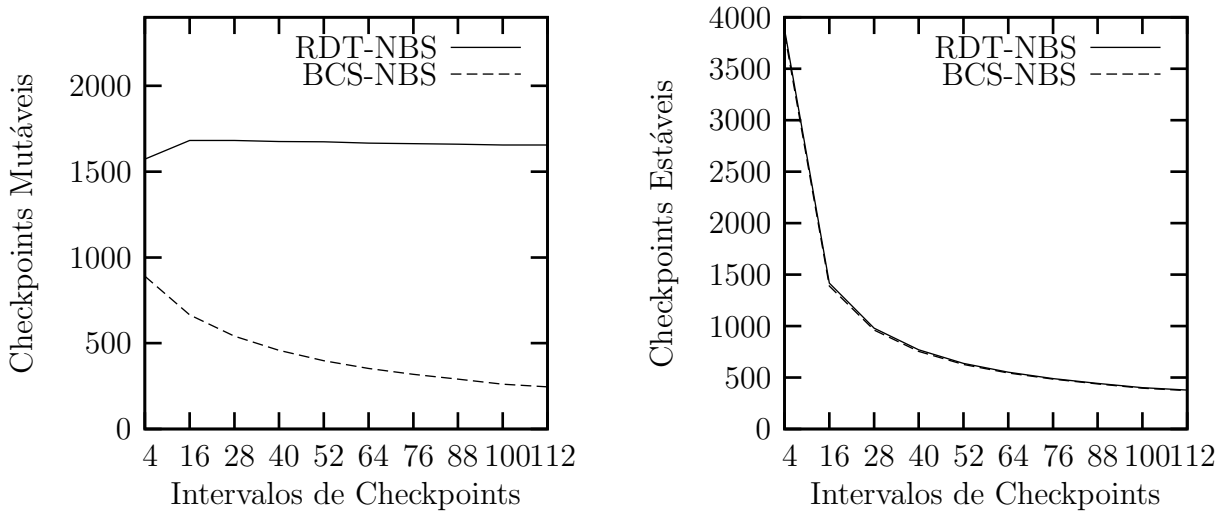
(a) *Checkpoints* mutáveis

(b) *Checkpoints* estáveis

Figura 5.21: Um único iniciador

5.5.2 Iniciadores Concorrentes

A comparação dos protocolos RDT-NBS e BCS-NBS que permitem iniciações concorrentes foi realizada utilizando-se exatamente os mesmos padrões de *checkpoints* e mensagens. Assim como o protocolo BCS-Aftersend salva menos *checkpoints* forçados comparado ao FDAS, o protocolo BCS-NBS apresentou um número menor de *checkpoints* mutáveis em relação ao protocolo RDT-NBS (Figura 5.22 (a)). Porém o número de *checkpoints* salvos em memória estável (*checkpoints* provisórios e *checkpoints* mutáveis transformados em provisórios) por ambos os protocolos foi praticamente o mesmo (Figura 5.22 (b)).



(a) *Checkpoints* mutáveis

(b) *Checkpoints* estáveis

Figura 5.22: *Checkpoints* salvos na presença de iniciadores concorrentes

As mensagens de controle são compostas por mensagens de requisição, resposta e liberação. Apesar do protocolo RDT-NBS enviar as mensagens de controle diretamente do iniciador aos participantes de uma construção consistente, o número de mensagens de controle utilizadas por esse protocolo foi maior comparado ao protocolo BCS-NBS (Figura 5.23). Devemos lembrar que esses dois protocolos salvam *checkpoints* mutáveis em pontos diferentes, o que influencia o conjunto de participantes para a construção de *checkpoints* globais consistentes (Seção 5.2).

O simulador nos permitiu comparar também os protocolos síncronos não-bloqueantes RDT-NBS e BCS-NBS com os respectivos protocolos quase-síncronos FDAS e BCS-Aftersend. Nos protocolos síncronos não-bloqueantes é permitida a presença de iniciadores

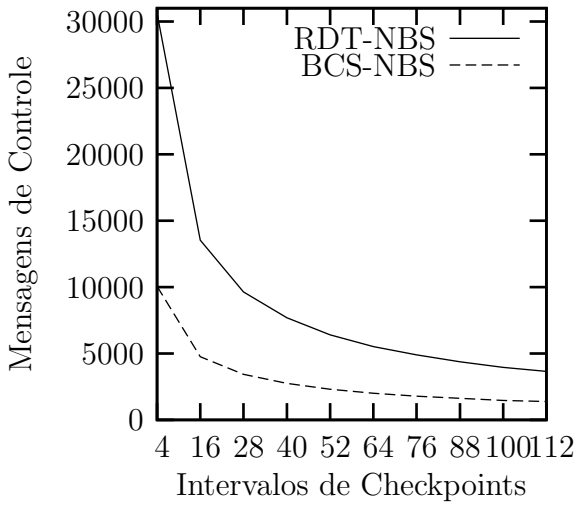


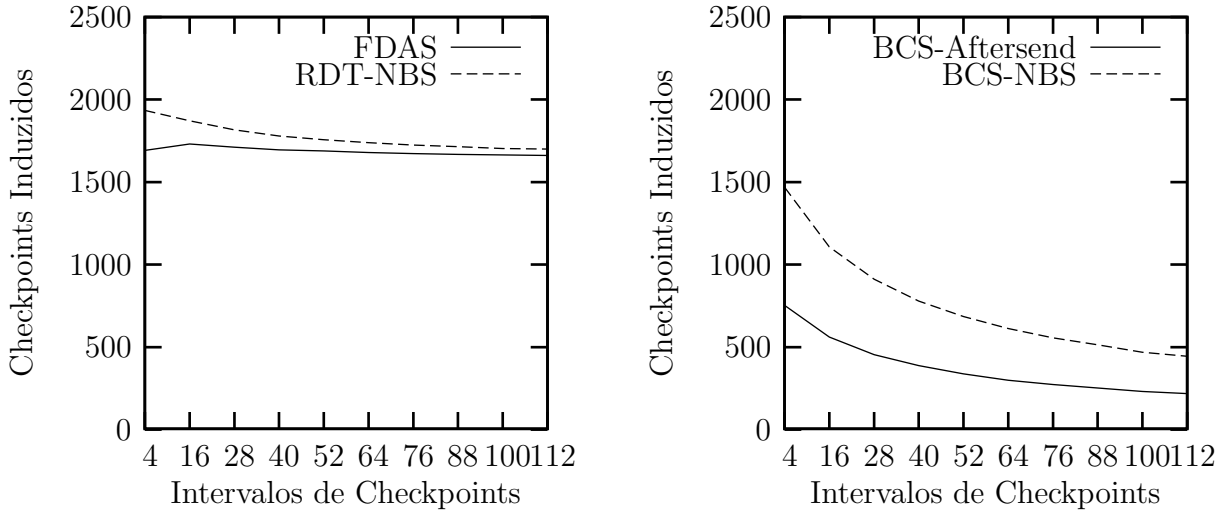
Figura 5.23: Mensagens de controle na presença de iniciadores concorrentes

concorrentes e para cada *checkpoint* básico armazenado por um protocolo quase-síncrono, uma nova construção consistente é iniciada pelo protocolo síncrono. A Figura 5.24 ilustra o número de *checkpoints* forçados armazenado pelos protocolos quase-síncronos e o número de *checkpoints* induzidos (mutáveis e estáveis) pelos protocolos síncronos. Os protocolos síncronos apresentam um número maior de *checkpoints* induzidos comparados com os respectivos protocolos quase-síncronos pois para cada *checkpoint* básico salvo, um novo *checkpoint* global consistente é construído. Assim, os protocolos síncronos não-bloqueantes mantêm *checkpoints* globais consistentes mais recentes que os protocolos quase-síncronos.

Apesar dos protocolos síncronos salvarem mais *checkpoints* que os respectivos protocolos quase-síncronos, a maioria dos *checkpoints* são salvos primeiro como mutáveis e apenas alguns são utilizados para formar *checkpoints* globais consistentes, ou seja, são transformados em estáveis. De acordo com a Figura 5.25, mais da metade dos *checkpoints* mutáveis são descartados antes de serem transformados em estáveis o que implica que os protocolos síncronos armazenam um número menor de *checkpoints* estáveis.

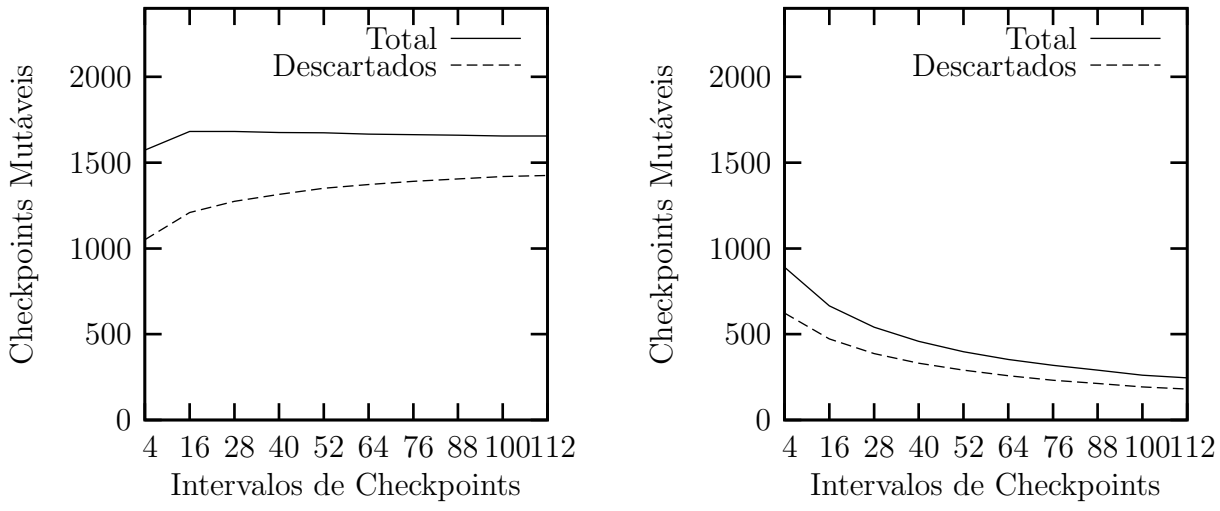
A Figura 5.26 mostra o número total de *checkpoints* estáveis salvos por esses protocolos. Podemos notar que os protocolos síncronos não-bloqueantes salvam um número menor de *checkpoints* em memória estável.

Os protocolos síncronos descartam *checkpoints* após a execução de uma construção consistente enquanto os protocolos quase-síncronos requerem a implementação de um protocolo independente para a coleta de lixo. Para qualquer protocolo RDT, incluindo o



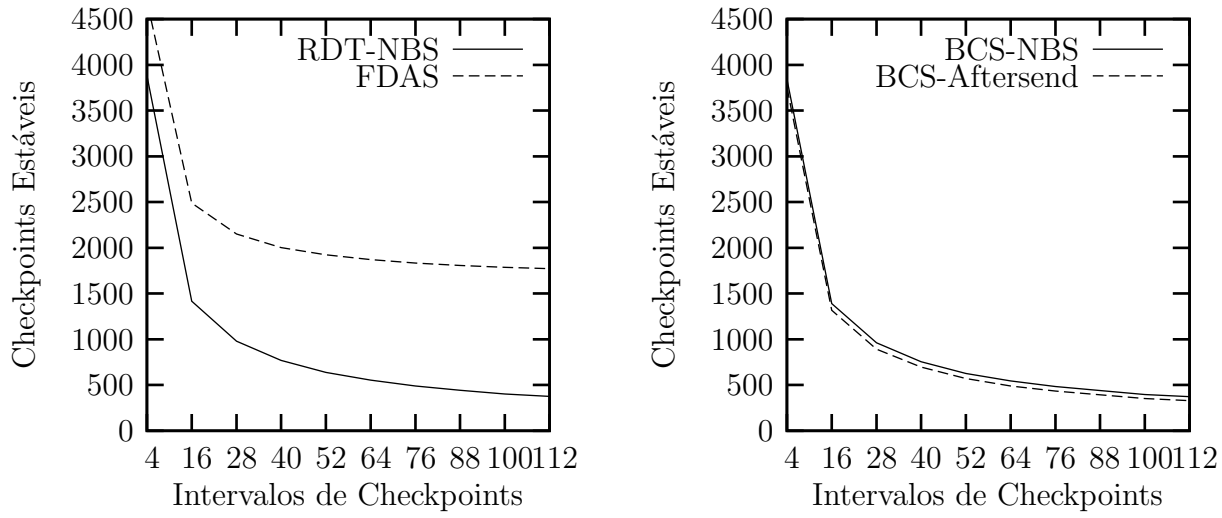
(a) FDAS e RDT-NBS (b) BCS-Aftersend e BCS-NBS

Figura 5.24: *Checkpoints* induzidos



(a) RDT-NBS (b) BCS-NBS

Figura 5.25: *Checkpoints* mutáveis



(a) FDAS e RDT-NBS

(b) BCS-Aftersend e BCS-NBS

Figura 5.26: *Checkpoints* estáveis

FDAS, pode-se usar o protocolo de coleta de lixo proposto por Schmidt et al [37].

5.6 Sumário

Os protocolos síncronos mais simples requerem que todos os processos armazenem *checkpoints* durante uma construção consistente. Na tentativa de reduzir o *overhead* associado aos protocolos síncronos, os protocolos síncronos minimais induzem apenas um número minimal de processos a salvarem *checkpoints*. Porém os protocolos minimais são bloqueantes, ou seja, os processos envolvidos devem suspender suas aplicações durante a construção consistente. Alguns protocolos síncronos que reduzem o número de *checkpoints* a cada construção consistente e não requerem o bloqueio dos processos foram propostos na literatura [30, 9]. Esses protocolos são baseados em protocolos síncronos minimais e não permitem a presença de iniciadores concorrentes. Recentemente, Cao e Singhal introduziram um novo tipo de *checkpoint* armazenado em memória não-estável (*checkpoint* mutável) para desenvolver um protocolo que garante um número minimal de *checkpoints* em memória estável.

Notamos que a minimalidade no número de *checkpoints* estáveis depende do ponto de execução onde os *checkpoints* mutáveis são salvos e provamos que não é possível garantir um número mínimo de *checkpoints* durante toda a execução da aplicação. Além disso,

provamos que é impossível garantir minimalidade no número de *checkpoints* estáveis na presença de iniciadores concorrentes.

Neste Capítulo, apresentamos as características dos protocolos síncronos não-bloqueantes e notamos que várias dessas características são semelhantes às dos protocolos quase-síncronos. Assim, mostramos que é possível desenvolver protocolos síncronos não-bloqueantes baseando-se em protocolos quase-síncronos. Esses protocolos permitem a presença de iniciadores concorrentes, já que em protocolos quase-síncronos não existe o conceito de um único iniciador, pois os processos têm autonomia para salvar *checkpoints* da aplicação.

O protocolo RDT-NBS é um protocolo síncrono não-bloqueante baseado no protocolo quase-síncrono FDAS, porém pode ser facilmente modificado baseado-se em qualquer protocolo da classe RDT. Este protocolo utiliza vetores de dependências para rastrear as dependências entre *checkpoints*. Assim, o iniciador conhece todos os participantes de sua construção consistente e as mensagens são enviadas diretamente do iniciador aos participantes e dos participantes ao iniciador, o que permitiu o desenvolvimento de métodos simples em todas as fases de *checkpointing*.

O protocolo BCS-NBS é um protocolo síncrono não-bloqueante baseado no protocolo quase-síncrono BCS. Este protocolo requer a propagação de apenas um inteiro com as mensagens da aplicação e número de *checkpoints* induzidos é menor do que qualquer protocolo baseado em protocolos quase-síncronos da classe RDT. Este protocolo leva em consideração todas as características citadas na Seção 5.1.2 e não sofre o efeito avalanche de *checkpoints*.

Um simulador foi utilizado para comparar esses dois protocolos. O protocolo BCS-NBS apresentou um número menor de *checkpoints* mutáveis em comparação ao protocolo RDT-NBS, porém, o número de *checkpoints* estáveis armazenados por ambos os protocolos foi praticamente o mesmo. Apesar dos protocolos síncronos não-bloqueantes salvarem um número maior de *checkpoints* comparado aos seus respectivos protocolos quase-síncronos, notamos que mais da metade dos *checkpoints* salvos como mutáveis são descartados antes de serem transferidos para memória estável e o número de *checkpoints* salvos em memória estável pelos protocolos síncronos é menor. Além disso, os protocolos síncronos implementam um protocolo de coleta de lixo simples enquanto os protocolos quase-síncronos necessitam de um protocolo de coleta de lixo independente ao protocolo de *checkpointing*.

Capítulo 6

Conclusão

Checkpointing é um mecanismo utilizado para garantir tolerância a falhas em sistemas distribuídos. Um protocolo de *checkpointing* é responsável pelo armazenamento de estados em memória estável, chamados de *checkpoints*, que podem ser utilizados para o restabelecimento da computação após a ocorrência de uma falha. Os protocolos de *checkpointing* podem ser classificados como assíncronos, quase-síncronos ou síncronos. Nesta tese, mostramos que é possível aplicar alguns conceitos utilizados na teoria dos protocolos quase-síncronos para analisar e desenvolver protocolos síncronos, especialmente protocolos síncronos não-bloqueantes que reduzem o número de *checkpoints* a cada construção consistente e permitem a presença de iniciadores concorrentes.

Os protocolos de *checkpointing* síncronos que induzem um número minimal de processos a armazenarem *checkpoints* durante uma construção consistente são chamados de minimais. O primeiro protocolo minimal proposto necessita manter os processos bloqueados durante uma construção consistente e requer um alto número de mensagens de controle [20]. Notamos na literatura, um esforço em tentar reduzir o número de mensagens de controle nos protocolos minimais [22, 8, 35, 36] ou eliminar o mecanismo de bloqueio dos processos [8, 33].

Existem duas abordagens para o desenvolvimento de protocolos minimais. Na primeira abordagem, um processo iniciador salva um *checkpoint* e envia mensagens de requisição para um conjunto de processos. Cada processo que recebe uma mensagem de requisição, salva um *checkpoint* e propaga mensagens de requisição para um novo conjunto de processos formando assim, uma propagação da mensagens de requisição em níveis. Na segunda abordagem, o iniciador faz um *broadcast* de mensagens de bloqueio e, em um primeiro instante, todos os processos ficam bloqueados até o iniciador definir quais processos devem armazenar *checkpoints* para a construção de um *checkpoint* global consistente.

Independente da abordagem escolhida para propagar as mensagens de requisição, os protocolos minimais devem utilizar um mecanismo para rastrear as dependências entre

checkpoints. Provamos que protocolos minimais que anotam apenas a recepção de mensagens, mas não os intervalos de *checkpoints* de origem dessas mensagens não garantem minimalidade no número de *checkpoints*. Os protocolos síncronos minimais propostos nesta tese utilizam vetores de dependências para rastrear as dependências entre *checkpoints* e garantir um número minimal de *checkpoints*. O primeiro protocolo, chamado de VD-minimal, requer um número menor de mensagens de controle e utiliza um novo mecanismo de detecção de terminação [35]. O protocolo Broad-minimal utiliza a abordagem *broadcast* para garantir um número minimal de *checkpoints* e é uma correção de um protocolo existente na literatura [36].

Os protocolos minimais são bloqueantes, ou seja, é impossível desenvolver um protocolo síncrono não-bloqueante que armazena um número minimal de *checkpoints* a cada construção consistente [10]. Para amenizar este problema, Cao e Singhal propuseram um novo tipo de *checkpoint* chamado de *checkpoint* mutável [7, 8]. Um *checkpoint* mutável é um *checkpoint* que pode ser armazenado em memória não-estável e é utilizado para reduzir a quantidade de dados que devem ser transferidos para memória estável. Utilizando *checkpoints* mutáveis é possível garantir um número minimal de *checkpoints* estáveis a cada construção consistente. Porém, provamos que na presença de iniciadores concorrentes não é possível garantir minimalidade no número de *checkpoints* estáveis. Além disso, provamos que é impossível garantir número mínimo de *checkpoints* estáveis em um protocolo não-bloqueante que usa *checkpoints* mutáveis durante toda a execução da aplicação.

Notamos que muitas das características de um protocolo síncrono não-bloqueante que utiliza *checkpoints* mutáveis são semelhantes às características de protocolos quase-síncronos. Ao utilizarmos protocolos quase-síncronos para desenvolver protocolos síncronos não-bloqueantes com *checkpoints* mutáveis, obtivemos protocolos simples, de fácil compreensão e que permitem iniciadores concorrentes. Além disso, as provas de correção foram escritas baseando-se no conhecimento da teoria dos protocolos quase-síncronos. O protocolo RDT-NBS é um protocolo síncrono não-bloqueante baseado no protocolo FDAS da classe RDT do modelo quase-síncrono [33]. Este protocolo é simples em todas as fases de *checkpointing*, porém requer a propagação de um vetor de inteiros com as mensagens da aplicação e armazena um alto número de *checkpoints* mutáveis. Já o protocolo BCS-NBS é baseado no protocolo BCS da classe ZCF do modelo quase-síncrono [25]. Este protocolo possui a fase de requisições mais complexa que o protocolo RDT-NBS, pois a propagação das mensagens de requisição é feita em níveis. Porém, o protocolo BCS-NBS requer a propagação de apenas um número inteiro com as mensagens da aplicação e armazena um número menor de *checkpoints* mutáveis comparado ao protocolo RDT-NBS. Testes em um simulador foram realizados para validar esses resultados.

Os protocolos síncronos não-bloqueantes requerem o armazenamento de um número

maior de *checkpoints* comparado aos protocolos quase-síncronos, pois induzem *checkpoints* na construção de *checkpoints* globais consistentes. Porém, a maioria dos *checkpoints* salvos por esses protocolos são *checkpoints* mutáveis descartados antes de serem transferidos para memória estável. Por esse motivo, os protocolos síncronos armazenam um número menor de *checkpoints* estáveis em relação aos seus respectivos protocolos quase-síncronos.

Os protocolos síncronos implementam uma coleta de lixo bem simples, já que para cada *checkpoint* requerido pela aplicação, um novo *checkpoint* global consistente é construído e os *checkpoints* salvos anteriormente podem ser removidos. Assim, esses protocolos mantêm um menor número de *checkpoints* em memória estável. Em especial, em protocolos síncronos bloqueantes, um processo necessita manter no máximo, dois *checkpoints* em memória estável em um determinado instante de tempo. Além disso, os protocolos síncronos garantem *checkpoints* globais consistentes mais recentes que os protocolos quase-síncronos.

Nesta mesma linha de pesquisa, alguns pontos ainda podem ser investigados. Outros protocolos síncronos não-bloqueantes baseados em diferentes protocolos quase-síncronos poderiam ser desenvolvidos para analisar e avaliar o desempenho e custo desses protocolos. Além disso, o simulador poderia ser modificado para permitir também os protocolos síncronos bloqueantes. Acreditamos que se desacoplarmos a detecção da terminação em protocolos síncronos não-bloqueantes e implementarmos um protocolo de coleta de lixo, poderemos obter protocolos mais simples e com menor número de mensagens de controle.

Apêndice A

Protocolos Quase-Síncronos

Nesta seção, apresentamos os protocolos quase-síncronos analisados durante o desenvolvimento deste estudo.

A.1 CASBR

O protocolo CASBR (*Checkpoint-After-Send-Before-Receive*) pertence à classe SZPF dos protocolos quase-síncronos [45].

Protocolo A.1 CASBR

Início:
salvaCheckpoint()

Envio da mensagem (m) da aplic. para p_k:
envia a mensagem (m) da aplicação
salvaCheckpoint()

Recepção da mensagem (m) da aplic. de p_k:
salvaCheckpoint()
entrega a mensagem m para a aplicação

salvaCheckpoint()
grava um checkpoint em dispositivo estável

A.2 CBR

O protocolo CBR (*Checkpoint-Before-Receive*) pertence à classe SZPF dos protocolos quase-síncronos [45].

Protocolo A.2 CBR

Início:
salvaCheckpoint()

Envio da mensagem (m) da aplic. para p_k:
envia a mensagem (m) da aplicação
salvaCheckpoint()

Recepção da mensagem (m) da aplic. de p_k:
salvaCheckpoint()
entrega a mensagem m para a aplicação

salvaCheckpoint()
grava um checkpoint em dispositivo estável

A.3 NRAS

O protocolo NRAS (*No-Receive-After-Send*) pertence à classe SZPF dos protocolos quase-síncronos [45].

Protocolo A.3 NRAS

Variáveis do processo:
enviou ≡ booleano

Início:
salvaCheckpoint()

Envio da mensagem (m) da aplic. para p_k:
enviou ← *verdadeiro*
envia a mensagem (m) da aplicação

Recepção da mensagem (m) da aplic. de p_k:
se (enviou)
salvaCheckpoint()
entrega a mensagem m para a aplicação

salvaCheckpoint()
enviou ← *falso*
grava um checkpoint em dispositivo estável

A.4 FDI

O protocolo FDI (*Fixed-Dependency-Interval*) pertence à classe ZPF dos protocolos quase-síncronos [45].

Protocolo A.4 FDI

Variáveis do processo:
VD ≡ vetor[0 . . . $n-1$] de inteiros
pid ≡ inteiro

Início:
$\forall i$: VD[i] ← 0
salvaCheckpoint()

Envio da mensagem (m) da aplic. para p_k:
m.VD ← VD
envia a mensagem (m) da aplicação

Recepção da mensagem (m) da aplic. de p_k:
se (m.VD[k] > VD[k])
salvaCheckpoint()
$\forall i$: VD[i] ← max(VD[i], m.VD[i])
entrega a mensagem m para a aplicação

salvaCheckpoint()
grava um checkpoint em dispositivo estável
VD[pid] ← VD[pid] + 1

A.5 RDT-Partner

O protocolo RDT-Partner pertence à classe ZPF dos protocolos quase-síncronos [15].

Protocolo A.5 RDT-Partner

Variáveis do processo:
VD ≡ vetor[0 . . . $n-1$] de inteiros
simples ≡ vetor[0 . . . $n-1$] de booleano
parceiro ≡ inteiro
pid ≡ inteiro

Início:
$\forall i$: VD[i] ← 0
salvaCheckpoint()

Envio da mensagem (m) da aplic. para p_k:
se (parceiro = -1)
parceiro ← k
senão se (parceiro ≠ k)
parceiro ← -2
m.VD ← VD
m.simplesRec ← simples[k]
envia a mensagem (m) da aplicação

Recepção da mensagem (m) da aplic. de p_k:
se (m.VD[k] > VD[k])
se (parceiro ≠ -2 *e* (parceiro ≠ k *ou*
(parceiro = k *e* m.VD[pid] = VD[pid] *e*
não m.simplesRec)))
salvaCheckpoint()
simples[k] ← *verdadeiro*
$\forall i$: VD[i] ← max(VD[i], m.VD[i])
entrega a mensagem m para a aplicação

salvaCheckpoint()
grava um checkpoint em dispositivo estável
VD[pid] ← VD[pid] + 1
$\forall i$: simples[i] ← *falso*
simples[pid] ← *verdadeiro*
parceiro ← -1

A.6 RDT-Minimal

O protocolo RDT-Minimal pertence à classe ZPF dos protocolos quase-síncronos [18].

Protocolo A.6 RDT-Minimal

Variáveis do processo:
VD ≡ vetor[0 . . . $n-1$] de inteiros
simples ≡ vetor[0 . . . $n-1$] de booleano
igual ≡ vetor[0 . . . $n-1$] de booleano
enviou ≡ vetor[0 . . . $n-1$] de booleano
fase ≡ inteiro
pid ≡ inteiro

Início:
$\forall i$: VD[i] ← 0
salvaCheckpoint()

Envio da mensagem (m) da aplic. para p_k:
enviou[k] ← *verdadeiro*
se (fase = 0)
fase ← 1
m.VD ← VD
m.simples ← simples
m.igual ← igual
envia a mensagem (m) da aplicação

Recepção da mensagem (m) da aplic. de p_k:
se (m.VD[k] > VD[k])
se (testaForçado(m))
salvaCheckpoint()
$\forall i$: se (m.VD[i] > VD[i])
VD[i] ← m.VD[i]
simples[i] ← m.simples[i]
senão se (m.VD[i] = VD[i])
simples[i] ← simples[i] *ou* m.simples[i]
se (m.VD[k] = VD[k])
$\forall i$: igual[i] ← igual[i] *ou* m.igual[i]
fase ← 2
entrega a mensagem m para a aplicação

salvaCheckpoint()
grava um checkpoint em dispositivo estável
VD[pid] ← VD[pid] + 1
$\forall i$: simples[i] ← *falso*
$\forall i$: igual[i] ← *falso*
$\forall i$: enviou[i] ← *falso*
simples[pid] ← *verdadeiro*
igual[pid] ← *verdadeiro*
fase ← 0

testaForçado(mensagem m)
se (fase = 0) retorna *falso*
se (fase = 2 *ou* m.VD[pid] = VD[pid] *e não*
m.simples[pid])
retorna *verdadeiro*
$\forall i$: se (enviou[i] *e não* m.igual[i])
retorna *falso*

Apêndice B

Protocolos Minimais

Nesta seção, apresentamos os protocolos síncronos minimais analisados durante o desenvolvimento deste trabalho.

B.1 Leu e Bhargava

O protocolo proposto por Leu e Bhargava garante minimalidade por meio da construção de uma árvore que representa as dependências entre *checkpoints* do intervalo de *checkpoints* corrente [22].

Protocolo B.1 Leu e Bhargava (declarações)

Variáveis do processo:
VI ≡ vetor$[0 \ldots n-1]$ de inteiros
filhos ≡ vetor$[0 \ldots n-1]$ de booleanos
pai ≡ inteiro
num_filhos ≡ inteiro
cont_req ≡ inteiro
bloqueado ≡ booleano
pid ≡ inteiro

Tipos de mensagem:
aplicação: ind ≡ inteiro
requisição: ind ≡ inteiro
confirmação: filho ≡ booleano
resposta
liberação

Protocolo B.1 Leu e Bhargava

Início:
  $\forall i$: VI[i] ← 0
  $\forall i$: filhos[i] ← *falso*
  num_filhos ← 0
  cont_req ← 0
  bloqueado ← *falso*
  salvaCkpt(permanente)

Envio da mensagem (m) da aplic. para p_k:
  se (*não* bloqueado)
    m.ind ← VI[pid]
  envia mensagem da aplicação (m)

Recepção da mensagem (m) da aplic. de p_k:
  se (*não* bloqueado)
    VI[k] ← m.ind
    processa a mensagem (m) da aplicação

Início da construção consistente:
  salvaCkpt(provisório)
  $\forall i$: se (VI[i] ≠ 0 *e* i ≠ pid)
      cont_req ← cont_req + 1
      req.ind ← VI[i]
      envia requisição (req) para p_i
  $\exists i$: (VI[i] ≠ 0 *e* i ≠ pid)
    pai ← *nulo*
    bloqueado ← *verdadeiro*

Recepção da requisição (req) de p_k:
  se (bloqueado)
    conf.filho ← *falso*
    envia confirmação (conf) para p_k
  senão
    se (VI[pid] ≤ req.ind)
      conf.filho ← *verdadeiro*
      envia confirmação (conf) para p_k
      pai ← k
      salvaCkpt(provisório)
      $\forall i$: se (VI[i] ≠ 0 *e* i ≠ pid *e*
          i ≠ pai)
        cont_req ← cont_req + 1
        req.ind ← VI[i]
        envia requisição (req) para p_i
      se (cont_req = 0)
        envia resposta (res) para p_{pai}
      senão
        bloqueado ← *verdadeiro*
    senão
      conf.filho ← *falso*
      envia confirmação (conf) para p_k

Recepção da confirmação (conf) de p_k:
  se (conf.filho)
    num_filhos ← num_filhos + 1
    filhos[k] ← *verdadeiro*
  cont_req ← cont_req - 1
  se (cont_req = 0 *e* num_filhos = 0)
    se (pai = *nulo*) bloqueado ← *falso*
    senão envia resposta (res) para p_{pai}

Recepção da resposta (res) de p_k:
  num_filhos ← num_filhos - 1
  se (num_filhos = 0)
    se (pai = *nulo*)
      transformaCkpt()
      $\forall i$: se (filhos[i] *e* i ≠ pid)
          envia liberação (lib) para p_i
    senão
      envia resposta (res) para p_{pai}

Recepção da liberação (lib) de p_k:
  se (bloqueado)
    transformaCkpt()
    $\forall i$: se (filhos[i] *e* i ≠ pid)
        envia liberação (lib) para p_i

salvaCkpt(tipo)
  VI[pid] ← VI[pid] + 1
  armazena um checkpoint de acordo com tipo

transformaCkpt()
  transforma o *checkpoint* provisório
    em permanente
  bloqueado ← *falso*
  $\forall i$: VI[i] ← 0
  $\forall i$: filhos[i] ← *falso*

B.2 Prakash e Singhal

O protocolo proposto por Prakash e Singhal utiliza vetores de bits para rastrear as dependências entre *checkpoints* [28]. Este protocolo não garante a minimalidade no número de *checkpoints*.

Protocolo B.2 Prakash e Singhal 93 (declarações)

Variáveis do processo:
- vbits ≡ vetor[0 . . . $n-1$] de bits
- VP ≡ vetor[0 . . . $n-1$] de bits
- peso ≡ real
- bloqueado ≡ booleano
- pid ≡ inteiro

Tipos de mensagem:
- aplicação:
  - vbits ≡ vetor[0 . . . $n-1$] de bits
- requisição:
  - peso ≡ real
  - ipid ≡ inteiro
  - VP ≡ vetor[0 . . . $n-1$] de bits
- resposta:
  - peso ≡ real
- liberação

Protocolo B.2 Prakash e Singhal 93

Início:
  $\forall i$: vbits[i] ← 0
  vbits[pid] ← 1
  peso ← 0
  bloqueado ← *falso*
  salvaCkpt(permanente)

Envio da mensagem (m) da aplic. para p_k:
  se (*não* bloqueado)
    m.vbits ← vbits
    envia mensagem da aplicação (m)

Recepção da mensagem (m) da aplic. de p_k:
  se (*não* bloqueado)
    vbits ← vbits *ou* m.vbits
    processa a mensagem da aplicação (m)

Início da construção consistente:
  bloqueado ← *verdadeiro*
  peso ← 1
  salvaCkpt(provisório)
  $\forall i$: se (vbits[i] = 1 *e* $i \neq$ pid)
      peso ← peso / 2
      req.peso ← peso
      req.ipid ← pid
      req.VP ← vbits
      envia requisição (req) para p_i
  se (peso = 1)
    transformaCkpt()

Recepção da requisição (req) de p_k:
  se (*não* bloqueado)
    bloqueado ← *verdadeiro*
    peso ← req.peso
    salvaCkpt(provisório)
    VP ← vbits *ou* req.VP
    $\forall i$: se (vbits[i] = 1 *e*
        req.VP[i] = 0 *e* $i \neq$ pid)
        peso ← peso / 2
        req.peso ← peso
        req.ipid ← req.ipid
        req.VP ← VP
        envia requisição (req) para p_i
    res.peso ← peso
    envia resposta (res) para $p_{req.ipid}$
  senão
    res.peso ← req.peso
    envia resposta (res) para $p_{req.ipid}$

Recepção da resposta (res) de p_k:
  peso ← peso + res.peso
  vbits[k] ← 1
  se (peso = 1)
    transformaCkpt()
    $\forall i$: se (vbits[i] = 1 *e* $i \neq$ pid)
      envia liberação (lib) para p_i

Recepção da liberação (lib) de p_k:
  transformaCkpt()

salvaCkpt(tipo)
  armazena um checkpoint de acordo com tipo

transformaCkpt()
  transforma o *checkpoint* provisório
    em permanente
  $\forall i$: vbits[i] ← 0
  $\forall i$: VP[i] ← 0
  vbits[pid] ← 1
  peso ← 0
  bloqueado ← *falso*

B.3 Cao e Singhal - abordagem *broadcast*

Este protocolo proposto por Cao e Singhal foi o primeiro protocolo a utilizar a abordagem *broadcast* para definir o conjunto de processos participantes de uma construção consistente [8]. Porém, este protocolo não garante minimalidade no número de *checkpoints*.

Protocolo B.3 Cao e Singhal - abordagem *broadcast* (declarações)

Variáveis do processo:
- R ≡ vetor[0...$n-1$] de bits
- MR ≡ matriz[0...$n-1$][0...$n-1$] de bits
- VP ≡ vetor[0...$n-1$] de bits
- respostas ≡ matriz[0...$n-1$][0...$n-1$] de bits
- bloqueado ≡ booleano
- pid ≡ inteiro

Tipos de mensagem:
- aplicação
- bloqueio
- resposta ao bloqueio
- requisição
- resposta
- liberação

Protocolo B.3 Cao e Singhal - abordagem *broadcast*

Início:
$\forall i$: R[i] $\leftarrow$ 0
R[pid] $\leftarrow$ 1
$\forall i$: $\forall j$: MR[i][j] $\leftarrow$ 0
$\forall i$: VP[i] $\leftarrow$ 0
$\forall i$: respostas[i] $\leftarrow$ 0
bloqueado $\leftarrow$ *falso*
salvaCkpt(permanente)

Envio da mensagem da aplic. (m) para p_k:
se (*não* bloqueado)
envia mensagem da aplicação (m)

Recepção da mensagem da aplic. (m) de p_k:
se (*não* bloqueado)
    R[k] $\leftarrow$ 1
    processa a mensagem da aplicação (m)

Início da construção consistente:
salvaCkpt(provisório)
MR[pid] $\leftarrow$ R
bloqueado $\leftarrow$ *verdadeiro*
envia broadcast do bloqueio (bloq)

Recepção do bloqueio (bloq) de p_k:
bloqueado $\leftarrow$ *verdadeiro*
rb.R $\leftarrow$ R
envia resposta ao bloqueio (rb) para p_k

Recepção da resp. ao bloqueio (rb) de p_k:
aux $\equiv$ vetor[0...$n-1$] de bits
MR[k] $\leftarrow$ rb.R
VP[k] $\leftarrow$ 1
se ($\forall l$: VP[l] = 1)
    VP $\leftarrow$ MR[pid]
faça
    aux $\leftarrow$ VP
    VP $\leftarrow$ VP $*$ MR
enquanto (VP $\neq$ aux)
se ($\forall i \neq$ pid: VP[i] = 0)
    transformaCkpt()
senão
    $\forall i \neq$ pid: se (VP[i] = 1)
        envia requisição (req) para p_i
    senão
        envia liberação (lib) para p_i

Recepção da requisição (req) de p_k:
salvaCkpt(provisório)
envia resposta (res) para p_k

Recepção da resposta (res) de p_k:
respostas[k] $\leftarrow$ 1
se (VP = respostas)
    transformaCkpt()
    $\forall i \neq$ pid: se (VP[i] = 1)
        envia liberação (lib) para p_i

    $\forall i$: VP[i] $\leftarrow$ 0, respostas[i] $\leftarrow$ 0
    $\forall i$: $\forall j$: MR[i][j] $\leftarrow$ 0

Recepção da liberação (lib) de p_k:
transformaCkpt()

salvaCkpt(tipo)
armazena um checkpoint de acordo com tipo

transformaCkpt()
transforma o *checkpoint* provisório
    em permanente
bloqueado $\leftarrow$ *falso*

B.4 VD-minimal na presença de um único iniciador

O protocolo VD-minimal utiliza vetores de dependências para rastrear as dependências entre *checkpoints* e garantir um número minimal de *checkpoints* a cada construção consistente [35]. Apresentamos aqui uma versão simplificada do protocolo que permite apenas um iniciador a cada instante de tempo.

Protocolo B.4 VD-minimal (declarações)

Variáveis do processo:
VD ≡ vetor[0 . . . n − 1] de inteiros
perm_VD ≡ vetor[0 . . . n − 1] de inteiros
VP ≡ vetor[0 . . . n − 1] de inteiros
respostas ≡ vetor[0 . . . n − 1] de inteiros
bloqueado ≡ booleano
pid ≡ inteiro

Tipos de mensagem:
aplicação:
VD ≡ vetor[0 . . . n − 1] de inteiros
requisição:
ipid ≡ inteiro
iind ≡ inteiro
VP ≡ vetor[0 . . . n − 1] de inteiros
resposta:
VP ≡ vetor[0 . . . n − 1] de inteiros
liberação

Protocolo B.4 VD-minimal

Início:
  $\forall i$: VD[i] ← 0
  $\forall i$: perm_VD[i] ← 0
  $\forall i$: respostas[i] ← 0
  bloqueado ← *falso*
  salvaCkpt(permanente)

Envio da mensagem da aplic. (m) para p_k:
  se (*não* bloqueado)
    m.VD ← VD
    envia mensagem da aplicação (m)

Recepção da mensagem da aplic. (m) de p_k:
  se (*não* bloqueado)
    VD ← max(m.VD, VD)
    processa a mensagem da aplicação (m)

Início da construção consistente:
  bloqueado ← *verdadeiro*
  salvaCkpt(provisório)
  verificaVP()
  $\forall i \neq$ pid: se (VP[i] ≠ 0)
      req.ipid ← pid
      req.iind ← VD[pid]
      req.VP ← VP
      envia requisição (req) para p_i
  se ($\forall i \neq$ pid: VP[i] = 0)
    transformaCkpt()

Recepção da requisição (req) de p_k:
  se (*não* bloqueado)
    se (req.VP[pid] = VD[pid])
      bloqueado ← *verdadeiro*
      VD[req.ipid] ← req.iind
      salvaCkpt(provisório)
      verificaVP()
      VP ← max(VP, req.VP)
      $\forall i$: se (VP[i] ≠ 0 *e* req.VP[i] = 0)
          p_req.ipid ← req.ipid
          p_req.iind ← req.iind
          p_req.VP ← VP
          envia requisição (p_req) para p_i
      res.VP ← VP
      envia resposta (res) para $p_{req.ipid}$
    senão
      se (req.iind > VD[req.ipid])
        VD[req.ipid] ← req.iind
        res.VP ← req.VP
        envia resposta (res) para $p_{req.ipid}$

Recepção da resposta (res) de p_k:
  respostas[k] ← res.VP[k]
  VP ← max(VP, res.VP)
  se ($\forall i \neq$ pid: respostas[i] ≥ VP[i])
    transformaCkpt()
    $\forall i$: se (VP[i] ≠ 0 *e* $i \neq$ pid)
        envia liberação (lib) para p_i
    $\forall i$: respostas[i] ← 0

Recepção da liberação (lib) de p_k:
  transformaCkpt()

salvaCkpt(tipo)
  armazena um checkpoint de acordo com tipo
  VD[pid] ← VD[pid] + 1

transformaCkpt()
  transforma o *checkpoint* provisório
    em permanente
  perm_VD ← VD
  bloqueado ← *falso*

verificaVP()
  $\forall i$: se (VD[i] > perm_VD[i])
      VP[i] ← VD[i]
    senão
      VP[i] ← 0

Apêndice C

Protocolos Não-Bloqueantes

Nesta seção, apresentamos os protocolos síncronos não-bloqueantes, que reduzem o número de *checkpoints* a cada construção consistente, analisados durante o desenvolvimento deste trabalho.

C.1 Prakash e Singhal 96

O protocolo proposto por Prakash e Singhal considera canais confiáveis FIFO (*First-In First-Out*) e foi o primeiro protocolo que une as características de minimalidade e não-bloqueio [30]. Porém, este protocolo pode gerar inconsistências [9].

Protocolo C.1 Prakash e Singhal 96 (declarações)

Variáveis do processo:
- R ≡ vetor[0 . . . n − 1] de bits
- VP ≡ vetor[0 . . . n − 1] de bits
- partic_cgc ≡ vetor[0 . . . n − 1] de bits
- colaboradores ≡ vetor[0 . . . n − 1] de bits
- env_msg ≡ vetor[0 . . . n − 1] de booleanos
- rec_msg ≡ booleano
- VI ≡ vetor[0 . . . n − 1] de inteiros
- peso ≡ real
- pid ≡ inteiro
- inic { pid ≡ inteiro
  ind ≡ inteiro }

Tipos de mensagem:
- aplicação:
  - ind ≡ inteiro
  - R ≡ vetor[0 . . . n − 1] de bits
  - inic { pid ≡ inteiro
    ind ≡ inteiro }
- requisição:
  - peso ≡ real
  - VP ≡ vetor[0 . . . n − 1] de bits
  - ind ≡ inteiro
  - inic { pid ≡ inteiro
    ind ≡ inteiro }
- resposta:
  - peso ≡ real
- liberação

Protocolo C.1 Prakash e Singhal 96

Início:
∀i: R[i] ← 0, VP[i] ← 0
partic_cgc[i] ← 0, colaboradores[i] ← 0
env_msg[i] ← *falso*, VI[i] ← 1
R[pid] ← 1
rec_msg ← *falso*
peso ← 0
armazena o *checkpoint*

Envio da mensagem (m) da aplic. para p_k
m.ind ← VI[pid]
m.R ← R
se (*não* env_msg[k])
env_msg[k] ← *verdadeiro*
m.inic ← inic
senão
m.inic ← NULL
envia mensagem da aplicação (m)

Recepção da mensagem (m) da aplica. de p_k:
se (m.ind ≤ ind[k])
processa (m) da aplicação
senão
ind[k] ← m.ind
se (m.inic.pid = inic.pid)
se (m.inic.ind = inic.ind)
processa (m) da aplicação
senão
propaga_ckpt(m.R, m.inic, 0)
processa (m) da aplicação
rec_msg ← *verdadeiro*
senão
se (*não* rec_msg)
propaga_ckpt(m.R, m.inic, 0)
processa (m) da aplicação
rec_msg ← *verdadeiro*
senão
processa (m) da aplicação

Início da construção consistente:
∀i: env_msg[i] ← *falso*
rec_msg ← *falso*
salvaCkpt(provisório)
peso ← 1
inic.pid ← pid
inic.ind ← VI[pid]
∀i: se (R[i] = 1 *e* $i \neq$ pid)
peso ← peso / 2
req.peso ← peso
req.VP ← R
req.ind ← VI[pid]
req.inic ← inic
envia requisição (req) para p_i
∀i: R[i] ← 0
R[pid] ← 1

Recepção da requisição (req) de p_k**:**
peso ← req.peso
se (m.inic = inic)
∀i: se (colaboradores[i] = 1 *e* $i \neq$ pid)
peso ← peso / 2
req.peso ← peso
envia requisição (req) para p_i
∀i: colaboradores [i] ← 0
res.peso ← peso
envia resposta (res) para $p_{req.inic.pid}$
senão
VI[k] ← req.ind
propaga_ckpt(req.R, req.inic, peso)
∀i: colaboradores [i] ← 0

Recepção da resposta (res) de p_k**:**
peso ← peso + res.peso
partic_cgc[k] ← 1
se (peso = 1)
transformaCkpt()
∀i: se (partic_cgc[i] = 1)
envia liberação (lib) para p_i
∀i: partic_cgc[i] ← 0

Recepção da liberação (lib) de p_k**:**
transformaCkpt()

Protocolo C.1 Prakash e Singhal 96 (continuação)

```
propaga_ckpt(mR, minic, pesoP)
  salvaCkpt(provisório)
  ∀i: env_msg[i] ← falso
  rec_msg ← falso
  inic ← minic
  ∀i: se (R[i] = 1 e mR[i] = 0 e i ≠ pid)
          colaboradores[i] ← 1
  VP ← R ou mR
  ∀i: se (colaboradores[i] = 1 e i ≠ pid)
          pesoP ← pesoP / 2
          req.peso ← pesoP
          req.VP ← VP
          req.ind ← VI[pid]
          req.inic ← inic
          envia requisição (req) para p_i
  ∀i: R[i] ← 0
  R[pid] ← 1
  res.peso ← pesoP
  envia resposta (res) para p_{inic.pid}

salvaCkpt(tipo)
  armazena um checkpoint de acordo com tipo
  VI[pid] ← VI[pid] + 1

transformaCkpt()
  transforma o checkpoint provisório
      em permanente
  ∀i: VP[i] ← 0
```

C.2 Cao e Singhal 98

O protocolo proposto por Cao e Singhal corrige o problema encontrado no protocolo de Prakash e Singhal descrito na seção anterior [9].

Protocolo C.2 Cao e Singhal 98 (declarações)

Variáveis do processo:
- R ≡ vetor[0 . . . n − 1] de bits
- VP ≡ vetor[0 . . . n − 1] de bits
- partic_cgc ≡ vetor[0 . . . n − 1] de bits
- colaboradores ≡ vetor[0 . . . n − 1] de bits
- env_msg ≡ vetor[0 . . . n − 1] de booleanos
- rec_msg ≡ booleano
- VI ≡ vetor[0 . . . n − 1] de inteiros
- peso ≡ real
- pid ≡ inteiro
- inic { pid ≡ inteiro
  ind ≡ inteiro }
- cp ≡ vetor[0 . . . n − 1] de lista do tipo ckpt
- ckpt { ckpttemp ≡ info do ckpt temporário
  R ≡ vetor[0 . . . n − 1] de bits
  cinic ≡ inic
  env_msg ≡ vetor[0 . . . n − 1] de booleanos }

Tipos de mensagem:
- aplicação:
  - ind ≡ inteiro
  - R ≡ vetor[0 . . . n − 1] de bits
  - inic { pid ≡ inteiro
    ind ≡ inteiro }
- requisição:
  - peso ≡ real
  - VP ≡ vetor[0 . . . n − 1] de bits
  - ind ≡ inteiro
  - inic { pid ≡ inteiro
    ind ≡ inteiro }
- resposta:
  - peso ≡ real
- liberação:
  - inic { pid ≡ inteiro
    ind ≡ inteiro }

Protocolo C.2 Cao e Singhal 98

Início:
$\forall i$: R[i] ← 0
VP[i] ← 0
partic_cgc[i] ← 0
colaboradores[i] ← 0
env_msg[i] ← *falso*
VI[i] ← 0
R[pid] ← 1
inic.pid ← pid
inic.ind ← 0
cp ← *nulo*
rec_msg ← *falso*
peso ← 0
armazena o checkpoint inicial

Envio da mensagem (m) da aplic. para p_k
se (*não* env_msg[k])
env_msg[k] ← *verdadeiro*
m.inic ← inic
senão
m.inic ← *nulo*
m.ind ← VI[pid]
m.R ← R
envia mensagem da aplicação (m)

Recepção da mensagem (m) da aplic. de p_k:
se (m.ind ≤ VI[k])
processa a mensagem da aplicação (m)
senão
VI[k] ← m.ind
se (m.inic = inic)
processa a mensagem da aplicação (m)
senão
se ($\forall i$: env_msg[i] = *falso*)
se (cp ≠ *nulo*)
cp[ultimo].cinic ←
cp[ultimo].cinic ∪ m.inic
senão
armazena *checkpoint* em
cp[ultimo+1].ckptemp
cp[ultimo+1].cinic ← m.inic
cp[ultimo+1].R ← m.R
$\forall i$: R[i] ← 0
env_msg[i] ← *falso*
inic ← m.inic
VI[pid] ← VI[pid] + 1
processa a mensagem da aplicação (m)

Início da construção consistente:
$\forall i$: partic_cgc[i] ← 0
armazena o *checkpoint*
VI[pid] ← VI[pid] + 1
peso ← 1
inic.pid ← pid
inic.ind ← VI[pid]
propaga_ckpt(*nulo*, inic, peso)

Recepção da requisição (req) de p_k:
peso ← req.peso
VI[k] ← req.ind
se (m.inic = inic)
se ($\exists i$: req.inic ∈ cp[i].cinic)
armazena o *checkpoint* cp[i].ckpttemp
propaga_ckpt(req.R, inic, peso)
senão
res.peso ← peso
envia resposta (res) para $p_{req.inic.pid}$
senão
se ($\forall i$: env_msg[i] = *falso*)
res.peso ← peso
envia resposta (res) para $p_{req.inic.pid}$
senão
armazena o *checkpoint*
VI[pid] ← VI[pid] + 1
inic ← req.inic
propaga_ckpt(req.R, inic, peso)

Protocolo C.2 Cao e Singhal 98 (continuação)

Recepção da resposta (res) de p_k:
peso ← peso + res.peso
partic_cgc[k] ← 1
se (peso = 1)
  $\forall i$: se (partic_cgc[i] = 1)
    lib.inic ← inic
    envia liberação (lib) para p_i
  $\forall i$: partic_cgc[i] ← 0
    VP[i] ← 0
    colaboradores[i] ← 0
peso ← 0

Recepção da liberação (lib) de p_k:
se ($\exists i$: lib.inic ∈ cp[i].cinic)
  $\forall j, i < j \leq$ ultimo: R ← R ∪ cp[j].R
    env_msg ← env_msg ∪ cp[j].env_msg
  cp ← *nulo*
$\forall i$: VP[i] ← 0
  colaboradores[i] ← 0
peso ← 0

propaga_ckpt(mR, minic, pesoP)
se ($\exists i$: minic ∈ cp[i].cinic)
  VP ← cp[i].R
  $\forall j < i$: VP[j] ←
    VP[j] ∪ cp[j].R
senão
  VP ← R
  $\forall j <$ ultimo: VP[j] ←
    VP[j] ∪ cp[j].R
$\forall i$: se (participante[i] = 1 *e* m.R[i] = 0 *e* $i \neq$ pid)
    colaboradores[i] ← 1
VP ← VP *ou* m.R
$\forall i$: se (colaboradores[i] = 1 *e* $i \neq$ pid)
    pesoP ← pesoP / 2
    req.peso ← pesoP
    req.VP ← VP
    req.ind ← VI[pid]
    req.inic ← minic
    envia requisição (req) para p_i
res.peso ← pesoP
envia resposta (res) para $p_{minic.pid}$
$\forall i$: R[i] ← 0
  env_msg[i] ← *falso*
R[pid] ← 1

C.3 RDT-Partner-NBS

O protocolo RDT-Partner-NBS é um protocolo síncrono não-bloqueante baseado no protocolo quase-síncrono RDT-Partner.

Protocolo C.3 RDT-Partner-NBS (declarações)

Variáveis do processo:
- VD $\equiv$ vetor$[0 \ldots n-1]$ de inteiros
- simples $\equiv$ vetor$[0 \ldots n-1]$ de booleano
- parceiro $\equiv$ inteiro
- VP $\equiv$ vetor$[0 \ldots n-1]$ de inteiros
- respostas $\equiv$ vetor$[0 \ldots n-1]$ de inteiros
- pid $\equiv$ inteiro

Tipos de Mensagem:
- aplicação:
  - VD $\equiv$ vetor$[0 \ldots n-1]$ de inteiros
  - simplesRec $\equiv$ vetor$[0 \ldots n-1]$ de booleano
- requisição:
  - ipid $\equiv$ inteiro
  - ind $\equiv$ inteiro
- resposta:
  - ind $\equiv$ inteiro
- liberação:
  - ind $\equiv$ inteiro

Protocolo C.3 RDT-Partner-NBS

Início:
  $\forall i$: VD[i] ← 0
  salvaCkpt (permanente)

Envio da mensagen da aplic. (m) para p_k
  se (parceiro = -1)
    parceiro ← k
  senão se (parceiro ≠ k)
    parceiro ← -2
  m.VD ← VD
  m.simplesRec ← simples[k]
  envia a mensagem (m) da aplicação

Recepção da mensagem da aplic. (m) de p_k:
  se (m.VD[k] > VD[k])
    se (parceiro ≠ -2 *e* (parceiro ≠ k *ou*
      (parceiro = k *e* m.VD[pid] = VD[pid] *e*
      não m.simplesRec)))
    salvaCheckpoint(mutável)
    simples[k] ← *verdadeiro*
  $\forall i$: VD[i] ← max(VD[i], m.VD[i])
  processa a mensagem da aplicação (m)

Início da construção consistente:
  c ← salvaCkpt (provisório)
  verificaVP(c)
  se ($\exists i \neq$ pid: VP[i] > 0)
    $\forall i$: se (VP[i] ≠ 0)
        req.ipid ← pid
        req.ind ← VP[i]
        envia requisição (req) para p_i
    $\forall i$: respostas[i] ← 0
  senão
    transformaCkpt(c, permanente)

Recepção da requisição (req) de p_{ini}:
  se (req.ind ≥ VD[pid])
    res.ind ← VD[pid]
    salvaCkpt (provisório)
  senão
    c ← procuraCkpt(req.ind)
    se (c é mutável)
      transformaCkpt(c, provisório)
    res.ind ← c.VD[pid]
  envia resposta (res) para p_{ini}

Recepção da resposta (res) de p_k:
  c ← procuraCkpt(VP[pid])
  se (c *não* é *nulo e* c é provisório)
    respostas[k] ← res.ind
    se ($\nexists i \neq$ pid: VP[i] ≠ 0 *e*
      VP[i] > respostas[i])
      $\forall i \neq$ pid: se (VP[i] ≠ 0)
          lib.ind ← respostas[i]
          envia liberação (lib) para p_i
      transformaCkpt(c, permanente)

Recepção da liberação (lib) de p_k:
  c ← procuraCkpt(lib.ind)
  transformaCkpt(c, permanente)

salvaCkpt(tipo)
  salva um checkpoint de acordo com o tipo
  VD[pid] ← VD[pid] + 1
  $\forall i$: simples[i] ← *falso*
  simples[pid] ← *verdadeiro*
  parceiro ← -1

verificaVP(c)
  p ← primeiro checkpoint permanente
  $\forall i$: se (c.VD[i] > p.VD[i])
      VP[i] ← VD[i]
    senão
      VP[i] ← 0

transformaCkpt(c, tipo)
  transforma c de acordo com tipo
  se (tipo = permanente)
    remove os ckpts salvos antes de c

procuraCkpt(ind)
  retorna o primeiro checkpoint c com
    c.VD[pid] ≥ ind

C.4 RDT-Minimal-NBS

O protocolo RDT-Minimal-NBS é um protocolo síncrono não-bloqueante baseado no protocolo quase-síncrono RDT-Minimal.

Protocolo C.4 RDT-Minimal-NBS (declarações)

Variáveis do processo:
- VD ≡ vetor[0 . . . $n-1$] de inteiros
- simples ≡ vetor[0 . . . $n-1$] de booleano
- igual ≡ vetor[0 . . . $n-1$] de booleano
- enviou ≡ vetor[0 . . . $n-1$] de booleano
- fase ≡ inteiro
- VP ≡ vetor[0 . . . $n-1$] de inteiros
- respostas ≡ vetor[0 . . . $n-1$] de inteiros
- pid ≡ inteiro

Tipos de Mensagem:
- aplicação:
  - VD ≡ vetor[0 . . . $n-1$] de inteiros
  - simples ≡ vetor[0 . . . $n-1$] de booleano
  - igual ≡ vetor[0 . . . $n-1$] de booleano
- requisição:
  - ipid ≡ inteiro
  - ind ≡ inteiro
- resposta:
  - ind ≡ inteiro
- liberação:
  - ind ≡ inteiro

Protocolo C.4 RDT-Minimal-NBS

Início:
$\forall i$: VD[i] $\leftarrow$ 0
salvaCkpt (permanente)

Envio da mensagen da aplic. (m) para p_k
enviou[k] $\leftarrow$ *verdadeiro*
se (fase = 0) fase $\leftarrow$ 1
m.VD $\leftarrow$ VD
m.simples $\leftarrow$ simples
m.igual $\leftarrow$ igual
envia a mensagem (m) da aplicação

Recepção da mensagem da aplic. (m) de p_k:
se (m.VD[k] > VD[k])
se (testaForçado(m))
salvaCheckpoint(mutável)
$\forall i$: se (m.VD[i] > VD[i])
VD[i] $\leftarrow$ m.VD[i]
simples[i] $\leftarrow$ m.simples[i]
senão se (m.VD[i] = VD[i])
simples[i] $\leftarrow$ simples[i] *ou* m.simples[i]
se (m.VD[k] = VD[k])
$\forall i$: igual[i] $\leftarrow$ igual[i] *ou* m.igual[i]
fase $\leftarrow$ 2
processa a mensagem da aplicação (m)

Início da construção consistente:
c $\leftarrow$ salvaCkpt (provisório)
verificaVP(c)
se ($\exists i \neq$ pid: VP[i] > 0)
$\forall i$: se (VP[i] $\neq$ 0)
req.ipid $\leftarrow$ pid
req.ind $\leftarrow$ VP[i]
envia requisição (req) para p_i
$\forall i$: respostas[i] $\leftarrow$ 0
senão
transformaCkpt(c, permanente)

Recepção da requisição (req) de p_{ini}:
se (req.ind $\geq$ VD[pid])
res.ind $\leftarrow$ VD[pid]
salvaCkpt (provisório)
senão
$c \leftarrow$ procuraCkpt(req.ind)
se (c é mutável)
transformaCkpt(c, provisório)
res.ind $\leftarrow$ c.VD[pid]
envia resposta (res) para p_{ini}

Recepção da resposta (res) de p_k:
c $\leftarrow$ procuraCkpt(VP[pid])
se (c *não é nulo e* c é provisório)
respostas[k] $\leftarrow$ res.ind
se ($\nexists i \neq$ pid: VP[i] $\neq$ 0 *e*
VP[i] > respostas[i])
$\forall i \neq$ pid: se (VP[i] $\neq$ 0)
lib.ind $\leftarrow$ respostas[i]
envia liberação (lib) para p_i
transformaCkpt(c, permanente)

Recepção da liberação (lib) de p_k:
$c \leftarrow$ procuraCkpt(lib.ind)
transformaCkpt(c, permanente)

salvaCkpt(tipo)
salva um checkpoint de acordo com o tipo
VD[pid] $\leftarrow$ VD[pid] + 1
$\forall i$: simples[i] $\leftarrow$ *falso*
$\forall i$: igual[i] $\leftarrow$ *falso*
$\forall i$: enviou[i] $\leftarrow$ *falso*
simples[pid] $\leftarrow$ *verdadeiro*
igual[pid] $\leftarrow$ *verdadeiro*
fase $\leftarrow$ 0

testaForçado(mensagem m)
se (fase = 0) retorna *falso*
se (fase = 2 *ou* m.VD[pid] = VD[pid] *e não*
m.simples[pid])
retorna *verdadeiro*
$\forall i$: se (enviou[i] *e não* m.igual[i])
retorna *falso*

verificaVP(c)
p $\leftarrow$ primeiro checkpoint permanente
$\forall i$: se (c.VD[i] > p.VD[i])
VP[i] $\leftarrow$ VD[i]
senão
VP[i] $\leftarrow$ 0

transformaCkpt(c, tipo)
transforma c de acordo com tipo
se (tipo = permanente)
remove os ckpts salvos antes de c

procuraCkpt(ind)
retorna o primeiro checkpoint c com
c.VD[pid] $\geq$ ind

Bibliografia

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

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

[3] R. Baldoni, F. Quaglia, and B. Ciciani. A VP-accordant Checkpointing Protocol Preventing Useless Checkpoints. In *17th IEEE Symposium on Reliable Distributed Systems*, pages 61–67, Indiana, USA, October 1998.

[4] Roberto Baldoni, Jean Michel Helary, Achour Mostefaoui, and Michel Raynal. A Communication-Induced Checkpoint Protocol that Ensures Rollback Dependency Trackability. In *IEEE Symposium on Fault Tolerant Computing*, pages 68–77, June 1997.

[5] B. Bhargava and S. Lian. Independent Checkpointing and Concurrent Rollback-Recovery – An Optimistic Approach. In *7th IEEE Symposium on Reliable Distributed Systems*, pages 3–12, 1988.

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

[7] G. Cao and M. Singhal. Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing Systems. *IEEE Transaction on Parallel and Distributed Systems*, 12(2):157–172, February 2001.

[8] G. Cao and M. Singhal. Checkpointing with Mutable Checkpoints. *Theoretical Computer Science*, 290(2):1127–1148, jan 2003.

[9] Guohong Cao and Mukesh Singhal. On Coordinated Checkpointing in Distributed Systems. *IEEE Transaction on Parallel and Distributed Systems*, 9(12):1213–1225, December 1998.

[10] Guohong Cao and Mukesh Singhal. On the Impossibility of Min-process Non-blocking Checkpointing and an Efficient Checkpointing Algorithm for Mobile Computing Systems. In *Proc. 27th International Conference on Parallel Processing*, pages 37–44, New York, 1998. IEEE Press.

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

[12] E. W. Dijkstra and C. S. Scholten. Termination Detection for diffusing computations. *Information Processing Letters*, 11(4):1–4, August 1980.

[13] E. N. Elnozahy and D. B. Johnson ad W. Zwaenepoel. The Performance of Consistent Checkpointing. In *11th IEEE Symposium on Reliable Distributed Systems*, pages 86–95, October 1992.

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

[15] Islene C. Garcia, Gustavo M. D. Vieira, and Luiz 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.

[16] Islene Calciolari Garcia and Luiz Eduardo Buzato. Progressive Construction of Consistent Global Checkpoints. In *19th IEEE International Conference on Distributed Computing Systems*, Austin, Texas, EUA, June 1999.

[17] Islene Calciolari Garcia and Luiz Eduardo Buzato. Using Common Knowledge to Improve Fixed-Dependency-After-Send. In *Segundo Workshop de Testes e Tolerância a Falhas*, pages 16–21, July 2000.

[18] Islene Calciolari Garcia and Luiz Eduardo Buzato. An Efficient Checkpointing Protocol for the Minimal Characterization of Operational Rollback-Dependency Trackability. In *23rd IEEE Symposium on Reliable Distributed Computing Systems*, pages 126–135, Florianópolis, Santa Catarina, October 2004.

[19] S. T. Huang. Detecting Termination of Distributed Computations by External Agents. In *9th International Conference on Distributed Computing Systems*, pages 79–84, June 1989.

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

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

[22] P. J. Leu and B. Bhargava. Concurrent Robust Checkpointing and Recovery in Distributed Systems. In *4th IEEE International Conference on Data Engineering*, pages 154–163, 1988.

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

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

[25] D. Manivannan and M. Singhal. Quasi-Synchronous Checkpointing: Models, Characterization, and Classification. In *IEEE Transaction on Parallel and Distributed Systems*, volume 10, pages 703–713, July 1999.

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

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

[28] R. Prakash and M. Singhal. Minimal Global Snapshot and Failure Recovery using Infection. Technical Report OSU-CISRC-12/93-TR42, Department of Computer Science, The Ohio State University, 1993.

[29] R. Prakash and M. Singhal. Maximal Global Snapshot with Concurrent Initiators. In *6th IEEE Symposium on Parallel and Distributed Processing*, pages 344–351, oct 1994.

[30] R. Prakash and M. Singhal. Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems. *IEEE Transaction on Parallel and Distributed Systems*, 7(10):1035–1048, October 1996.

[31] F. Quaglia, R. Baldoni, and B. Ciciani. On the No-Z-Cycle Property of Distributed Executions. *Journal of Computer and System Sciences*, 61(3):400–427, December 2000.

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

[33] T. C. Sakata and I. C. Garcia. Non-Blocking Synchronous Checkpointing Based on Rollback-Dependency Trackability. In *25th IEEE Symposium on Reliable Distributed Computing Systems*, pages 411–420, Leeds, UK, October 2006.

[34] T. C. Sakata, I. C. Garcia, and L. E. Buzato. BCS-SNB: Um Protocolo de Checkpointing Síncrono Não Bloqueante. In *II Workshop de Teses e Dissertações em Computação Tolerante a Falhas*, pages 73–78, São Paulo, Brazil, October 2003.

[35] T. C. Sakata, I. C. Garcia, and L. E. Buzato. Checkpointing Síncrono Bloqueante Minimal com Iniciadores Concorrentes. In *Simpósio Brasileiro de Redes de Computadores*, pages 681–696, Natal, Rio Grande do Norte, Brazil, May 2003.

[36] T. C. Sakata, I. C. Garcia, and L. E. Buzato. Uso de Broadcast na Sincronização de Checkpoints em Protocolos Minimais. In *V Workshop de Testes e Tolerância a Falhas*, pages 153–164, Rio Grande do Sul, Brazil, May 2004.

[37] Rodrigo Schmidt, Islene C. Garcia, Fernando Pedone, and Luiz Eduardo Buzato. Optimal Asynchronous Garbage Collection for RDT Checkpointing Protocols. In *19th IEEE International Conference on Distributed Computing Systems*, pages 167–176, Ohio, EUA, June 2005.

[38] Rodrigo Malta Schmidt. Coleta de Lixo para Protocolos de Checkpointing. Master's thesis, Instituto de Computação–Universidade Estadual de Campinas, October 2003.

[39] L. M. Silva and J. G. Silva. Global Checkpointing for Distributed Programs. In *11th IEEE Symposium on Reliable Distributed Systems*, pages 155–162, October 1992.

[40] A. S. Tanenbaum and M. Steen. *Distributed Systems Principles and Paradigms*. Alan Apt, 2002.

[41] Jichiang Tsai, Sy-Yen Kuo, and Yi-Min Wang. Theoretical Analysis for Communication-Induced Checkpointing Protocols with Rollback-Dependency Trackability. *IEEE Transaction on Parallel and Distributed Systems*, 9(10):963–971, October 1998.

[42] Gustavo Maciel Dias Veira, Islene Calciolari Garcia, and Luiz Eduardo Buzato. Systematic Analysis of Index-Based Checkpointing Algorithms using Simulation. In *IX Simpósio de Computação Tolerante a Falhas*, pages 31–42, Florianópolis, Santa Catarina, March 2001.

[43] G. M. D. Vieira and L. E. Buzato. Distributed Checkpointing: Analysis and Benchmarks. In *Simpósio Brasileiro de Redes de Computadores*, Curitiba, Paraná, May 2006.

[44] Gustavo Maciel Dias Vieira. Estudo Comparativo de Algoritmos para Checkpointing. Master's thesis, Instituto de Computação–Universidade Estadual de Campinas, December 2001.

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

[46] Y. M. Wang and W. K. Fuchs. Lazy Checkpoint Coordination for Bounding Rollback Propagation. In *12th IEEE Symposium on Reliable Distributed Systems*, pages 78–85, October 1993.

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