Visões Progressivas de Computações Distribuídas

Islene Calciolari Garcia

Tese de Doutorado

Instituto de Computação
Universidade Estadual de Campinas

Visões Progressivas de Computações Distribuídas

Islene Calciolari Garcia*

18 de dezembro de 2001

Instituto de Computação
Universidade Estadual de Campinas

Visões Progressivas de Computações Distribuídas

Islene Calciolari Garcia*

18 de dezembro de 2001

Banca Examinadora:

- Prof. Dr. Luiz Eduardo Buzato (Orientador)
- Profª Drª Taisy Silva Weber
  Instituto de Informática—UFRGS
- Prof. Dr. Marcos José Santana
  Instituto de Ciências Matemáticas e de Computação—USP
- Prof. Dr. Edson Norberto Cáceres
  Departamento de Computação e Estatística—UFMS
- Prof. Dr. Ricardo de Oliveira Anido
  Instituto de Computação—UNICAMP
- Prof. Dr. Edmundo Roberto Mauro Madeira
  Instituto de Computação—UNICAMP

*Apoio financeiro da FAPESP, processo número 99/01293-2 para Islene Calciolari Garcia, 96/1532-9 para o Laboratório de Sistemas Distribuídos e 97/10982-0 para o Laboratório de Alto Desempenho. Durante o início do curso, a aluna recebeu apoio financeiro do CNPq, processo número 145563/98-7. Para a divulgação do trabalho, a aluna recebeu apoio financeiro do convênio PRONEX/Finep, processo número 76.97.1022.00 (projeto Sistemas Avançados de Informação).

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Garcia, Islene Calciolari

G165v Visões progressivas de computações distribuídas / Islene Calciolari Garcia -- Campinas, [S.P. :s.n.], 2001.

Orientador : Luiz Eduardo Buzato

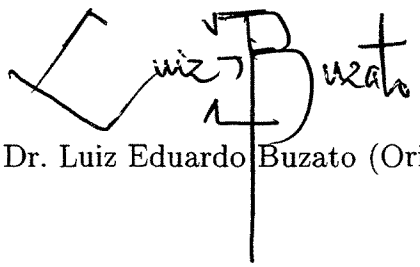
Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Algoritmos. 2. Tolerância a falhas. 3. Processameto distribuído. I. Buzato, Luiz Eduardo. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Visões Progressivas de Computações Distribuídas

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Islene Calciolari Garcia e aprovada pela Banca Examinadora.

Campinas, 7 de Janeiro de 2002.



Prof. Dr. Luiz Eduardo Buzato (Orientador)

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

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 18 de dezembro de 2001, pela Banca Examinadora composta pelos Professores Doutores:

Profa. Dra. Taisy Silva Weber
UFRGS

Prof. Dr. Marcos José Santana
USP

Prof. Dr. Edson Norberto Cáceres
UFMS

Prof. Dr. Ricardo de Oliveira Anido
UNICAMP

Prof. Dr. Edmundo Roberto Mauro Madeira
UNICAMP

Prof. Dr. Luiz Eduardo Buzato
UNICAMP

Agradecimentos

Ao Prof. Dr. Luiz Eduardo Buzato, por ter acompanhado de perto todos os meus passos nesta pesquisa, por ter dado credibilidade às minhas suposições muito antes de eu conseguir prová-las, pela orientação e pela amizade.

Ao Gustavo Maciel Dias Vieira, por ter me auxiliado na compreensão do comportamento dos protocolos quase-síncronos.

À Tiemi Christine Sakata e ao Rodrigo Malta Schmidt, por aceitarem temas de pesquisa que darão continuidade a este trabalho.

Aos professores do Instituto de Computação e do antigo Departamento de Ciência da Computação, por terem colaborado muito para a minha formação.

Ao Prof. Dr. Pedro Jussieu de Rezende, pela orientação durante a minha iniciação científica e durante o meu estágio de capacitação docente.

Aos meus ex-alunos de MC202 (estrutura de dados, primeiro semestre de 2000), por terem tornado especial a minha primeira experiência como docente.

Aos membros da banca examinadora pelas sugestões e pelo incentivo.

Aos funcionários do Instituto de Computação, em especial à Vera Lúcia de Oliveira Ragazzi e ao Daniel de Jesus Capeleto, pela eficiência e cordialidade.

À FAPESP e ao CNPq, pelo apoio financeiro.

À Profª Drª Claudia Maria Bauzer Medeiros por me conceder auxílio, dentro do projeto Sistemas Avançados de Informação, para a apresentação de dois artigos no exterior.

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

A todos os meus colegas de doutorado, principalmente Ana Maria, Christiane, Helena e Guilherme Albuquerque, pelas experiências compartilhadas.

A todos os meus amigos, por compreenderem a minha ausência nesses anos de pesquisa.

Aos meus sobrinhos Tainã, Maíra e Matheus e às crianças Ana Elisa, João Gabriel, Tamiris e Aline pela alegria.

Aos meus irmãos, Sérgio e Jônatas, às minhas cunhadas Sílvia e Carla, e minha avó Olívia pelo apoio e pelo carinho.

Ao meu pai e ao meu avô Vital (ambos *in memoriam*), por tudo o que ensinaram e deixaram para mim.

À minha mãe, pelo amor e pela dedicação constantes.

Ao meu noivo Alexandre, pelo amor e por não me deixar fugir dos desafios.

Resumo

Um *checkpoint* é um estado selecionado por um processo durante a sua execução. Um *checkpoint* global é composto por um *checkpoint* de cada processo e é consistente se representa uma fotografia da computação que poderia ter sido capturada por um observador externo. Soluções para vários problemas em sistemas distribuídos necessitam de uma seqüência de *checkpoints* globais consistentes que descreva o progresso de uma computação distribuída. Como primeira contribuição desta tese, apresentamos um conjunto de algoritmos para a construção destas seqüências, denominadas *visões progressivas*. Outras contribuições provaram que certas suposições feitas na literatura eram falsas utilizando o argumento de que algumas propriedades precisam ser válidas ao longo de todo o progresso da computação.

Durante algumas computações distribuídas, todas as dependências de retrocesso entre *checkpoints* podem ser rastreadas em tempo de execução. Esta propriedade é garantida através da indução de *checkpoints* imediatamente antes da formação de um padrão de mensagens que poderia dar origem a uma dependência de retrocesso não rastreável. Estudos teóricos e de simulação indicam que, na maioria das vezes, quanto mais restrito o padrão de mensagens, menor o número de *checkpoints* induzidos. Acreditava-se que a caracterização minimal para a obtenção desta propriedade estava estabelecida e que um protocolo baseado nesta caracterização precisaria da manutenção e propagação de informações de controle com complexidade $O(n^2)$, onde n é o número de processos na computação. A complexidade quadrática tornava o protocolo baseado na caracterização mimimal menos interessante que protocolos baseados em caracterizações maiores, mas com complexidade linear.

A segunda contribuição desta tese é uma prova de que a caracterização considerada minimal podia ser reduzida, embora a complexidade requerida por um protocolo baseado nesta nova caracterização minimal continuasse indicando ser quadrática. A terceira contribuição desta tese é a proposta de um pequeno relaxamento na caracterização minimal que propicia a implementação de um protocolo com complexidade linear e desempenho semelhante à solução quadrática. Como última contribuição, através de um estudo detalhado das variações da informação de controle durante o progresso de uma computação, propomos um protocolo que implementa exatamente a caracterização minimal, mas com complexidade linear.

Abstract

A checkpoint is a state selected by a process during its execution. A global checkpoint is composed of one checkpoint from each process and it is consistent if it represents a snapshot of the computation that could have been taken by an external observer. The solution to many problems in distributed systems requires a sequence of consistent global checkpoints that describes the progress of a distributed computation. As the first contribution of this thesis, we present a set of algorithms to the construction of these sequences, called *progressive views*. Additionally, the analysis of properties during the progress of a distributed computation allowed us to verify that some assumptions made in the literature were false.

Some checkpoint patterns present only on-line trackable rollback-dependencies among checkpoints. This property is enforced by taking a checkpoint immediately before the formation of a message pattern that can produce a non-trackable rollback-dependency. Theoretical and simulation studies have shown that, most often, the more restricted the pattern, the more efficient the protocol. The minimal characterization was supposed to be known and its implementation was supposed to require the processes of the computation to maintain and propagate $O(n^2)$ control information, where n is the number of processes in the computation. The quadratic complexity makes the protocol based on the minimal characterization less interesting than protocols based on wider characterizations, but with a linear complexity.

The second contribution of this thesis is a proof that the characterization that was supposed to be minimal could be reduced. However, the complexity required by a protocol based on the new minimal characterization seemed to be also quadratic. The third contribution of this thesis is a protocol based on a slightly weaker condition than the minimal characterization, but with linear complexity and performance similar to the quadratic solution. As the last contribution, through a detailed analysis of the control information computed and transmitted during the progress of distributed computations, we have proposed a protocol that implements exactly the minimal characterization, but with a linear complexity.

Conteúdo

Lista de Figuras

Lista de Classes

Lista de Abreviaturas

Abreviatura	Significado	Páginas
C-path	*Causal path*	12, 23
CC-path	*Causal-Causal path*	24, 69
CM-path	*Causal-Message path*	24, 69
EPCM-path	*Elementary-Prime-Causal-Message path*	26, 70
EPSCM-path	*Elementary-Prime-Simple-Causal-Message path*	26, 70
PCM-path	*Prime-Causal-Message path*	25, 87
PMM-path	*Prime-Message-Message path*	71, 89
MM-path	*Message-Message path*	70
Z-cycle	*Zigzag cycle*	13, 38
Z-path	*Zigzag path* (não causal)	12, 23
BCS	Protocolo proposto por Briattico, Ciuffoletti e Simoncini	20, 103
BQF	Protocolo proposto por Baldoni, Quaglia e Fornara	108
BHMR	Protocolo proposto por Baldoni, Helary, Mostefaoui e Raynal	27, 89
CAS	*Checkpoint After Send*	18
CASBR	*Checkpoint After Send Before Receive*	18
CBR	*Checkpoint Before Receive*	18
CCP	*Checkpoint and Communication Pattern*	65
FDAS	*Fixed Dependency After Send*	22, 59
FDI	*Fixed Dependency Interval*	19
PZCF	*Partially Z-Cycle Free*	21, 40
NRAS	*No Receive After Send*	18
RDT	*Rollback-dependency trackability*	21–30
SZPF	*Strictly Z-Path Free*	18
ZCF	*Z-Cycle Free*	20, 40
ZPF	*Z-Path Free*	19, 40

Capítulo 1

Introdução

Um *checkpoint* é um estado selecionado durante a execução de um processo. Um *checkpoint* global é composto por um *checkpoint* de cada processo e é consistente se representa uma fotografia da computação que poderia ter sido capturada por um observador externo. Soluções para vários problemas em sistemas distribuídos necessitam de uma seqüência de *checkpoints* globais consistentes que descreva o progresso de uma computação distribuída. Depuradores de computações distribuídas, por exemplo, podem interromper a computação após a validação de uma seqüência de predicados globais; outros exemplos incluem sistemas de visualização e reconfiguração de computações distribuídas.

Como primeira contribuição desta tese, apresentamos um conjunto de algoritmos para a construção destas seqüências, denominadas *visões progressivas* (Capítulo 3). Comentamos também como o conceito de visões progressivas pode ser útil para o desenvolvimento de protocolos para *checkpointing* e para a integração de mecanismos de *checkpointing*, recuperação por retrocesso de estado e monitorização (Capítulo 4). Outras contribuições foram baseadas na observação de que algumas propriedades precisam ser válidas ao longo de todo o progresso da computação. Em particular, provamos que algumas suposições feitas na literatura com relação à propriedade *Rollback-Dependency Trackability* (RDT) [6, 7, 50] não eram verdadeiras (Capítulos 5, 6 e 9).

Rollback-Dependency Trackability

Um padrão de *checkpoints* é o conjunto de todos os *checkpoints* selecionados durante a execução de uma computação distribuída. Em padrões RDT, todas as dependências de retrocesso entre *checkpoints* podem ser rastreadas em tempo de execução. Esta propriedade permite soluções simples para a determinação de *checkpoints* globais consistentes que incluem um grupo de *checkpoints*. Entre as aplicações que podem se beneficiar desta

propriedade podemos citar: recuperação por retrocesso, recuperação de erros de *software*, recuperação de *deadlocks*, computação móvel e depuração distribuída [50].

Um protocolo RDT permite que os processos da computação selecionem *checkpoints* livremente, mas induz um *checkpoint* forçado imediatamente antes da formação de um padrão de mensagens que possa dar origem a uma dependência de retrocesso não rastreável [7, 50]. Os protocolos mais simples que garantem RDT baseiam-se apenas em eventos de *checkpoint*, envio e recepção de mensagens [36, 50], mas são propensos a induzir um número muito alto de *checkpoints*. O protocolo *Checkpoint-Before-Receive* (CBR), por exemplo, induz um *checkpoint* imediatamente antes da recepção de cada uma das mensagens da computação.

É possível diminuir o número de *checkpoints* forçados através da manutenção e propagação de vetores de dependências, análogos, como mecanismo, a vetores de relógios [16, 37]. O protocolo *Fixed-Dendency-Interval* (FDI) economiza *checkpoints* em relação ao CBR por não induzir um *checkpoint* forçado quando um processo recebe uma mensagem cujo vetor de dependências não traz informações novas [31, 50]. O protocolo *Fixed-Dendency-After-Send* (FDAS) economiza *checkpoints* forçados em relação ao FDI por não induzir um *checkpoint* forçado antes do primeiro evento de envio em um intervalo de *checkpoints* [50].

Baldoni, Helary e Raynal fizeram um estudo detalhado da propriedade RDT [6, 7] de maneira a tentar reduzir ainda mais o número de *checkpoints* forçados. Eles trabalharam em um nível de abstração no qual dependências de retrocesso entre *checkpoints* são capturadas por seqüências de mensagens denominadas *zigzag paths* [38]. Existem dois tipos de *zigzag paths*: causais *(C-paths)* e não causais *(Z-paths)*. Em um padrão RDT, todas as *Z-paths* devem estar duplicadas causalmente, ou seja, todos os *checkpoints* conectados por uma *Z-path* devem estar também conectados por uma *C-path*. A abordagem proposta por Baldoni, Helary e Raynal consiste em tentar reduzir o conjunto de *Z-paths* que precisam ser duplicadas para que a propriedade RDT seja garantida. Eles propuseram uma hierarquia de *Z-paths* e concluíram que um determinado sub-conjunto, formado por *EPSCM-paths*, representava o conjunto minimal de *Z-paths* que precisariam ser duplicadas para que um padrão de *checkpoints* obedeça à propriedade RDT.

A formação de uma *EPSCM-path* pode ser evitada por um processo p_i se um *checkpoint* forçado for induzido em p_i imediatamente antes de sua formação. No entanto, uma *EPSCM-path* não precisa ser evitada se no momento de sua formação o processo p_i é capaz de concluir que esta *EPSCM-path* está duplicada visivelmente. Baseados nesta observação e no resultado de minimalidade comentado acima, Baldoni, Helary e Raynal conjeturaram que o conjunto formado por *EPSCM-paths* não duplicadas visivelmente determina o menor conjunto de *Z-paths* que devem ser evitadas em tempo de execução por

um protocolo RDT [6]. Eles também afirmaram que um protocolo que evita *EPSCM-paths* não duplicadas visivelmente precisaria da manutenção e propagação de informações de controle com complexidade $O(n^2)$, onde n é o número de processos na computação [7].

Tsai, Kuo e Wang provaram que a implementação de um protocolo RDT ótimo, ou seja, que induza o menor número de *checkpoints* forçados para qualquer computação distribuída, não é possível [45]. No entanto, estudos teóricos e de simulação indicam que, na maioria das vezes, quanto mais restritiva a condição para indução de *checkpoints*, menor o número de *checkpoints* induzidos [3, 7]. Apesar da diminuição no número de *checkpoints* forçados, a complexidade quadrática tornava o protocolo baseado na caracterização mimimal menos interessante que protocolos baseados em caracterizações maiores, mas com complexidade linear.

Contribuições

Na literatura, a condição para indução de *checkpoints* pelo protocolo FDAS baseia-se na comparação de *todas* as entradas do vetor de dependências recebido em uma mensagem e o vetor de dependências do processo receptor da mensagem [3, 17, 36, 50]. No Capítulo 5, provamos que esta condição pode ser simplificada e é necessário comparar apenas a entrada correspondente ao emissor da mensagem.

Esta pequena otimização indicou que era possível implementar um protocolo RDT testando apenas *PMM-paths*, um sub-conjunto de *EPSCM-paths*, que não aparecia entre as várias caracterizações propostas por Baldoni, Helary e Raynal. Esta foi a primeira evidência de que a conjetura proposta por eles poderia ser falsa. No Capítulo 6, provamos que o conjunto formado por *PMM-paths* não duplicadas visivelmente determina o menor conjunto de *Z-paths* cuja formação deve ser evitada por um protocolo que garante RDT em tempo de execução. O argumento desta prova estava relacionado ao fato da propriedade RDT precisar ser garantida ao longo de todos os instantes (cortes consistentes) da computação e não apenas em relação aos *checkpoints*.

Apesar de termos obtido uma redução na caracterização considerada minimal, a complexidade de um protocolo baseado na nova caracterização parecia continuar quadrática. No Capítulo 7, propomos um protocolo RDT, chamado RDT-Partner, baseado em um super-conjunto da caracterização minimal com as seguintes características: (i) complexidade linear e (ii) segundo dados de simulação, desempenho semelhante à solução quadrática.

O bom desempenho do RDT-partner está relacionado à observação de que a implementação de uma condição mais restritiva C_R pode não ser mais vantajosa que a implementação de uma condição menos restritiva C_r. Isto ocorre quando as situações em que a condição C_R economiza *checkpoints* em relação à condição C_r acontecem raramente. No

Capítulo 8, mostramos um exemplo desta situação no contexto de protocolos para *checkpointing* baseados em índices. Apresentamos dados de simulação do protocolo proposto por Briatico, Ciuffoletti e Simoncini (BCS) [11] e exploramos o impacto de algumas otimizações deste protocolo propostas na literatura [9, 28]. Os resultados encontrados indicam que uma otimização cara e complexa do protocolo BCS pode não reduzir o número de *checkpoints* forçados em comparação a uma otimização mais barata e simples. Os resultados apresentados nos Capítulos 7 e 8 foram elaborados em conjunto com Gustavo Maciel Dias Vieira. A ferramenta que utilizamos para colher os dados de simulação, chamada Metapromela, foi desenvolvida como parte de seu projeto de mestrado [47, 48].

No Capítulo 9, apresentamos um estudo detalhado das variações dos vetores de dependências durante o progresso de uma computação distribuída sob um protocolo RDT. Graças a esta análise, propomos um protocolo que implementa exatamente a caracterização minimal, mas com complexidade linear.

Estrutura da tese

Esta tese é formada por uma coletânea de artigos que, com exceção do último, foram publicados em eventos da área. A notação escolhida para cada artigo foi a que melhor descrevia o resultado e/ou a que facilitava a comparação com trabalhos relacionados. Desta forma, o leitor precisa ficar atento ao fato de a notação não ser uniforme ao longo do texto. Os algoritmos, bem como os outros protocolos apresentados nesta tese, são descritos em Java* [26, 44]. Escolhemos Java por ser uma linguagem de leitura simples e de descrição precisa.

Antes de apresentarmos os artigos, faremos uma breve introdução sobre *checkpointing* no Capítulo 2. Note que optamos por não traduzir o termo *checkpoint*, visto que a tradução freqüentemente encontrada na literatura, *ponto de recuperação*, não abrange este conceito em contextos mais amplos como monitorização e reconfiguração de computações distribuídas. Além disso, acreditamos que a manuntenção de termos como *checkpoint*, *zigzag path*, *rollback-dependency-trackability* e a padronização das várias siglas utilizadas facilita a leitura dos textos em inglês que virão a seguir.

O Capítulo 3 apresenta o artigo *Progressive Construction of Consistent Global Checkpoints*, publicado na IEEE 19th International Conference on Distributed Computing Systems, ocorrida em Austin, Texas, Estados Unidos, em junho de 1999. Este artigo introduz um significado intuitivo para *zigzag paths* e apresenta um conjunto de algoritmos para a construção de visões progressivas de computações distribuídas. Parte destes resultados foram obtidos previamente [19].

*Java is a trademark of Sun Microsystems, Inc.

O Capítulo 4 apresenta o resumo *Monitorização e Recuperação por Retrocesso Utilizando Visões Progressivas de Computações Distribuídas*, que foi publicado no VIII Simpósio de Computação Tolerante a Falhas, ocorrido em Campinas, São Paulo, em julho de 1999. Este resumo comenta como o conceito de visões progressivas pode ser útil para o desenvolvimento de protocolos para *checkpointing* mais eficientes e para a integração de mecanismos de *checkpointing*, recuperação por retrocesso de estado e monitorização.

No Capítulo 5, apresentamos o artigo *Using Common Knowledge to Improve Fixed-Dependency-After-Send*, que foi publicado no II Workshop de Testes e Tolerância a Falhas, ocorrido em Curitiba, Paraná, em julho de 2000. Este artigo prova que a condição para a indução de *checkpoints* no protocolo FDAS [50] pode ter sua complexidade reduzida de $O(n^2)$ para $O(n)$, onde n é o número de processos na computação.

No Capítulo 6, apresentamos o artigo *On the Minimal Characterization of the Rollback-Dependency Trackability Property*, publicado na 21th IEEE International Conference on Distributed Computing Systems, ocorrida em Phoenix, Arizona, Estados Unidos, em abril de 2001. Neste artigo, nós provamos que a conjetura a respeito da caracterização minimal para RDT proposta por Baldoni, Helary e Raynal [6] era falsa. Propusemos a caracterização minimal para RDT e uma abordagem original para análise de protocolos que obedecem a esta propriedade.

O Capítulo 7 apresenta o artigo *RDT-Partner: An Efficient Checkpointing Protocol that Enforces Rollback-Dependency Trackability* publicado no 19º Simpósio Brasileiro de Redes de Computadores, ocorrido em Florianópolis, Santa Catarina, em maio de 2001. Este artigo propõe um protocolo RDT com complexidade linear que economiza *checkpoints* forçados em relação ao protocolo FDAS [50]. Estudos teóricos e de simulação indicam que o RDT-Partner apresenta desempenho semelhante ao protocolo proposto por Baldoni, Helary, Mostefaoui e Raynal (BHMR) [3], que tem complexidade quadrática. Este trabalho foi feito em conjunto com Gustavo Maciel Dias Vieira.

No Capítulo 8, apresentamos o artigo *Systematic Analysis of Index-Based Checkpointing Algorithms using Simulation* publicado no IX Simpósio de Computação Tolerante a Falhas, ocorrido em Florianópolis, Santa Catarina, em março de 2001. Este artigo apresenta dados de simulação para comparação de protocolos para *checkpointing* baseados em índices. Este trabalho também foi feito em conjunto com Gustavo Maciel Dias Vieira.

O Capítulo 9 apresenta o artigo *A Linear Approach to Enforce the Minimal Characterization of the Rollback-Dependency Trackability Property*, enviado para avaliação em uma conferência da área. Neste artigo, propomos um protocolo RDT que implementa a caracterização minimal com complexidade linear, contrariando resultados anteriores que indicavam que um limite inferior quadrático para esta abordagem [7].

Finalmente, o Capítulo 10 encerra este texto, apresentando as nossas conclusões e a proposta de trabalhos futuros.

Capítulo 2

Checkpointing

O propósito deste Capítulo é fazer um resumo de conceitos, definições e resultados relacionados a *checkpointing* de maneira a facilitar a leitura dos artigos que compõem esta tese.

1 Modelo computacional

Uma computação distribuída assíncrona é formada por um conjunto finito de processos $\{p_0, \ldots, p_{n-1}\}$ que executam eventos de maneira estritamente seqüencial e que se comunicam exclusivamente através de troca de mensagens. Não existem mecanismos para compartilhamento de memória, acesso a relógio global, sincronização de relógios locais ou conhecimento a respeito das diferenças de velocidade entre os processadores. A comunicação é feita através de canais unidirecionais entre pares de processos e todos os processos conseguem se comunicar diretamente ou através de processos intermediários. Há garantia de entrega de mensagens, mas estas podem sofrer atrasos arbitrários e inclusive chegar aos seus destinos fora de ordem. Computações distribuídas são usualmente representadas através de diagramas espaço-tempo, onde linhas horizontais representam a execução dos processos e arestas representam mensagens (Figura 1).

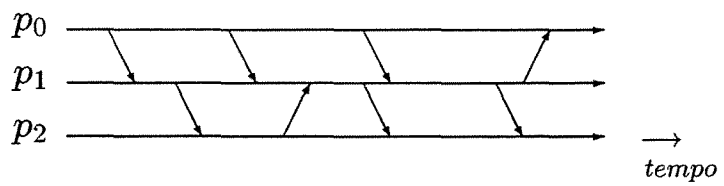


Figura 1: Diagrama espaço-tempo para uma computação distribuída

A execução de uma computação distribuída pode ser observada e controlada por um monitor. Neste caso, consideramos uma arquitetura de *software* para o programa distribuído completo que é composta pela superposição de dois sistemas reativos (Figura 2):

(i) o *programa da aplicação*, que implementa os aspectos funcionais da aplicação, executados pelos processos $\{p_0, \ldots, p_{n-1}\}$ e

(ii) o *programa de controle*, responsável pelos aspectos gerencias.

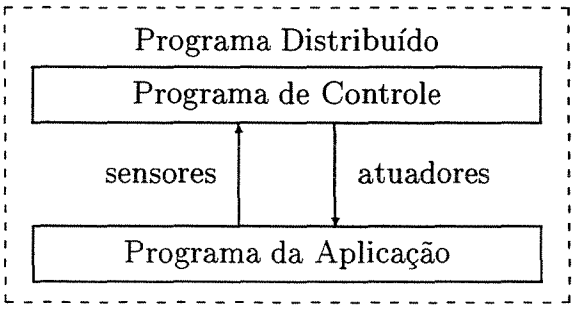


Figura 2: Arquitetura de *software* para monitorização

O sistema de controle amostra o estado do programa da aplicação através de *sensores* e executa as ações de controle através de *atuadores*. Considere, por exemplo, uma aplicação distribuída baseada em um modelo cliente-servidor, para a qual o programa de controle é responsável pela detecção e recuperação de *deadlocks* [16]. Através dos sensores, o programa de controle é capaz de verificar a ocorrência de um *deadlock*, e, através dos atuadores, pode agir sobre o programa da aplicação eliminando o *deadlock*. A noção de consistência entre eventos é essencial para que o sistema de controle possa interpretar corretamente o estado global do programa da aplicação e atuar somente quando for realmente necessário.

1.1 Consistência entre eventos

A execução de um processo p_i é modelada por uma seqüência possivelmente infinita de eventos $(e_i^0, e_i^1, \ldots)$ que podem ser divididos em eventos internos ou de comunicação. A noção de consistência está fortemente acoplada ao conceito de precedência causal entre eventos, proposto por Lamport [33, 43].

Definição 1.1 Precedência causal—*Um evento e_a^α precede um evento e_b^β ($e_a^\alpha \rightarrow e_b^\beta$) se*

(i) $a = b$ *and* $\beta = \alpha + 1$, *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 Figura 3 ilustra os casos em que ocorre relação de precedência: $e_0^0 \rightarrow e_0^1$ pois ambos os eventos pertencem ao mesmo processo e e_0^0 ocorre imediatamente antes de e_0^1; $e_0^1 \rightarrow e_1^1$ pois m foi enviada em e_0^1 e recebida em e_1^1; $e_0^0 \rightarrow e_1^1$ pois $e_0^0 \rightarrow e_0^1$ e $e_0^1 \rightarrow e_1^1$. Dois eventos e e e' são *concorrentes* ($e \parallel e'$) se não estiverem relacionados pela precedência causal. Na Figura 3, podemos observar que $e_0^2 \parallel e_1^1$ e $e_0^2 \parallel e_1^2$.

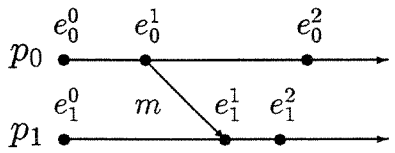


Figura 3: Precedência causal entre eventos

Um corte de uma computação distribuída contém um prefixo da execução de cada um dos processos e é *consistente* se for fechado à esquerda com relação à precedência causal.

Definição 1.2 Corte consistente—*Um corte $\mathcal{C}$ é consistente se, e somente se,*

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

A Figura 4 apresenta um corte inconsistente $\mathcal{I}$ e um corte consistente $\mathcal{C}$. O corte $\mathcal{I}$ é inconsistente porque a mensagem m foi recebida, mas não enviada em $\mathcal{I}$. O corte $\mathcal{C}$ é consistente pois todas as mensagens recebidas em $\mathcal{C}$ também foram enviadas em $\mathcal{C}$. Note que $\mathcal{C}$ é consistente apesar de a mensagem m' ter sido enviada, mas não recebida, em $\mathcal{C}$.

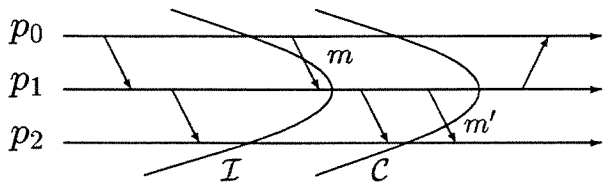


Figura 4: Consistência entre eventos

Um estado global consistente é formado pelos estados de cada um dos processos na fronteira de um corte consistente. Considerando a Definição 1.2, um conjunto de eventos $\{e_0^{\iota_0}, e_1^{\iota_1}, \ldots, e_{n-1}^{\iota_{n-1}}\}$ é fronteira de um corte consistente se, e somente se,

$$\forall i, j : 0 \leq i, j < n : e_i^{\iota_i+1} \not\rightarrow e_j^{\iota_j}$$

1.2 Consistência entre *checkpoints*

Uma computação distribuída pode gerar um número excessivamente alto de eventos, tornando inviável a análise completa de sua execução. Este fato leva à necessidade de se fazer uma abstração da computação através da seleção de alguns eventos de interesse, denominados *checkpoints*. Na Figura 5 mostramos uma abstração no diagrama espaço-tempo utilizando quadrados para representar *checkpoints*.

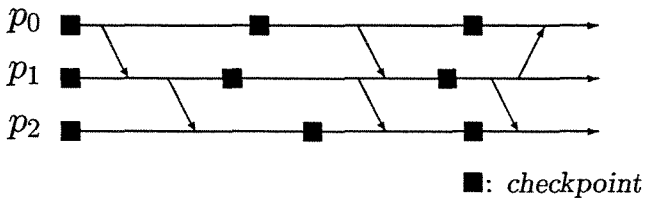
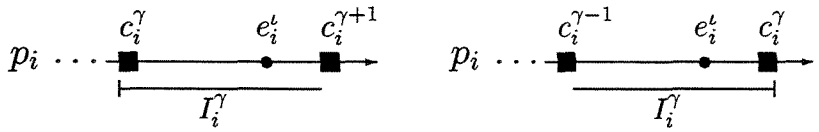


Figura 5: Abstração de uma computação distribuída

O γ-ésimo *checkpoint* de um processo p_i é representado por c_i^γ e deve corresponder a algum evento interno $e_i^{\gamma'}$ com $\gamma \leq \gamma'$. Consideramos que os processos selecionam um *checkpoint* inicial imediatamente após o começo da computação e um *checkpoint* final imediatamente antes do término da computacão.

Dois *checkpoints* consecutivos em um mesmo processo determinam um intervalo de *checkpoints* que abstrai o conjunto de eventos executados entre estes dois *checkpoints*. Intervalos de *checkpoint* podem ser rotulados de duas maneiras. Na rotulação à esquerda, o intervalo entre c_i^γ e $c_i^{\gamma+1}$ é rotulado I_i^γ (Figura 6 (a)). Na rotulação à direita, o intervalo entre $c_i^{\gamma-1}$ e c_i^γ é rotulado I_i^γ (Figura 6 (b)).



(a) rotulação à esquerda (b) rotulação à direita

Figura 6: Rótulos para intervalos de *checkpoint*

Um *checkpoint* global é formado selecionando-se um *checkpoint* para cada um dos processos da computação. Um *checkpoint* global é consistente se estiver associado a um corte consistente. Na Figura 7, por exemplo, o corte consistente C define um *checkpoint* global consistente.

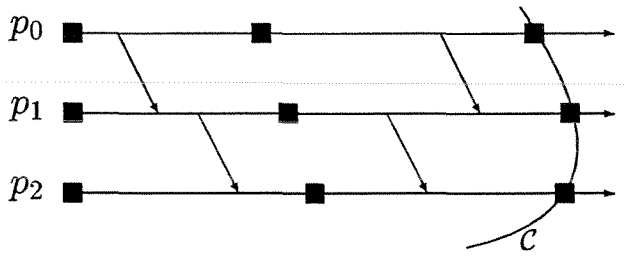


Figura 7: Consistência entre *checkpoints*

Dizemos que $c_a^\alpha \rightarrow c_b^\beta$ se o evento correspondente a c_a^α precede causalmente o evento correspondente a c_b^β. A restrição de que *checkpoints* estão relacionados a eventos internos permite definir *checkpoints* globais consistentes em função da relação de precedência causal entre *checkpoints*.

Definição 1.3 Checkpoint global consistente—*Um checkpoint global* $\hat{\Sigma} = \{c_0^{\iota_0}, \ldots, c_{n-1}^{\iota_{n-1}}\}$ *é consistente se, e somente se,*

$$\forall i, j : 0 \leq i, j < n : c_i^{\iota_i} \not\rightarrow c_j^{\iota_j}$$

Segundo a Definição 1.3, *checkpoints* globais consistentes devem conter apenas *checkpoints* concorrentes. No entanto, dois *checkpoints* podem ser concorrentes entre si e não fazer parte de nenhum *checkpoint* global consistente. Na Figura 8, os *checkpoints* c_0^0 e c_2^1 são concorrentes, mas não há *checkpoint* em p_1 que forme um *checkpoint* global consistente com eles: $\{c_0^0, c_1^0, c_2^1\}$ é inconsistente pois $c_1^0 \rightarrow c_2^1$ e $\{c_0^0, c_1^1, c_2^1\}$ é inconsistente pois $c_0^0 \rightarrow c_1^1$.

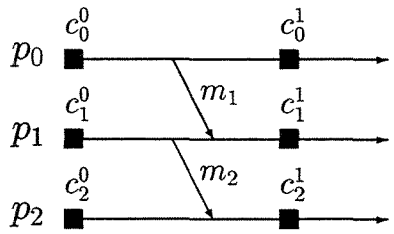


Figura 8: Checkpoints c_0^0 e c_2^1 não são consistentes

1.3 *Zigzag paths*

Netzer e Xu determinaram as condições necessárias e suficientes para que um conjunto de *checkpoints* possa fazer parte de um mesmo *checkpoint* global consistente através de seqüências de mensagens denominadas *zigzag paths* [38].

Definição 1.4 ***Zigzag paths***—*Uma seqüência de mensagens $m_1, m_2, \ldots, m_k$ é uma zigzag path que liga um checkpoint c_a^α a um checkpoint c_b^β se, e somente se:*

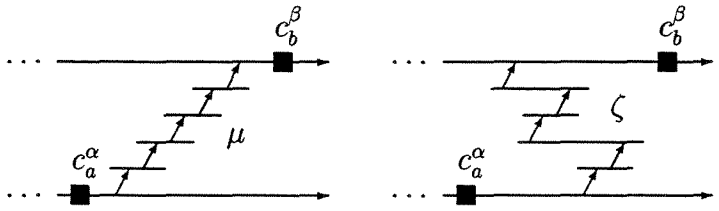
(i) a mensagem m_1 é enviada por p_a após c_a^α e

(ii) a mensagem m_i, $(i < k)$, é recebida pelo mesmo processo que envia m_{i+1}, mas m_{i+1} não pode ser enviada em um intervalo anterior ao que m_i foi recebida, e

(iii) m_k é recebida por p_b antes de c_b^β.

Um conjunto de *checkpoints* não pode fazer parte de um mesmo *checkpoint* global consistente se houver uma *zigzag path* entre membros deste conjunto. Na Figura 8, devido à *zigzag path* $[m_1, m_2]$ que liga c_0^0 a c_2^1, não há *checkpoint* em p_1 que forme um *checkpoint* global consistente com estes *checkpoints*.

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} (Figura 9 (a)). 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} (Figura 9 (b)). Alguns artigos utilizam o termo *C-path* para denotar *zigzag paths* causais e *Z-path* para denotar *zigzag paths* não causais [6, 7]. Em outros trabalhos, o termo *Z-path* é utilizado como abreviatura para *zigzag path*, independentemente do fato de a *zigzag path* ser causal ou não [1, 28, 40, 46].



(a) causal (b) não-causal

Figura 9: *Zigzag paths*

Um *zigzag cycle (Z-cycle)* é formado por uma *zigzag path* que liga um *checkpoint* a ele mesmo. Um *Z-cycle* corresponde a um *checkpoint* inútil, ou seja, um *checkpoint* que não pode fazer parte de nenhum *checkpoint* global consistente [38]. Na Figura 10, c_b^β é inútil porque não há *checkpoint* em p_a que possa fazer parte de um *checkpoint* global consistente com c_b^β, visto que a *zigzag path* $[m_1, m_2]$ liga c_a^α a c_b^β e a *zigzag path* $[m_3]$ liga c_b^β a $c_a^{\alpha+1}$.

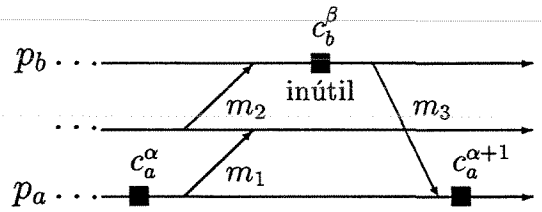


Figura 10: *Z-cycle*

Manivannan, Netzer e Singhal [34] estenderam a definição de *zigzag paths* para incluir *checkpoints* selecionados pelo mesmo processo. Pela definição estendida, sempre existe uma *zigzag path* ligando c_i^γ a $c_i^{\gamma+1}$ independentemente do fato de existir uma seqüência de mensagens causal ou não causal ligando c_i^γ a $c_i^{\gamma+1}$. Apesar de mais uniforme, esta definição é pouco utilizada na literatura.

1.4 Rastreando dependências entre *checkpoints*

Um mecanismo transitivo para propagação de informação entre os processos pode ser aplicado para se rastrear dependências entre *checkpoints*. Vamos supor que os intervalos são rotulados à direita (Figura 6 (b)) e que existe uma *zigzag path* do intervalo I_a^α para o intervalo I_b^β se existe uma *zigzag path* ligando o *checkpoint* $c_a^{\alpha-1}$ a c_b^β.

Cada processo p_i armazena e propaga um vetor de dependências, dv_i, com n entradas tais que a entrada $dv_i[i]$ indica o intervalo corrente de p_i e as outras entradas $dv_i[j]$, $j \neq i$, indicam o índice do último intervalo de p_j que p_i teve conhecimento. Todas as entradas de dv são iniciadas com 0 e a entrada $dv_i[i]$ é incrementada imediatamente *após* a retirada de um *checkpoint*, incluindo o *checkpoint* inicial.

Quando o processo p_i envia uma mensagem m, ele agrega o seu vetor de dependências à mensagem, que é denotado por $m.dv$. Quando p_i recebe uma mensagem m de p_j, p_i atualiza o vetor dv_i fazendo uma operação de máximo para cada uma das entradas correspondentes a um mesmo processo p_k da seguinte forma:

$$\forall k : 0 \leq k < n : dv_i[k] \leftarrow \max(dv_i[k], m.dv[k])$$

A Figura 11 (a) apresenta uma computação distribuída com vetores de dependências. O vetor de dependências associado ao *checkpoint* c_2^1 é (1 1 1) e captura a existência das seguintes *zigzag paths* que alcançam c_2^1:

- $[m_1, m_2]$ que parte do intervalo I_0^1 e
- $[m_2]$ que parte do intervalo I_1^1.

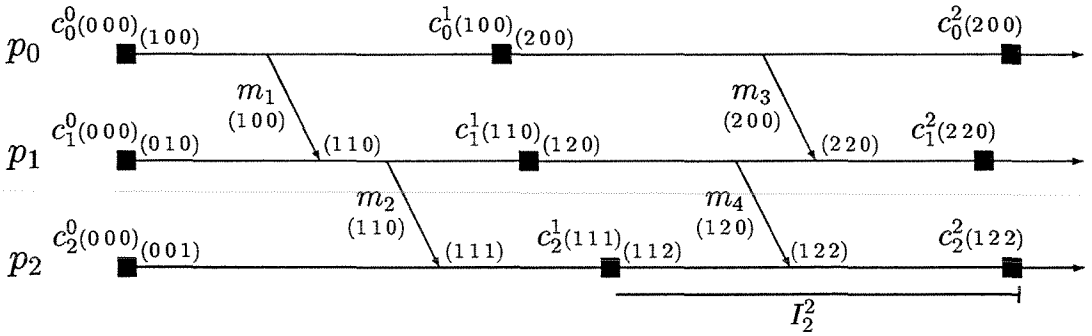
Desta forma, o vetor captura (1 1 1) corretamente todos intervalos dos quais partem *zigzag paths* que alcançam o *checkpoint* c_2^1. No entanto, nem todas as *zigzag paths* podem ser rastreadas utilizando-se este mecanismo. O vetor de dependências associado ao *checkpoint* c_2^2 é (1 2 2) e não captura a *zigzag path* $[m_3, m_4]$ que parte de I_0^2 e alcança o *checkpoint* c_2^2. Isto ocorreu porque p_1 alterou o seu vetor de dependências após ter enviado m_4.

Existem mecanismos equivalentes para o rastreamento de dependências, com pequenas variações de numeração e nomenclatura. Vamos supor que os intervalos entre *checkpoint* são rotulados à esquerda (Figura 6 (a)) e que existe uma *zigzag path* do intervalo I_a^α para o intervalo I_b^β se existe uma *zigzag path* ligando o *checkpoint* c_a^α a c_b^β.

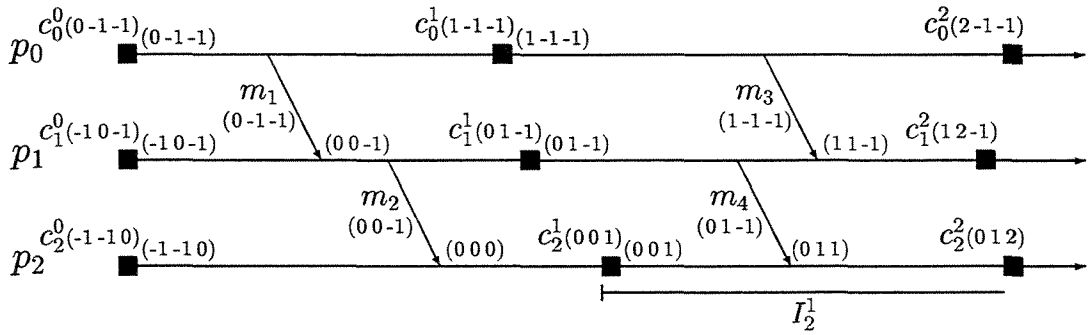
Cada processo p_i armazena e propaga um vetor de relógios, vc_i, com n entradas tais que a entrada $vc_i[i]$ indica o último *checkpoint* retirado por p_i e as outras entradas $vc_i[j]$, $j \neq i$, indicam o índice do último *checkpoint* de p_j que p_i teve conhecimento. Todas as entradas de vc são iniciadas com -1 e a entrada $vc_i[i]$ é incrementada imediatamente *antes* da retirada de um *checkpoint*, incluindo o *checkpoint* inicial.

A Figura 11 (b) apresenta o mesmo cenário da Figura 11 (a) com vetores de relógios. Pode-se observar novamente que o vetor associado ao *checkpoint* c_2^1 captura todas as *zigzag paths* que alcançam o intervalo I_2^0, enquanto o vetor associado ao *checkpoint* c_2^2 não captura a *zigzag paths* que alcançam o intervalo I_2^1.

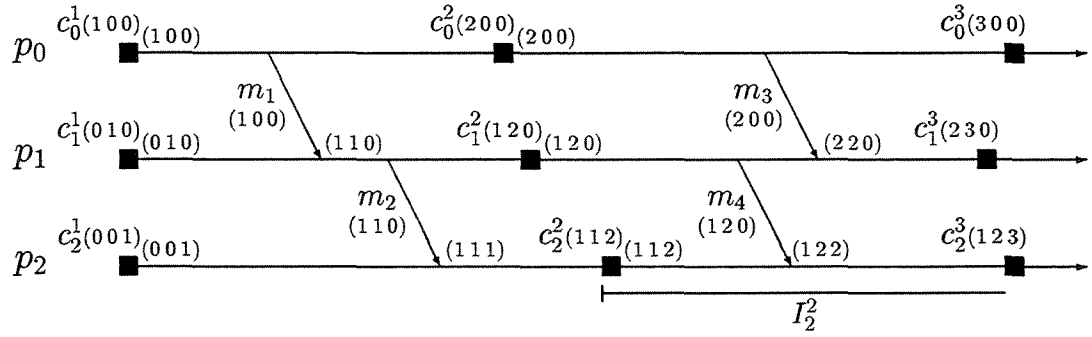
Para se evitar a utilização de números negativos, pode-se iniciar todas as entradas vc_i com 0. Neste caso, seria mais coerente rotular o *checkpoint* inicial com c_i^1 e não com c_i^0. A Figura 11 (c) apresenta o mesmo cenário das Figuras 11 (a,b) com esta opção para vetores de relógios. Note que apesar de a informação propagada na Figura 11 (c) ser idêntica à apresentada na Figura 11 (a), os vetores associados aos *checkpoints* são distintos.



(a) vetores de dependências



(b) vetores de relógios



(c) vetores de relógios iniciados com $(0, 0, \ldots, 0)$

Figura 11: Rastreando dependências entre *checkpoints*

2 Abordagens para a seleção de *checkpoints*

Quando a abstração de uma computação distribuída tem por objetivo a formação de *checkpoints* globais consistentes, os *checkpoints* desta abstração não podem ser escolhidos aleatoriamente. Esta limitação foi originalmente detectada por Randell no contexto de recuperação de falhas por retrocesso de estado [41]. Em um cenário denominado *efeito dominó*, uma aplicação pode ter de retroceder ao seu estado inicial após a ocorrência de uma falha, apesar de ter gravado *checkpoints* ao longo de sua execução (Figura 12).

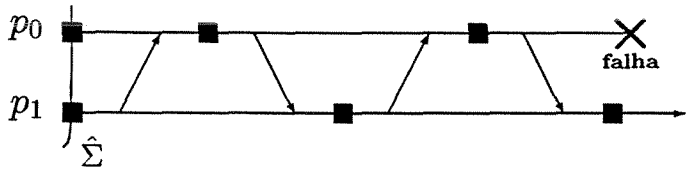


Figura 12: Efeito dominó

Para garantir a ausência de efeito dominó, podemos acrescentar sincronismo à seleção de *checkpoints*. Quando um processo p_i seleciona um *checkpoint*, ele assume a postura de um coordenador, enviando mensagens de requisição para que os outros processos da computação também selecionem *checkpoints*. Além disso, p_i interrompe a sua execução até a recepção de mensagens de confirmação da retirada de *checkpoints* pelos outros processos da computação (Figura 13). Na ausência de falhas durante este processo, os *checkpoints* mais recentes de cada processo formam um *checkpoint* global consistente (mensagens de controle não influenciam a consistência) [14, 32].

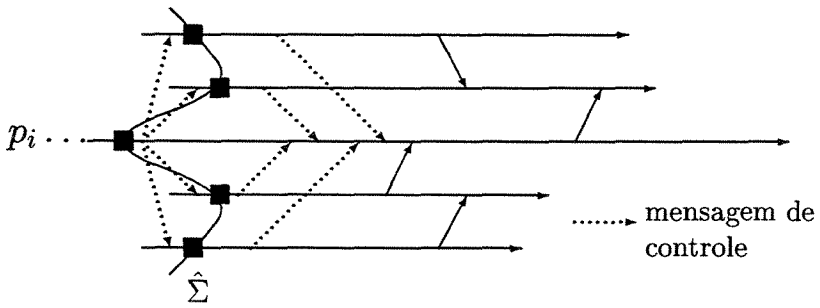


Figura 13: Abordagem síncrona

A abordagem síncrona é intrusiva e pode resultar em perda de desempenho devido às mensagens de controle e à suspensão das atividades da computação durante a retirada do

checkpoint global. Recentemente, foram propostos algoritmos síncronos que não bloqueiam a execução dos processos durante a retirada do *checkpoint* global, mas estes algoritmos ainda apresentam um custo alto em termos de mensagens de controle [12, 13, 39].

A abordagem quase-síncrona, também chamada de induzida por comunicação, está baseada em um protocolo obedecido pelos processos da aplicação para a seleção de *checkpoints*. Prioritariamente, os processos selecionam *checkpoints* livremente, chamados *checkpoints* básicos. Eventualmente, os processo podem ser induzidos a selecionar *checkpoints* adicionais, chamados *checkpoints* forçados, segundo predicados avaliados sobre informações de controle propagadas através das mensagens da aplicação [17, 36]. Posteriormente, *checkpoints* globais consistentes são formados a partir dos *checkpoints* selecionados (Figura 14). Esta abordagem apresenta um compromisso entre a autonomia dos processos para a escolha dos *checkpoints* e garantias oferecidas para a formação de *checkpoints* globais consistentes.

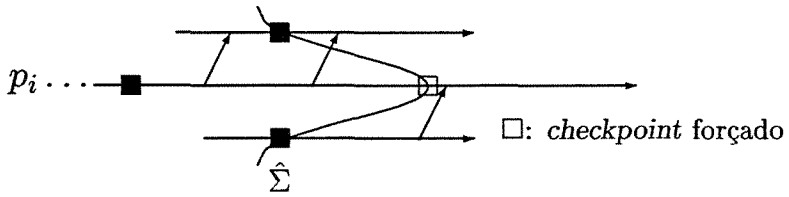


Figura 14: Abordagem quase-síncrona

3 Protocolos quase-síncronos

Considerando os conceitos de *zigzag paths* e *Z-cycles* [38], Manivannan e Singhal dividiram os padrões de *checkpoint* em quatro classes distintas: *Strictly Z-Path Free* (SZPF), *Z-Path Free* (ZPF), *Z-Cycle Free* (ZCF), e *Partially Z-Cycle Free* (PZCF) [36]. Estas classes respeitam a relação de continência ilustrada na Figura 15.

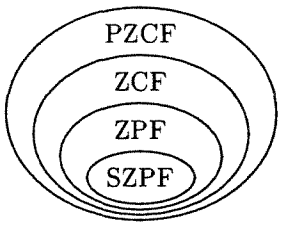


Figura 15: Classes de protocolos quase-síncronos

3.1 *Strictly Z-Path Free (SZPF)*

Em um padrão SZPF, todas as *zigzag paths* devem ser causais. Desta forma, através da indução de *checkpoints* forçados, é garantido que todos os eventos de recepção de mensagem precedem os eventos de envio de mensagem em um mesmo intervalo de *checkpoints*. Em 1980, esta abordagem foi proposta através da especificação do Modelo *Mark-Receive-Send* (Figura 16) [42].

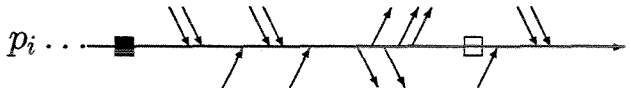
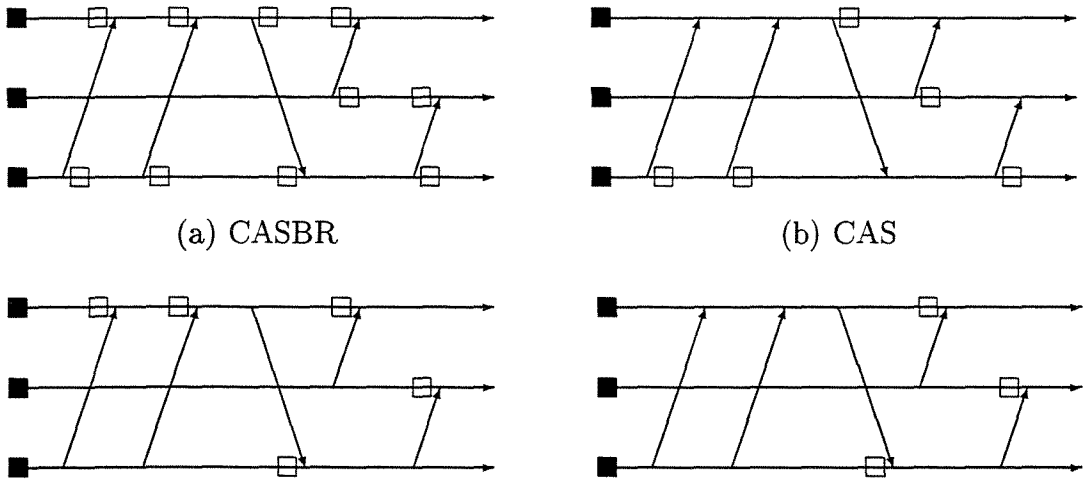


Figura 16: Modelo *Mark-Receive-Send*

Protocolos SZPF garantem a ausência de *checkpoints* inúteis e permitem que todas as dependências possam ser rastreadas utilizando-se vetores de dependências ou vetores de relógios. Wang identificou quatro protocolos SZPF: *Checkpoint-After-Send-Before-Receive*, *Checkpoint-After-Send*, *Checkpoint-Before-Receive* e *No-Receive-After-Send* [50], ilustrados na Figura 17.



(a) CASBR (b) CAS

(c) CBR (d) NRAS

Figura 17: Protocolos SZPF

3.2 *Z-Path Free (ZPF)*

Uma *zigzag path* não causal ζ que liga c_a^α a c_b^β está *duplicada causalmente* se existe uma *zigzag path* causal μ que também liga c_a^α a c_b^β (Figura 18) [3, 6, 7]. Um padrão ZPF pode conter *zigzag paths* não causais que estejam duplicadas causalmente.

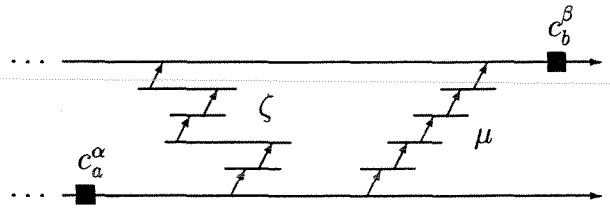


Figura 18: Duplicação causal

Considere que os processos da computação armazenam e propagam vetores de dependências como descrito na Seção 1.4. Se cada processo p_i selecionar um *checkpoint* imediatamente antes da recepção de uma mensagem m com informações novas em seu vetor de dependências, todas as *zigzag paths* não causais serão duplicadas. Na Figura 19, um *checkpoint* forçado é induzido em p_1 antes de m_1 mas não antes de m_3. Além disso, podemos observar que a *zigzag path* não causal $[m_3, m_2]$ está duplicada causalmente pela *zigzag path* causal $[m_1, m_2]$. Este protocolo foi originalmente proposto por Venkatesh, Radhakrishnan e Li [31] e, posteriormente, refraseado por Wang e denominado *Fixed-Dependency-Interval* (FDI) [50].

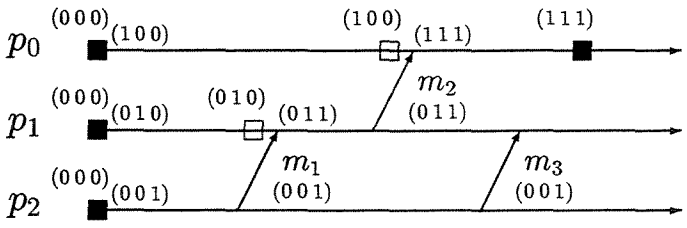


Figura 19: FDI

Padrões ZPF, incluindo os SZPF, obedecem à propriedade *Rollback-Dependency Trackability* (RDT), ou seja, todas as dependências entre *checkpoints* podem ser rastreadas em tempo de execução através da utilização de vetores de dependências ou vetores de relógios [50]. Na Seção 4, analisaremos esta propriedade e descreveremos outros protocolos ZPF.

3.3 *Z-Cycle Free (ZCF)*

Padrões ZCF podem conter *zigzag paths* não causais, mas não podem conter *Z-cycles*. Desta forma, garantem que todos os *checkpoints* são úteis, ou seja, podem fazer parte de pelo menos um *checkpoint* global consistente.

A maioria dos protocolos ZCF utiliza índices semelhantes aos relógios lógicos propostos por Lamport [33]. O protocolo ZCF mais simples foi proposto por Briatico, Ciuffoletti e Simoncini (BCS) [11]. Cada processo p_i armazena e propaga um índice, idx_i, tal que idx_i é iniciado com 0 e incrementado imediatamente antes de um *checkpoint* básico. Quando p_i envia uma mensagem m, o valor idx_i é agregado à mensagem m e denotado por $m.idx$. Quando p_i recebe uma mensagem m de p_j com $m.idx > idx_i$, p_i atualiza idx_i e um *checkpoint* forçado é induzido imediatamente antes do processamento de m. A Figura 20 ilustra o funcionamento do protocolo BCS. Note que existe uma *zigzag path* $[m_1, m_2]$ que não é duplicada causalmente e note que um *checkpoint* é induzido imediatamente antes da formação do *Z-cycle* $[m_1, m_2, m_3]$. Várias otimizações do protocolo BCS que geram padrões ZCF foram propostas na literatura [9, 27, 28, 35].

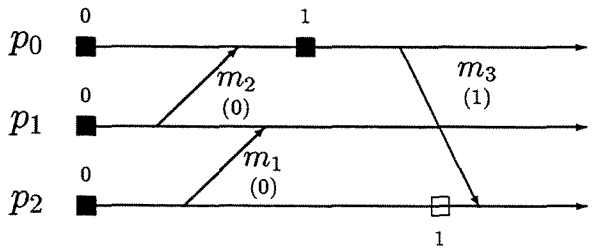


Figura 20: BCS

Outra abordagem para a obtenção de padrões ZCF foi apresentada no protocolo proposto por Baldoni, Quaglia e Ciciani (BQC) [8, 40]. Este protocolo detecta a formação de um padrão denominado *Suspect Z-cycle* (Figura 21) que corresponde à menor seqüência de mensagens rastreável em tempo de execução que pode dar origem a um *Z-cycle*. Altamente custoso, controla a indução de *checkpoints* através da propagação de matrizes de relógios.

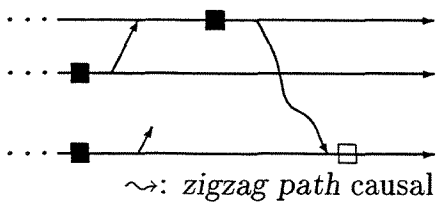


Figura 21: *Suspect Z-Cycle*

3.4 *Partially Z-cycle Free (PZCF)*

Um padrão PZCF pode conter *Z-cycles* e portanto *checkpoints* inúteis. No entanto, espera-se que seja feito um esforço para que pelo menos uma parte dos *checkpoints* sejam úteis. O protocolo proposto por Xu e Netzer (XN) [55] tenta diminuir a ocorrência de *checkpoints* inúteis, através da detecção de *Z-cycle* formados por uma mensagem m e uma *zigzag path* causal μ (Figura 22).

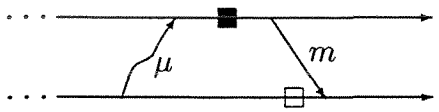


Figura 22: Protocolo proposto por Xu e Netzer

O protocolo proposto por Wang e Fuchs (WF) [52] propõe uma variação do protocolo BCS [11] que restringe a indução de *checkpoints* forçados para índices múltiplos de um determinado valor γ. Um *checkpoint* cujo índice é múltiplo de γ é garantidamente útil, enquanto os outros *checkpoints* podem ser úteis ou não. A Figura 23 ilustra o funcionamento do protocolo WF para $\gamma = 2$.

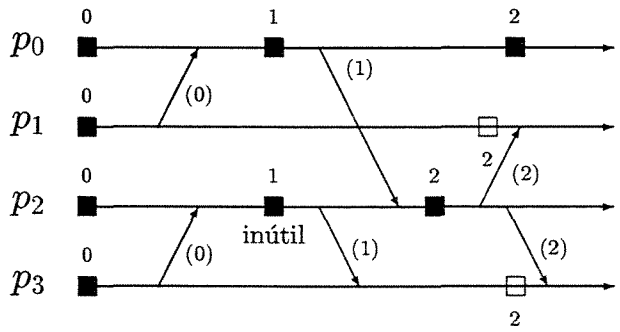


Figura 23: Protocolo proposto por Wang e Fuchs

4 *Rollback-Dependency Trackability*

Na literatura, existem duas abordagens para a análise da propriedade RDT. A primeira considera o estudo de dependências rastreáveis em tempo de execução utilizando vetores de dependências [50]. A segunda abordagem é baseada no estudo de dependências entre *checkpoints* utilizando *zigzag paths* [6, 7].

4.1 Caracterização baseada em vetores de dependências

A propriedade RDT foi introduzida por Wang utilizando *R-graphs*, grafos dirigidos [10] em que nós representam *checkpoints* e arestas representam dependências entre *checkpoints* [50]. Um caminho de c_a^α a c_b^β em um *R-graph* tem o seguinte significado: se p_a precisa retroceder a um estado anterior a c_a^α, então p_b também precisa retroceder a um estado anterior a c_b^β. As definições de *R-graph* e de RDT estão relacionadas à rotulação de intervalos à direita (Figura 6 (b)) e à utilização de vetores de dependências (Seção 1.4).

Definição 4.1 ***R-graph***—*Um R-graph é um grafo dirigido em que cada nó representa um checkpoint e há uma aresta de c_a^α a c_b^β se (i) $a = b$ e $\beta = \alpha + 1$ ou (ii) $a \neq b$ e uma mensagem m é enviada em I_a^α e recebida em I_b^β.*

Definição 4.2 Rollback-Dependency Trackability—*Um padrão de checkpoints obedece à propriedade RDT se para todo par de checkpoints $c_a^\alpha(\alpha \neq 0)$ e c_b^β a seguinte regra é válida:*

Existe um caminho de c_a^α para c_b^β no R-graph *se, e somente se, $dv(c_b^\beta)[a] \geq \alpha$.*

A Figura 24 reapresenta a computação distribuída da Figura 11 (a), apresentando o seu respectivo *R-graph*. Este padrão de *checkpoints* não obedece à propriedade RDT, pois existe um caminho no *R-graph* de c_0^2 a c_2^2, mas $dv(c_2^2)[0] = 1$.

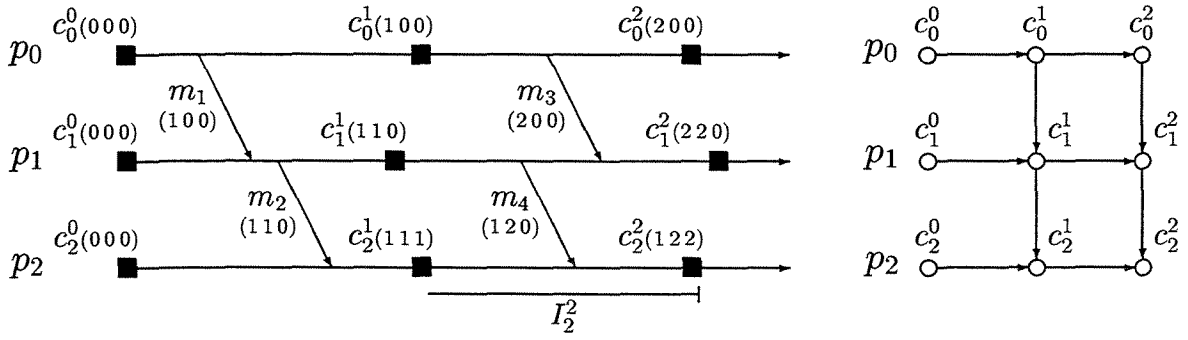


Figura 24: *R-graph*

Wang verificou que se um processo não alterar o seu vetor de dependências após o primeiro evento de envio em um intervalo, o padrão de *checkpoints* resultante obedece à propriedade RDT [50]. O protocolo baseado nesta observação foi denominado *Fixed-Dependency-After-Send* (FDAS) e é uma otimização do protocolo FDI descrito na Seção 3.2. A Figura 25 reapresenta o cenário da Figura 24 sob o protocolo FDAS e mostra o novo *R-graph* resultante.

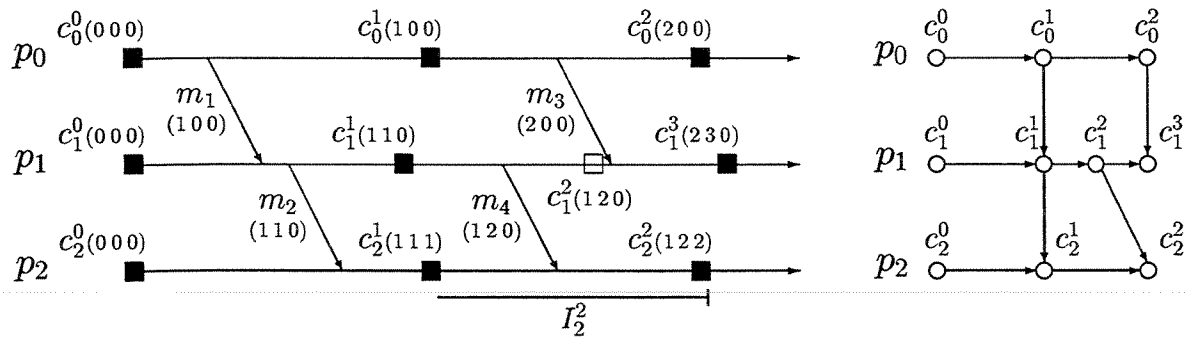


Figura 25: FDAS

Soluções simples para a construção de *checkpoints* globais consistentes que incluem um grupo de *checkpoints* são possíveis em padrões RDT [50]. Em particular, o vetor de dependências associado a um *checkpoint* c indica o *checkpoint* global consistente mínimo (mais à esquerda no diagrama espaço-tempo) ao qual c pertence. Na Figura 25, $dv(c_0^1)$ é (1 0 0) e $\{c_0^1, c_1^0, c_2^0\}$ é o *checkpoint* global consistente mínimo ao qual c_0^1 pertence.

Em padrões RDT, um grupo de *checkpoints* concorrentes C sempre pode fazer parte de um mesmo *checkpoint* global consistente. Em particular, o *checkpoint* global consistente mínimo que contém C pode ser calculado a partir dos vetores de dependências dos *checkpoints* em C. Para cada processo p_i, basta utilizar o valor máximo de $dv[i]$ presente nos vetores de dependências dos *checkpoints* em C. Na Figura 25, c_0^1 e c_1^1 são concorrentes, $dv(c_0^1) = (1\,0\,0)$, $dv(c_1^1) = (1\,1\,0)$ e $\{c_0^1, c_1^1, c_2^0\}$ é o *checkpoint* global consistente mínimo que contém C.

A determinação do *checkpoint* global consistente máximo (mais à direita no diagrama espaço tempo) que contém um grupo de *checkpoints* concorrentes C pode ser feita a partir de uma busca no *R-graph*. Seja C_{next} o conjunto dos *checkpoints* que sucedem imediatamente os *checkpoints* em C em seus respectivos processos. Todos os nós alcançáveis a partir de C_{next} devem ser marcados e os últimos nós não marcados em cada processo formam o *checkpoint* global consistente máximo que contém C. Vamos considerar $C = \{c_0^1\}$ e, portanto $C_{next} = \{c_0^2\}$. O único nó marcado em uma busca a partir de c_0^2 é c_1^3. Desta forma, o *checkpoint* global máximo que contém c_0^1 é $\{c_0^1, c_1^2, c_2^2\}$.

4.2 Caracterização baseada em *zigzag paths*

Nesta Seção, vamos utilizar o termo *C-path* para denotar uma *zigzag path* causal e *Z-path* para denotar uma *zigzag path* não causal. Em um padrão RDT, todas as *Z-paths* devem estar duplicadas causalmente, ou seja, todos os *checkpoints* conectados por uma *Z-path*

devem estar também conectados por uma *C-path*. A abordagem proposta por Baldoni, Helary e Raynal consiste na redução do conjunto de *Z-paths* que precisam ser duplicadas para que a propriedade RDT possa ser garantida [6, 7].

Uma *Z-path* ζ é formada pela concatenação de k *C-paths* $\zeta = \mu_1 \cdot \mu_2 \cdot \ldots \cdot \mu_k$, onde k é a *ordem* desta *Z-path* (Figura 26).

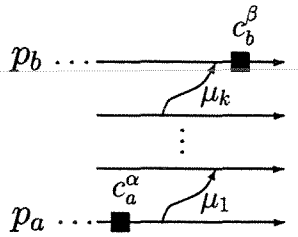
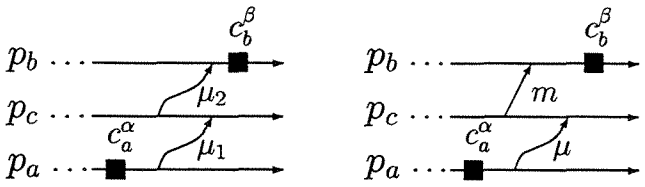


Figura 26: *Z-path* de ordem n

Uma *Z-path* de ordem 2 *(CC-path)* é composta de exatamente duas *C-paths* μ_1 e μ_2 (Figura 27 (a)). Baldoni, Helary e Raynal provaram que um padrão de *checkpoints* no qual todas as *CC-paths* estão duplicadas causalmente obedece à propriedade RDT [6, 7].

Uma *CM-path* é uma *CC-path* formada por uma *C-path* μ e uma única mensagem m (Figura 27 (b)). Baldoni, Helary e Raynal também provaram que basta duplicar causalmente todas as *CM-paths* para garantir que um padrão de *checkpoints* obedece à propriedade RDT [6, 7]. Uma *CM-path*$\mu \cdot [m]$ pode ser rastreada em tempo de execução pelo processo que envia m e portanto ser utilizada para a especificação de protocolos RDT.



(a) *CC-path* (b) *CM-path*

Figura 27: *Z-paths* de ordem 2

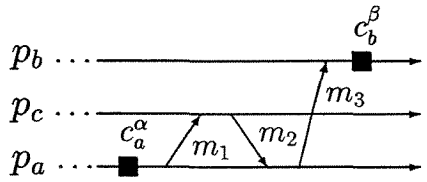
A seguir, definimos restrições que podem ser aplicadas à componente causal μ de maneira a diminuir o conjunto de *Z-paths* que devem estar duplicadas causalmente para garantir RDT.

Definição 4.3 Elementary path—*Uma C-path μ é uma elementary path se a seqüência de processos percorrida por μ não tem repetições.*

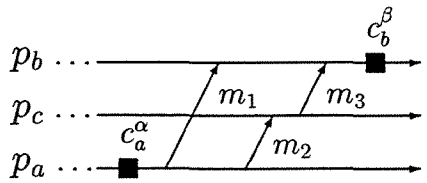
Definição 4.4 Prime path—*Uma C-path μ que conecta c_a^α a c_b^β é uma prime path se a última mensagem de μ é a primeira mensagem recebida por p_b que carrega informação a respeito de c_a^α.*

Definição 4.5 Simple path—*Uma C-path $\mu = [m_1, \ldots, m_k]$ é uma simple path se para toda mensagem m_i, $1 \leq i < k$, a recepção de m_i e o envio de m_{i+1} ocorrem no mesmo intervalo de checkpoints.*

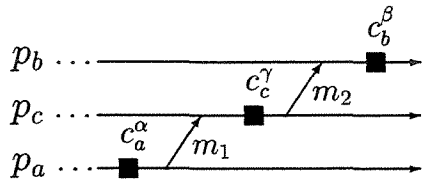
Considere as *C-paths* que ligam c_a^α a c_b^β, como mostrado nas Figuras 28 (a,b,c). Na Figura 28 (a), $[m_1, m_2, m_3]$ não é uma *elementary path* porque a seqüência de processos percorrida (p_a, p_c, p_a e p_b) apresenta uma repetição. Em contrapartida, a mensagem m_3 é uma *elementary path*. Na Figura 28 (b), $[m_2, m_3]$ não é uma *prime path* porque o processo p_b já tinha recebido informação a respeito do *checkpoint* c_a^α através da mensagem m_1. Neste caso, m_1 é uma *prime path*. Na Figura 28 (c), $[m_1, m_2]$ não é uma *simple path* devido à presença do *checkpoint* c_c^γ. Estes conceitos nos ajudam a definir *PCM-paths*, *EPCM-paths* e *EPSCM-paths* (Figura 29).



(a) $[m_1, m_2, m_3]$ não é uma *elementary path*



(b) $[m_2, m_3]$ não é uma *prime path*



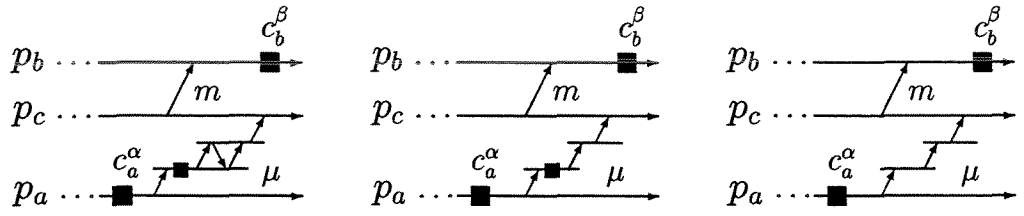
(c) $[m_1, m_2]$ não é uma *simple path*

Figura 28: Restrições para *C-paths*

Definição 4.6 PCM-path—*Uma PCM-path $\zeta = \mu \cdot [m]$ é uma CM-path formada por uma C-path μ e uma mensagem m tal que μ é uma prime path.*

Definição 4.7 EPCM-path—*Uma EPCM-path* $\zeta = \mu \cdot [m]$ *é uma PCM-path tal que* μ *é uma elementary path.*

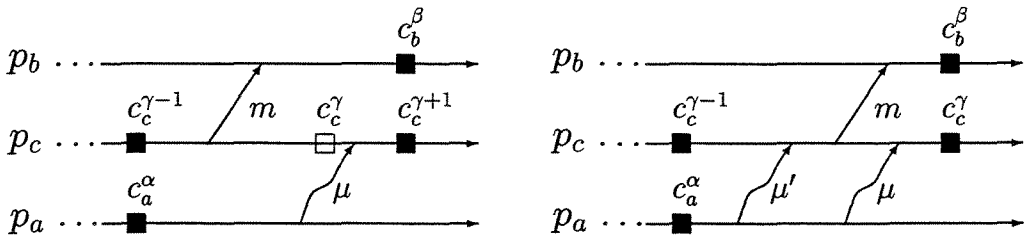
Definição 4.8 EPSCM-path—*Uma EPSCM-path* $\zeta = \mu \cdot [m]$ *é uma EPCM-path tal que* μ *é uma simple path.*



(a) *PCM-path* (b) *EPCM-path* (c) *EPSCM-path*

Figura 29: *CM-paths* com restrições

A implementação de um protocolo que evita a formação de *PCM-paths*, *EPCM-paths* ou *EPSCM-paths* é bastante simples [7]. Em particular, um protocolo que evita a formação de todas as *PCM-paths* pode ser visto como uma reinterpretação do protocolo FDAS (evitar a formação de uma *PCM-path* é equivalente a evitar uma alteração no vetor de dependências após um evento de envio). Na Figura 30 (a), um *checkpoint* forçado é induzido em p_c imediatamente antes da formação da *PCM-path* $\mu \cdot [m]$ ou para evitar a alteração da entrada $dv_c[a]$ após um evento de envio de mensagem. Na Figura 30 (b), não há a indução de um *checkpoint* forçado pois $\mu \cdot [m]$ não é uma *PCM-path* e a entrada $dv_c[a]$ não é alterada com a recepção da última mensagem de μ.



(a) p_c precisa evitar $\mu \cdot [m]$ (b) $\mu \cdot [m]$ não é uma *PCM-path*

Figura 30: FDAS evita a formação de *PCM-paths*

O processo p_c não precisa evitar a formação de uma *PCM-path* se p_c é capaz de detectar que esta *PCM-path* já está duplicada causalmente. Na Figura 31, a *PCM-path* $\mu \cdot [m]$ está duplicada causalmente pela *C-path* ν e p_c detectou esta duplicação através da *C-path* $\nu' \cdot \mu_2$. Neste caso, dizemos que a *PCM-path* $\mu \cdot [m]$ está *duplicada visivelmente.*

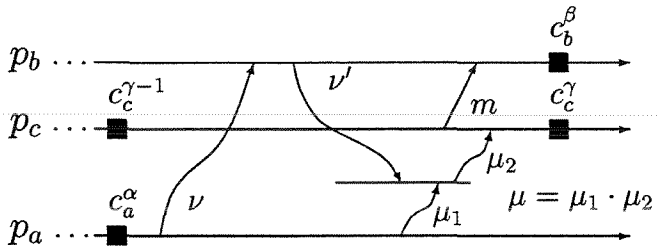


Figura 31: *PCM-path* duplicada visivelmente

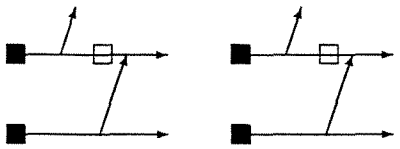
Definição 4.9 PCM-path duplicada visivelmente—*Uma PCM-path* $\mu \cdot [m]$ *é* duplicada visivelmente *se (i) é duplicada causalmente por uma C-path* ν *e (ii) a recepção da última mensagem de* ν *precede causalmente o envio da última mensagem de* μ.

O protocolo proposto por Baldoni, Helary, Mostefaoui e Raynal (BHMR) foi o primeiro protocolo a economizar *checkpoints* forçados devido à detecção de *PCM-paths* duplicadas visivelmente [3]. Posteriormente, Baldoni, Helary e Raynal apresentaram uma versão mais elaborada deste protocolo, que evita apenas a formação de *EPSCM-paths* não duplicadas visivelmente [7]. Estes dois protocolos foram implementados com a manutenção e propagação de informações de controle com complexidade $O(n^2)$ [7].

4.3 Condições para indução de *checkpoints* forçados

Baldoni, Helary e Raynal conjeturaram que o conjunto formado por *EPSCM-paths* não duplicadas visivelmente determina o menor conjunto de *Z-paths* que devem ser evitadas em tempo de execução por um protocolo RDT [6]. Tsai, Kuo e Wang analisaram o impacto teórico da utilização de condições mais restritas para a implementação de protocolos RDT [45].

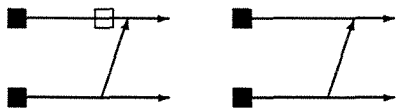
Vamos considerar dois protocolos RDT que induzem *checkpoints* forçados sob as condições $\mathcal{C}_1$ e $\mathcal{C}_2$. Dizemos que $\mathcal{C}_1$ é mais forte (mais restritiva) que $\mathcal{C}_2$ se, para um mesmo passado causal, $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$. Por exemplo, seja $\mathcal{C}_1$ a condição *No-Receive-After-Send* (NRAS) e $\mathcal{C}_2$ a condição *Checkpoint-Before-Receive* (CBR), descritas na Seção 3.1. Para um mesmo passado causal, se o protocolo NRAS induz um *checkpoint* forçado, o protocolo CBR também induzirá (Figura 32).



(a) NRAS (b) CBR

Figura 32: NRAS $\Rightarrow$ CBR

Por outro lado é possível construir um cenário em que, para um mesmo passado causal, o protocolo CBR induz um *checkpoint* forçado, mas o protocolo NRAS não (Figura 33). Logo, NRAS $\Rightarrow$ CBR, mas CBR $\not\Rightarrow$ NRAS.



(a) CBR (b) NRAS

Figura 33: CBR $\not\Rightarrow$ NRAS

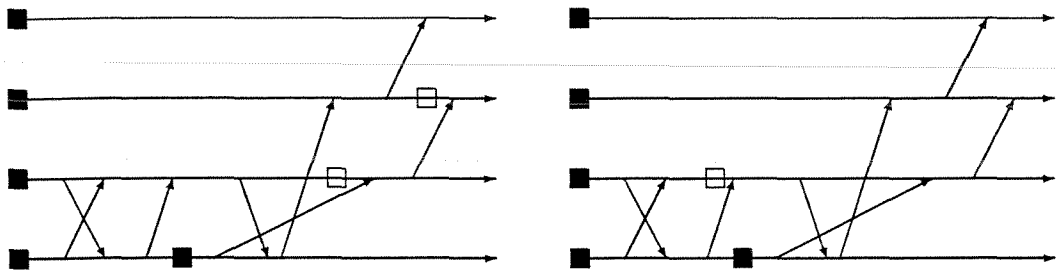
Seja #-forçados($\mathcal{C}$) o número de *checkpoints* forçados induzidos durante uma computação distribuída por um protocolo que utiliza a condição $\mathcal{C}$. Considerando os protocolos NRAS e CBR, podemos verificar que #-forçados($NRAS$) $\leq$ #-forçados(CBR), pois #-forçados(CBR) é determinado pelo número de mensagens da computação, enquanto #-forçados($NRAS$) é determinado pelo sub-conjunto de mensagens da computação que são recebidas após um evento de envio.

Utilizando um raciocínio semelhante, é possível concluir que a condição FDAS $\Rightarrow$ FDI e #-forçados(FDAS) $\leq$ #-forçados(FDI). Como vimos no final da Seção 4.2, a condição utilizada pelo protocolo BHMR é mais restritiva que a condição FDAS e Tsai, Kuo e Wang provaram que #-forçados(BHMR) $\leq$ #-forçados(FDAS) [45].

Intuitivamente, podemos pensar que se $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$, a seguinte relação seria sempre válida: #-forçados($\mathcal{C}_1$) $\leq$ #-forçados($\mathcal{C}_2$). Além disso, poderíamos pensar que um protocolo RDT que fosse implementado com a condição mais restritiva possível deveria induzir o número mínimo de *checkpoints* forçados para qualquer computação distribuída. No entanto, Tsai, Kuo e Wang provaram que estas duas suposições não são válidas.

Seja CP_n um protocolo que evita a formação de (i) todas as *PCM-paths* $\mu \cdot [m]$ de um processo p_a para um processo p_b, com $a \neq b$ e (ii) todas as *PCM-paths* $\mu \cdot [m]$ de um processo p_a para ele mesmo tais que a recepção de m precede o envio da primeira mensagem de μ. Seja CP_m um protocolo que evita a formação de (i) todas as *PCM-paths*

$\mu \cdot [m]$ de um processo p_a para um processo p_b, com $a \neq b$ e (ii) todas as *CM-paths* $\mu \cdot [m]$ de um processo p_a para ele mesmo tais que a recepção de m precede o envio da primeira mensagem de μ. Comparando-se a condição (ii) dos dois protocolos, podemos concluir que a condição CP_n é mais restritiva que a condição CP_m ($CP_n \Rightarrow CP_m$). No entanto, a Figura 34 mostra um cenário em que #-forçados(CP_n) > #-forçados(CP_m).



(a) CP_n (b) CP_m

Figura 34: $CP_n \Rightarrow CP_m$, mas #-forçados(CP_n) > #-forçados(CP_m)

Um protocolo RDT *ótimo* deve induzir o número mínimo de *checkpoints* forçados para qualquer computação distribuída. A Figura 35 reapresenta o cenário da Figura 34 para provar que a implementação de um protocolo RDT ótimo não é possível.

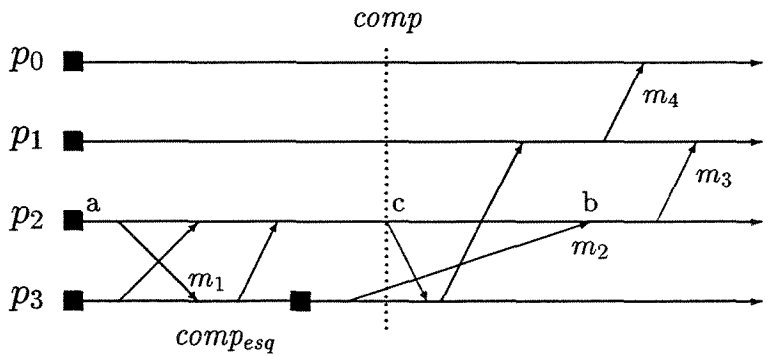


Figura 35: A implementação de protocolo RDT ótimo não é possível

Para a parte esquerda da computação ($comp_{esq}$), um protocolo ótimo não deveria induzir nenhum *checkpoint* forçado, como aconteceu com CP_n na Figura 34 (a). Para a computação toda (*comp*), um protocolo ótimo deveria induzir apenas um *checkpoint* forçado, como aconteceu com CP_m na Figura 34 (b).

No entanto, qualquer protocolo RDT ótimo para $comp_{esq}$, ou seja, que não induz um *checkpoint* forçado entre os pontos a e c de p_2, deve induzir um *checkpoint* forçado entre

c e b para evitar a formação do *Z-cycle* $[m_2, m_1]$. A presença de um *checkpoint* entre os pontos c e b induz p_1 a retirar um *checkpoint* forçado imediatamente antes de receber m_3 (Figura 34 (a)). Desta forma, um protocolo ótimo para $comp_{esq}$ não é ótimo para *comp* e um protocolo ótimo para *comp* não é ótimo para $comp_{esq}$.

5 Sumário

Iniciamos este Capítulo com a apresentação do modelo computacional, do conceito de precedência causal e da noção de cortes consistentes [16, 33]. Em seguida, definimos *checkpoints* globais consistentes e *zigzag paths* [38]. *Zigzag paths* são seqüências de mensagens que capturam a existência de dependências entre *checkpoints*. Existem dois tipos de *zigzag paths*: causais (*C-paths*) e não causais *(Z-paths)*. *C-paths* podem ser rastreadas em tempo de execução através da utilização de vetores de dependências ou relógios, mas *Z-paths* não podem ser rastreadas em tempo de execução.

Um grupo de *checkpoints* conectado por um *zigzag path* não pode fazer parte de um mesmo *checkpoint* global consistente. Além disso, um *zigzag path* que liga um *checkpoint* a ele mesmo forma um *Z-cycle* e determina um *checkpoint* inútil (que não não pode fazer parte de nenhum *checkpoint* global consistente). *Checkpoints* escolhidos aleatoriamente podem dar origem a *Z-cycles*. A abordagem síncrona evita a formação de *Z-cycles*, mas requer a propagação de mensagens de controle e pode acarretar a suspensão da execução da computação durante a retirada do *checkpoint* global [14, 32]. A abordagem quase-síncrona permite que os processos retirem *checkpoints* livremente, mas induz a retirada de *checkpoints* forçados para garantir a construção de *checkpoints* globais consistentes [17, 36].

Manivannan e Singhal dividiram os padrões de *checkpoint* gerados por protocolos quase-síncronos em quatro classes distintas: *Strictly Z-Path Free* (SZPF), *Z-Path Free* (ZPF), *Z-Cycle Free* (ZCF), e *Partially Z-Cycle Free* (PZCF) [36]. Protocolos PZCF não garantem a ausência total de *Z-cycles*, mas fazem um esforço para diminuir a sua ocorrência. Padrões ZCF garantem a ausência de *Z-cycles* e portanto a utilidade de todos os *checkpoints* do padrão. Padrões SZPF e ZPF obedecem à propriedade *Rollback-Dependency Trackability* (RDT), ou seja, todas as dependências entre *checkpoints* podem ser rastreadas em tempo de execução através da utilização de vetores de dependências ou vetores de relógios [50].

A propriedade RDT foi proposta por Wang utilizando *R-graphs* e vetores de dependências [50]. Baldoni, Helary e Raynal caracterizaram RDT a partir da duplicação causal de *Z-paths*. Eles conjeturaram que o conjunto formado por *EPSCM-paths* não duplicadas visivelmente determina o menor conjunto de Z-paths que devem ser evitadas em tempo de execução por um protocolo RDT [6]. Eles também afirmaram que um protocolo que

evita a formação dessas *Z-paths* precisaria da manutenção e propagação de informações de controle com complexidade $O(n^2)$, onde n é o número de processos na computação [7].

Tsai, Kuo e Wang analisaram o impacto teórico da utilização de condições mais restritas para a implementação de protocolos RDT [45]. Apesar de, na maioria das vezes, uma condição mais restritiva induzir sempre menos *checkpoints* forçados que uma condição menos restritiva, esta relação não é válida para todos os casos. Além disso, eles provaram que é impossível a implementação de um protocolo RDT ótimo, ou seja, que retire o menor número de *checkpoints* forçados para qualquer computação distribuída.

Capítulo 3

Progressive Construction of Consistent Global Checkpoints*

Islene Calciolari Garcia

Luiz Eduardo Buzato

Abstract

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

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

---

*Artigo publicado na *IEEE 19th International Conference on Distributed Computing Systems*, em Austin, Texas, Estados Unidos, em junho de 1999.

1 Introduction

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

A global checkpoint is a set of checkpoints, one per process. A global checkpoint is *consistent* if it could have been observed by an idealized external monitor [14]. The evaluation of global predicates [15]—the core of many monitors—is only meaningful when verified against consistent global checkpoints. Additionally, a *progressive view* of the computation may be required, in the sense that a monitor should construct a sequence of consistent global checkpoints that may have occurred in this order during the computation.

A checkpoint pattern may contain *useless* checkpoints, that is, checkpoints that cannot be part of any consistent global checkpoint [38]. Quasi-synchronous checkpointing protocols [36] allow processes to take checkpoints arbitrarily, but sometimes they are forced by the protocol to take additional checkpoints in order to reduce or eliminate the presence of useless checkpoints. The possibility of rollback recovery has been the motivation for the development of most of the quasi-synchronous checkpointing protocols [17]. When an error is detected, a procedure is triggered to rollback the application from its last global state to a consistent global checkpoint. Thus, we can identify two *orthogonal* problems: (i) the adequate selection of checkpoints, and (ii) the construction of consistent global checkpoints.

Our approach is to use quasi-synchronous checkpointing protocols to build a progressive view of a computation. In order to attain our goal, we have determined the Z-precedence between checkpoints: a checkpoint a must be observed before a checkpoint b in a progressive view if a Z-precedes b. This relationship is a generalization of Lamport's causal precedence [33] and is equivalent to zigzag paths proposed by Netzer and Xu [38], as defined by Manivannan, Netzer and Singhal [34]. Based on Z-precedence, we propose original algorithms for the progressive construction of consistent global checkpoints. We argue that Z-precedence can provide a better understanding of consistent global checkpoints due to its very simple, intuitive meaning.

The paper is structured as follows. Section 2 describes the computational model adopted. Section 3 explores the progressive view of a computation, introducing Z-precedence. Section 4 describes new algorithms to build consistent global checkpoints progressively. Section 5 discusses related work. Finally, Section 6 concludes the paper.

2 Model

A distributed application is composed of n sequential processes $(p_0, \ldots, p_{n-1})$ that communicate only by exchanging messages. Messages cannot be corrupted, but can be delivered out of order or lost. The activity of a process is modeled as a sequence of *events* that can be divided into internal events and communication events realized through the sending and the reception of messages. Checkpoints are internal events; each process takes an initial checkpoint (immediately after execution begins) and a final checkpoint (immediately before execution ends). Figure 1 illustrates a space-time diagram [33] augmented with checkpoints (black squares).

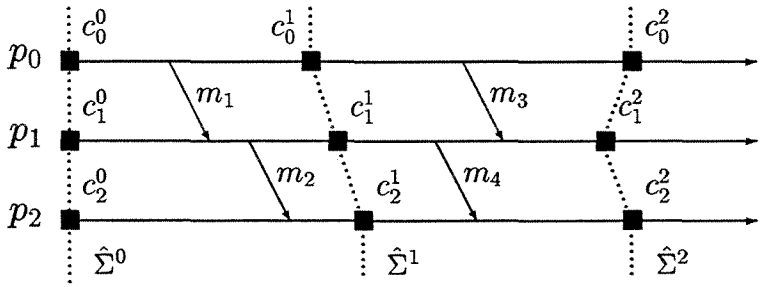


Figure 1: A distributed computation

Let c_i^γ denote the γth checkpoint taken by p_i. Considering Lamport's causal precedence [33] between events, let $c_a^\alpha \rightarrow c_b^\beta$ indicate that the event that generated c_a^α has causally preceded the event that generated c_b^β. In Figure 1, we can see that $c_0^0 \rightarrow c_1^1$ and $c_0^1 \rightarrow c_1^2$.

3 Progressive Views

A consistent global checkpoint is a set of checkpoints, one per process, that could have been observed by an idealized external monitor. Therefore, a consistent global checkpoint must contain only causally unrelated (concurrent) checkpoints [38].

Definition 3.1 Consistent Global Checkpoint—*A global checkpoint* $\hat{\Sigma} = \{c_0^{\iota_0}, \ldots, c_{n-1}^{\iota_{n-1}}\}$ *is consistent iff*

$$\forall i, j : 0 \leq i, j < n : c_i^{\iota_i} \not\rightarrow c_j^{\iota_j}$$

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

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

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

In Figure 1, the global checkpoints $\hat{\Sigma}^0$, $\hat{\Sigma}^1$, and $\hat{\Sigma}^2$ are examples of consistent global checkpoints and the sequence $(\hat{\Sigma}^0, \hat{\Sigma}^1, \hat{\Sigma}^2)$ forms a progressive view.

3.1 Z-Precedence Between Checkpoints

A precedence relation between a pair of checkpoints, say $c_a^\alpha \rightarrow c_b^\beta$, implies that they cannot be part of the same consistent global checkpoint. For example, consider checkpoints c_0^1 and c_1^2 (Figure 1). Message m_3 has been sent after c_0^1 and it has been received before c_1^2. Clearly, c_1^2 cannot be part of the same consistent global checkpoint with any checkpoint c_0^α such that $\alpha \leq 1$. Consequently, checkpoint c_0^1 must be *observed before* checkpoint c_1^2 in a progressive view.

Definition 3.3 Observed before—*A checkpoint* c_a^α *must be* observed before *a checkpoint* c_b^β *iff* c_b^β *cannot be part of the same consistent global checkpoint with* c_a^γ *such that* $\gamma \leq \alpha$.

Concurrent checkpoints may also have a well-defined order in a progressive view. Consider the concurrent checkpoints c_0^1 and c_2^2 in Figure 1. Due to m_4, c_1^1 must be observed before c_2^2. Thus, c_1^2 must be observed before or simultaneously to c_2^2. Due to m_3, c_0^1 must be observed before c_1^2. Thus, c_0^2 must be observed before or simultaneously to c_1^2. Consequently, checkpoint c_0^1 must be observed before c_2^2. Extending this scenario, we introduce Z-precedence between checkpoints. This relation indicates whether a checkpoint must be observed before another in a progressive view.

Definition 3.4 Z-precedence between checkpoints

Checkpoint c_a^α Z-precedes *checkpoint* c_b^β $(c_a^\alpha \rightsquigarrow c_b^\beta)$ *iff*

- $c_a^\alpha \rightarrow c_b^\beta$, *or*
- $\exists c_c^\gamma : (c_a^\alpha \rightsquigarrow c_c^\gamma) \wedge (c_c^{\gamma-1} \rightsquigarrow c_b^\beta)$.

Theorem 3.1 *If $c_a^\alpha \rightsquigarrow c_b^\beta$, c_a^α must be observed before c_b^β.*

Proof: The Z-precedence $c_a^\alpha \rightsquigarrow c_b^\beta$ can be expressed as a sequence of p causal precedence relations between pairs of checkpoints:

$$c_{i_0}^{\gamma_0-1} \rightarrow c_{i_1}^{\gamma_1}, c_{i_1}^{\gamma_1-1} \rightarrow c_{i_2}^{\gamma_2}, \ldots, c_{i_{p-1}}^{\gamma_{p-1}-1} \rightarrow c_{i_p}^{\gamma_p}$$

where $c_a^\alpha = c_{i_0}^{\gamma_0-1}$ and $c_{i_p}^{\gamma_p} = c_b^\beta$. By induction on p, we prove that c_a^α must be observed before c_b^β.

Base: $(p = 1)$ The causal precedence $c_a^\alpha \rightarrow c_b^\beta$ implies that c_a^α must be observed before c_b^β. Thus, $c_a^{\alpha+1}$ must be observed before or simultaneously to c_b^β.

Step: $(p > 1)$ Assume that $c_a^\alpha \rightsquigarrow c_{i_p}^{\gamma_p}$ implies that c_a^α must be observed before $c_{i_p}^{\gamma_p}$. The causal precedence $c_{i_p}^{\gamma_p-1} \rightarrow c_b^\beta$ implies that $c_{i_p}^{\gamma_p}$ must be observed before or simultaneously to c_b^β (Figure 2). Consequently, c_a^α must be observed before c_b^β. □

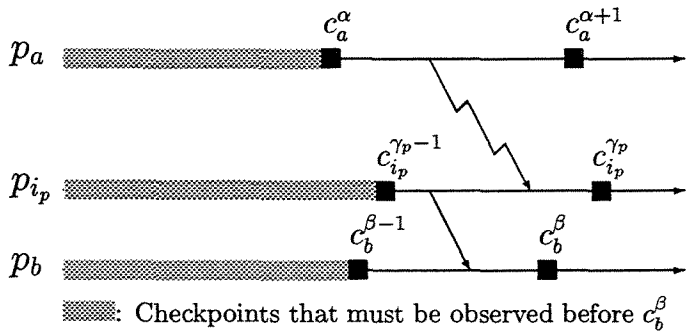


Figure 2: Induction step of Theorem 3.1

3.2 Useless Checkpoints

In this Section, we prove that a set of checkpoints unrelated by Z-precedence can be observed in a consistent global checkpoint. Let S be a set of checkpoints and let $S \not\rightsquigarrow S$ indicate that no checkpoint in S Z-precedes a checkpoint (including itself) in S. Given a single checkpoint c, let $c \not\rightsquigarrow S$ indicate that c does not Z-precede any checkpoint in S.

Theorem 3.2 *A set of checkpoints S can be part of the same consistent global checkpoint if $S \not\rightsquigarrow S$.*

Proof: We construct a consistent global checkpoint $\hat{\Sigma}$ that includes S taking, for every process p_j that does not have a checkpoint in S, a checkpoint $c_j^{\iota_j}$ such that

$$\iota_j = \min\{\gamma : c_j^\gamma \not\rightsquigarrow S\}$$

Note that if $\iota_j > 0$ there exists a checkpoint $c_i^{\iota_i}$ in S such that $c_j^{\iota_j - 1} \rightsquigarrow c_i^{\iota_i}$.

Assume that $\hat{\Sigma}$ is not consistent. There must exist a causal precedence between a pair of checkpoints in $\hat{\Sigma}$, say $c_a^\alpha \rightarrow c_b^\beta$ ($\beta > 0$). There are four possibilities of membership for these checkpoints:

- $c_a^\alpha \in S$, $c_b^\beta \in S$: Since $c_a^\alpha \rightsquigarrow c_b^\beta$, this violates the hypothesis that $S \not\rightsquigarrow S$.
- $c_a^\alpha \in \hat{\Sigma} \setminus S$, $c_b^\beta \in S$: The Z-precedence $c_a^\alpha \rightsquigarrow c_b^\beta$ violates the rule used to build $\hat{\Sigma}$.
- $c_a^\alpha \in S$, $c_b^\beta \in \hat{\Sigma} \setminus S$: There must exist $c_i^{\iota_i} \in S$ such that $c_b^{\beta-1} \rightsquigarrow c_i^{\iota_i}$. However, $c_a^\alpha \rightsquigarrow c_b^\beta$ and $c_b^{\beta-1} \rightsquigarrow c_i^{\iota_i}$ implies that $c_a^\alpha \rightsquigarrow c_i^{\iota_i}$, with $c_a^\alpha, c_i^{\iota_i} \in S$ and this also violates the hypothesis that $S \not\rightsquigarrow S$.
- $c_a^\alpha \in \hat{\Sigma} \setminus S$, $c_b^\beta \in \hat{\Sigma} \setminus S$: As in the previous case, there must exist $c_i^{\iota_i} \in S$ such that $c_b^{\beta-1} \rightsquigarrow c_i^{\iota_i}$. Since $c_a^\alpha \rightsquigarrow c_b^\beta$ and $c_b^{\beta-1} \rightsquigarrow c_i^{\iota_i}$ implies that $c_a^\alpha \rightsquigarrow c_i^{\iota_i}$, this violates the rule used to build $\hat{\Sigma}$. □

A checkpoint may have a Z-precedence to itself, called a Z-cycle. Using Theorem 3.1, we could conclude that such checkpoint must be observed before itself, what does not make sense. The conclusion to be drawn from Theorem 3.2 is that this checkpoint is *useless*: it cannot be part of any consistent global checkpoint. Therefore, the absence of a Z-precedence between a pair of checkpoints is a necessary and sufficient condition for them to be part of the same consistent global checkpoint. This result was first obtained by Netzer and Xu [38], our study demonstrates that Z-precedence is equivalent to their zigzag path abstraction (Section 5), as defined in [34].

Definition 3.5 Z-cycle—*A checkpoint c participates in a Z-cycle iff* $c \rightsquigarrow c$.

Corollary 3.3 *A checkpoint c that participates in a Z-cycle is a useless checkpoint.*

Useless checkpoints cannot be part of any progressive view. The progressive view $(\hat{\Sigma}^0, \hat{\Sigma}^1, \hat{\Sigma}^2)$ depicted in Figure 3 does not include the useless checkpoints c_2^1 and c_1^2.

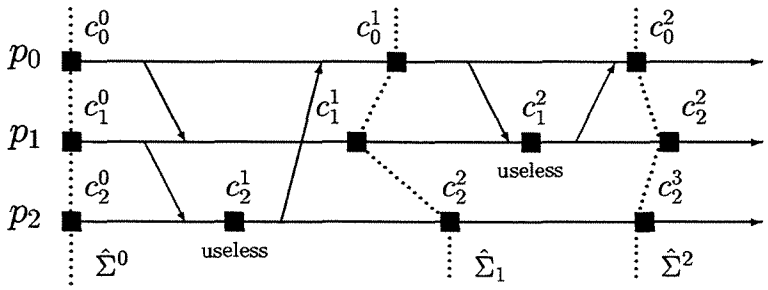


Figure 3: Useless checkpoints

3.3 A Step in a Progressive View

Assume that $\hat{\Sigma} = \{c_0^{\iota_0}, \ldots, c_{n-1}^{\iota_{n-1}}\}$ is a consistent global checkpoint of a progressive view $\mathcal{V}$ and we want to build $Next(\hat{\Sigma})$, the successor of $\hat{\Sigma}$ in $\mathcal{V}$. Let $c_i^{\iota_i+1}$ be the immediate successor of $c_i^{\iota_i}$, and let $S = \{c_0^{\iota_0+1}, \ldots, c_{n-1}^{\iota_{n-1}+1}\}$ be the set of immediate successors of the non-final checkpoints in $\hat{\Sigma}$. Intuitively, the set $Step(\hat{\Sigma}) \subseteq S$ of checkpoints that have no Z-predecessor in S (does not have any checkpoint in S that must be observed before it) can be replaced in $\hat{\Sigma}$ in order to build $Next(\hat{\Sigma})$.

Theorem 3.4 *Let $\hat{\Sigma} = \{c_0^{\iota_0}, \ldots, c_{n-1}^{\iota_{n-1}}\}$ be a consistent global checkpoint and let its succeeding checkpoints form the set $S = \{c_0^{\iota_0+1}, \ldots, c_{n-1}^{\iota_{n-1}+1}\}$. We define*

$$Step(\hat{\Sigma}) = \{c \in S :: \nexists c' \in S : c' \rightsquigarrow c\} \quad (*)$$

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

Proof: Assume that $Next(\hat{\Sigma})$ is inconsistent. Thus, there must exist a pair of causally related checkpoints in $Next(\hat{\Sigma})$, say $c_a^\alpha \rightarrow c_b^\beta$. There are four possibilities of membership for c_a^α and c_b^β:

- $c_a^\alpha \in \hat{\Sigma}$, $c_b^\beta \in \hat{\Sigma}$—Violates the hypothesis that $\hat{\Sigma}$ is a consistent global checkpoint.
- $c_a^\alpha \in \hat{\Sigma}$, $c_b^\beta \in Step(\hat{\Sigma})$—Since $c_a^{\alpha+1}$ does not belong to $Step(\hat{\Sigma})$, there must exist a checkpoint $c_i^{\iota_i+1}$ in S such that $c_i^{\iota_i+1} \rightsquigarrow c_a^{\alpha+1}$. The concatenation of $c_i^{\iota_i+1} \rightsquigarrow c_a^{\alpha+1}$ and $c_a^\alpha \rightarrow c_b^\beta$ forms a Z-precedence $c_i^{\iota_i+1} \rightsquigarrow c_b^\beta$ that violates rule $(*)$ used to build $Step(\hat{\Sigma})$.
- $c_a^\alpha \in Step(\hat{\Sigma})$, $c_b^\beta \in \hat{\Sigma}$—Violates the hypothesis that $\hat{\Sigma}$ is a consistent global checkpoint.
- $c_a^\alpha \in Step(\hat{\Sigma})$, $c_b^\beta \in Step(\hat{\Sigma})$—Violates rule $(*)$ used to form $Step(\hat{\Sigma})$. □

Unfortunately, $Step(\hat{\Sigma})$ can be empty. When this happens, for all checkpoints $c_i^{\iota_i+1}$ in S we can choose another checkpoint $c_j^{\iota_j+1}$ in S such that $c_j^{\iota_j+1} \rightsquigarrow c_i^{\iota_i+1}$. Since the number of checkpoints in S is finite, we have a Z-cycle. In order to build $Next(\hat{\Sigma})$ the checkpoints that participate in Z-cycles must be discarded and the immediate successors of these checkpoints must be considered.

4 Algorithms

In this Section, we introduce algorithms to build progressive views of distributed computations. The algorithms are presented in Java*, because it is the language we adopted for our implementation of the monitor. Additionally, Java [26, 44] is easy to read and has a precise specification. The details of each algorithm are dictated by the checkpoint pattern of the computation. We introduce a classification of checkpoint patterns that is based on the one proposed by Manivannan and Singhal [36].

- **Z-Precedence Free (ZPF) Pattern**: For every pair of checkpoints in this pattern, say c_a^α and c_b^β, the following condition holds: $(c_a^\alpha \rightarrow c_b^\beta) \iff (c_a^\alpha \rightsquigarrow c_b^\beta)$. In other words, all Z-precedences are causal precedences. Examples of protocols that generate this pattern are described in [3, 31, 42, 50].
- **Z-Cycle Free (ZCF) Pattern**: In this pattern, checkpoints do not participate in Z-cycles; it contains only useful checkpoints. Examples of protocols that generate this pattern are described in [8, 9, 11, 28].
- **Partially Z-Cycle Free (PZCF) Pattern**: In this pattern, checkpoints may participate in Z-cycles; it may contain useless checkpoints. Examples of protocols that generate this pattern are described in [52, 55].

Processes behavior: We assume that processes maintain and propagate vector clocks [37], as described by class `Process` (Class 1). Vectors clocks are used to characterize causal precedence among checkpoints. When a message is sent, the vector clock of the sender is piggybacked onto it. Before consuming a message, each process takes a component-wise maximum of its vector clock and the received vector clock. When a process takes a checkpoint, it increments its corresponding entry in the vector clock. A checkpoint is described by class `VC_Ckpt` (Class 2).

General structure of the algorithms: For each pattern, we define a method called `next`, whose function is to allow the monitor to move forward in its progressive view of the application. The monitor maintains variables `C` and `S` to implement the sets $\hat{\Sigma}$ and S, respectively, as in Theorem 3.4. It also maintains an auxiliary vector `M` to mark processes whose checkpoints in `S` are Z-preceded by other checkpoints in `S`. The checkpoints in `S` taken by the unmarked processes can be substituted in `C`. Useless checkpoints should be discarded. For simplicity, we assume that when `next` is called, `S` is complete.

*Java is a trademark of Sun Microsystems, Inc.

Class 1 Process.java

```
public class Process {

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

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

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

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

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

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

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

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

Class 2 VC_Ckpt.java

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

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

Progressive view in a ZPF pattern: In this pattern, $Step(\hat{\Sigma})$ can be simply determined by checkpoints in S that are not causally preceded by other checkpoints in S. Class `ZPF_Pattern` (Class 3) describes an implementation of `next` for this pattern. Figure 4 illustrates a result of its execution: p_2 has been marked by the algorithm because $c_2^{\iota_2+1}$ is causally preceded by $c_1^{\iota_1+1}$ and p_3 has been marked because $c_3^{\iota_3+1}$ is causally preceded by $c_0^{\iota_0+1}$. Thus, $Step(\hat{\Sigma})$ is formed by $\{c_0^{\iota_0+1}, c_1^{\iota_1+1}\}$ and $Next(\hat{\Sigma}) = \{c_0^{\iota_0+1}, c_1^{\iota_1+1}, c_2^{\iota_2}, c_3^{\iota_3}\}$.

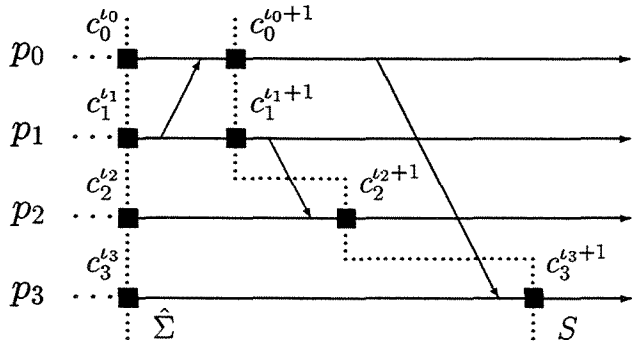


Figure 4: Progressive view in a ZPF pattern

Class 3 ZPF_Pattern.java

```
public class ZPF_Pattern {

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

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

    public void next() {

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

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

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

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

Since this pattern does not admit Z-cycles, $Step(\hat{\Sigma})$ is guaranteed to be non-empty (Section 3.2). This result is important not only because it guarantees that ZCF protocols originally developed for backward error recovery can also be used for monitoring. They can also be explored in the context of dependable systems where the seamless integration of error recovery and monitoring can be seen as a basic requirement.

Class `ZCF_Pattern` (Class 4) describes an implementation of method `next` for this pattern. In Figure 5, consider the Z-precedences within S: (i) $c_0^{\iota_0+1} \rightsquigarrow c_3^{\iota_3+1}$ and (ii) $c_0^{\iota_0+1} \rightsquigarrow c_2^{\iota_2+1}$. At this level of abstraction, (i) and (ii) are sufficient to compute $Step(\hat{\Sigma})$ and $Next(\hat{\Sigma})$. In contrast, the sequence of causal precedences $c_0^{\iota_0+1} \rightarrow c_3^{\iota_3+1}$ and $c_3^{\iota_3} \rightarrow c_2^{\iota_2+1}$ must be considered to determine that $c_3^{\iota_3+1}$ and $c_2^{\iota_2+1}$ cannot be part of $Step(\hat{\Sigma})$.

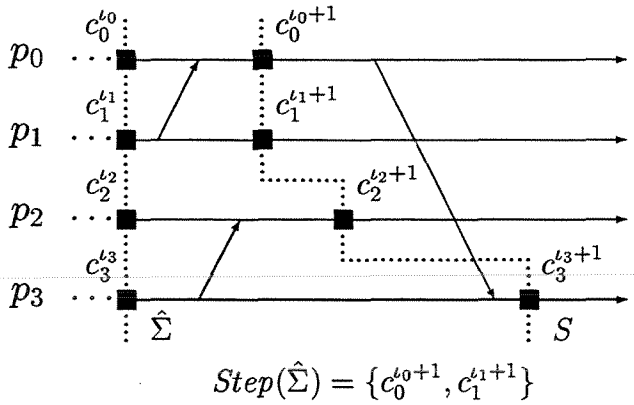


Figure 5: Progressive view in a ZCF pattern

Class 4 ZCF_Pattern.java

```
public class ZCF_Pattern {

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

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

    public void next() {

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

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

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

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

Progressive view in a PZCF pattern: In this pattern, it is necessary to know if a checkpoint Z-precedes itself: this information is necessary to determine useless checkpoints. An auxiliary matrix, called `Z`, is used. An entry `Z[i,j]` indicates that $c_i^{\iota_i+1} \rightsquigarrow c_j^{\iota_j+1}$ has been detected, considering the information in $\hat{\Sigma}$ and S. If `Z[i,i]` is true, $c_i^{\iota_i+1}$ is a useless checkpoint. Class `PZCF_Pattern` (Class 5) describes an implementation of method `next`, that is similar to the implementation `next` for the `ZCF_Pattern`, it only adds the computation of `Z`. This algorithm is guaranteed to progress, because if $Step(\hat{\Sigma})$ is empty, at least one useless checkpoint will be identified. Figure 6 illustrates a result of its execution in which $Step(\hat{\Sigma})$ is empty and checkpoints $c_0^{\iota_0+1}$ and $c_1^{\iota_1+1}$ are useless. To verify why $c_0^{\iota_0+1}$ and $c_1^{\iota_1+1}$ are useless we can iterate through the algorithm. The first iteration detects the Z-cycle: $c_0^{\iota_0+1} \rightarrow c_3^{\iota_3+1}$, $c_3^{\iota_3} \rightarrow c_1^{\iota_1+1}$, and $c_1^{\iota_1} \rightarrow c_0^{\iota_0+1}$, discarding $c_0^{\iota_0+1}$. The second iteration detects the Z-cycle: $c_1^{\iota_1+1} \rightarrow c_0^{\iota_0+2}$, $c_0^{\iota_0} \rightarrow c_3^{\iota_3+1}$, $c_3^{\iota_3} \rightarrow c_1^{\iota_1+1}$, discarding $c_1^{\iota_1+1}$. Finally, the monitor is able to construct $\hat{\Sigma}' = \{c_0^{\iota_0+2}, c_1^{\iota_1+2}, c_2^{\iota_2+1}, c_3^{\iota_3+1}\}$.

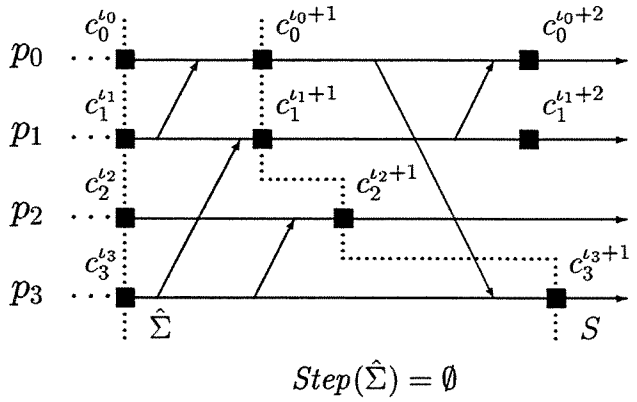


Figure 6: Progressive view in a PZCF pattern

Optimization: Monitoring of distributed computations make it interesting to construct consistent global checkpoints as soon as possible. In this case, the computation of the consistent global checkpoint can be triggered even when S is incomplete. For example, consider that checkpoint $c_j^{\iota_j+1}$ is not available to the monitor. Therefore, it must consider p_j as marked, and it must mark processes that have checkpoints in S Z-preceded by $c_j^{\iota_j}$. Also, for every available checkpoint $c_i^{\iota_i+1}$, the precedence test `S[j].precedes(S[i])` that evaluates whether $VC(S[j])[j] \leq VC(S[i])[j]$ must be substituted for one that evaluates whether $VC(C[j])[j] < VC(S[i])[j]$. In this case, however, the algorithm is not guaranteed to make progress (construct a new consistent global checkpoint or discard useless checkpoints).

Class 5 PZCF_Pattern.java

```
public class PZCF_Pattern {

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

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

    public void next() {

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

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

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

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

Another optimization would be the use of direct dependency tracking [53] instead of transitive dependency. This approach would greatly reduce the amount of information maintained by the processes and propagated to the monitor.

5 Related Work

This paper presents algorithms to build a progressive view of a distributed computation. Given its checkpoint pattern and precedence information about its checkpoints, it is possible to build a progressive view for it. We should note, however, that the complexity of the algorithm to be used is dependent on the class of the checkpoint pattern (Section 4).

There are other alternatives to build a progressive view of a distributed computation. If a synchronous algorithm [14] is called sequentially, the consistent global checkpoints obtained will form a progressive view. However, this approach is intrusive. Another approach is to use a quasi-synchronous checkpointing protocol that builds recovery lines with increasing index numbers. The well-known protocol proposed by Briatico, Ciuffoletti and Simoncini [11], and some variants of it [9, 28, 35] have this characteristic. Taking a different path, our result is more general because it does not have to take into account the strategy used for selecting checkpoints to build the progressive view.

Our algorithms were constructed considering Z-precedence, that establishes whether a checkpoint must be observed before another in a progressive view. This relationship is equivalent to zigzag paths proposed by Netzer and Xu [38], but represents a higher level abstraction. Zigzag paths [38] are defined on a sequence of *messages causally connected or not* while Z-precedence is based *only* on checkpoints and causal precedence between checkpoints. This is consistent with the abstraction level we are working with: events, including messages, belong to a lower level of abstraction. Finally, there exist other works [50, 4, 29] that introduce abstractions similar to zigzag paths and Z-precedence, but their abstractions consider checkpoint intervals.

The application of Z-precedence to alternative computational models of distributed systems helped to obtain positive evidences of its usefulness. Using it, we have been able to derive a protocol to asynchronously construct a progressive view of a computation in the object and action model [19]. In this model, distributed atomic actions are used to organize the flux of method invocations on the set of objects that form the distributed application [54]. An early account of our work within this model can be found in [20]; it represents an advance in relation to previous work by Fischer, Griffeth and Lynch [18]. Similarly, the work of Baldoni, Helary and Raynal [4] represents a step forward towards the construction of protocols to obtain consistent global states that are valid in different computational models. Their work resulted in the definition a computational model that includes the shared memory model and several specializations of the message passing models.

6 Conclusion

Algorithms for obtaining consistent global checkpoints are an useful aid in a vast class of problems that can be formulated as the evaluation of a predicate over the global state of an application [15]. In this paper we have introduced the notion of a *progressive view* of a computation: a sequence of consistent global checkpoints that may have occurred in this order during the execution of an application. We have also proposed original algorithms to construct such a view. We believe that monitoring and dynamic reconfiguration of

distributed systems can benefit from such algorithms to guarantee that reconfiguration occurs adequately.

During the development of our algorithms we have determined the exact conditions under which a checkpoint must be observed before another in a progressive view. *Z-precedence* is a generalization of Lamport's causal precedence [33], and it is equivalent to zigzag paths proposed by Netzer and Xu [38]. Besides its technical advantages, we believe that the intuitive meaning of Z-precedence can provide a better understanding of consistent global checkpoints.

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

Acknowledgment

We would like to thank Alexandre Oliva for his help with the paper, and Gustavo Maciel Dias Vieira for helping with the implementation and testing of our algorithms.

Capítulo 4

Monitorização e Recuperação por Retrocesso Utilizando Visões Progressivas de Computações Distribuídas*

Islene Calciolari Garcia

Luiz Eduardo Buzato

Resumo

Sistemas distribuídos tolerantes a falhas baseiam-se parcialmente na existência de mecanismos de *checkpointing* e recuperação por retrocesso de estado. Um outro mecanismo importante para esses sistemas é o que permite a *monitorização* do seu estado e a pronta reação a mudanças que afetam o seu funcionamento previsto. Apesar desses dois mecanismos estarem fortemente associados, a grande maioria dos sistemas tolerantes a falhas contruídos até hoje privilegia a implementação de mecanismos de *checkpointing*, em detrimento de mecanismos para monitorização. Neste artigo, comentamos a utilidade de visões progressivas (seqüências de *checkpoints* globais consistentes que poderiam ter ocorrido nesta ordem durante a execução de uma computação) para o desenvolvimento de protocolos mais eficientes para *checkpointing* e para a integração de mecanismos de *checkpointing*, recuperação de estado e monitorização.

Abstract

Fault-tolerant distributed systems are partially based on the implementation of checkpointing and rollback-recovery mechanisms. Another important mechanism associated to tolerance of faults is the one that allows a system to monitor its state and to react to exceptional behaviour. Checkpointing and rollback-recovery are already part of most of the fault-tolerant systems implemented to date. This situation does not apply to monitoring mechanisms, especially to systems that integrate monitoring, checkpointing and rollback-recovery. This article dicusses the application of progressive views (sequences of consistent global checkpoints that may have occurred in this order during the execution of the system) towards the deployment of more efficient checkpointing mechanisms and its application to the integration of checkpointing, rollback-recovery and monitoring for fault-tolerant distributed systems.

*Este resumo foi publicado no VIII Simpósio de Computação Tolerante a Falhas, que ocorreu em Campinas, São Paulo, em julho de 1999.

1 Introdução

Vários mecanismos que permitem a adição de tolerância a falhas a um sistema distribuído estão baseados na seleção de estados (*checkpoints*) dos componentes deste sistema. A avaliação de predicados sobre estados globais consistentes [14] pode ser utilizada na construção de algoritmos distribuídos para detecção de *deadlocks*, reconstrução de ficha perdida, coleta de lixo e reconfiguração de componentes [16]. A recuperação de falhas por retrocesso de estado permite que a aplicação possa restaurar um estado global consistente após a ocorrência de uma falha, de maneira a diminuir a quantidade de trabalho perdido [41]. Estes mecanimos podem se beneficiar da obtenção de visões progressivas de computações distribuídas, ou seja, seqüências de *checkpoints* globais consistentes que poderiam ter ocorrido nesta ordem durante a execução das computações [19, 21, 22].

Recentemente, obtivemos três resultados nesta área: (i) o estabelecimento de uma relação entre *checkpoints* que determina uma ordem para a observação destes *checkpoints* [19, 22], (ii) a proposta de algoritmos originais para a construção de visões progressivas a partir de *checkpoints* [19, 22] e (iii) o mapeamento destes resultados para o modelo de objetos e ações atômicas [19, 20]. Neste artigo, vamos comentar a utilidade do conceito de visões progressivas para o desenvolvimento de protocolos mais eficientes para *checkpointing* e para a implementação de mecanismos para tolerância a falhas, incluindo a integração de mecanismos de monitorização e de recuperação de falhas por retrocesso de estado.

Este artigo está estruturado da seguinte forma. A Seção 2 comenta abordagens para a seleção de *checkpoints*. A Seção 3 introduz visões progressivas de computações distribuídas. A Seção 4 apresenta duas arquiteturas de *software* para tolerância a falhas que se beneficiam da construção de visões progressivas. A Seção 5 encerra o artigo.

2 Protocolos para *Checkpointing*

Um estado global de um aplicação distribuída é formado pela união dos estados locais (*checkpoints*) dos componentes desta aplicação. Informalmente, um *checkpoint* global é *consistente* se corresponde a um estado global que poderia ter sido obtido por um observador onisciente externo [14]. Protocolos para *checkpointing* [17, 36] podem ser classificados segundo três abordagens [36]: (i) assíncrona, (ii) síncrona e (iii) quase-síncrona.

A primeira abordagem oferece autonomia máxima aos componentes de uma aplicação para a seleção de *checkpoints*, mas podem ser selecionados *checkpoints inúteis* (*checkpoints* que não podem fazer parte de nenhum *checkpoint* global consistente) [38]. A abordagem síncrona garante a utilidade de todos os *checkpoints*, mas restringe a escolha de

checkpoints, apresenta um custo extra de comunicação e pode acarretar a suspensão das atividades dos componentes durante a sincronização [14]. A abordagem quase-síncrona está baseada em um protocolo obedecido pelos componentes da aplicação para a seleção de *checkpoints*. Prioritariamente, os componentes selecionam *checkpoints* livremente, mas eventualmente podem ser induzidos a selecionar *checkpoints* adicionais de acordo com informações de controle propagadas através das mensagens da aplicação [17, 36].

A abordagem quase-síncrona apresenta um compromisso entre a autonomia dos componentes para a escolha dos *checkpoints* e as garantias oferecidas para a formação de estados globais consistentes a partir dos *checkpoints* selecionados. Esta abordagem é apropriada para a implementação de mecanismos para tolerância a falhas, como monitorização de aplicações distribuídas e recuperação de falhas por retrocesso de estado. No entanto, determinar qual protocolo quase-síncrono deve ser adotado não é uma tarefa simples, pois requer uma análise de custo/benefício que envolve vários fatores.

Existe uma distinção entre os *checkpoints básicos* (selecionados espontaneamente pela aplicação) e os *checkpoints* induzidos pelo protocolo. Em príncipio, um protocolo ideal deveria induzir o menor número possível de *checkpoints* e ainda assim garantir a utilidade de todos os *checkpoints* básicos. Infelizmente, a especificação deste protocolo parece ser impossível, pois dependeria de conhecimento sobre o futuro da execução da aplicação [36]. Cabe aos projetistas considerar um compromisso entre número de *checkpoints* induzidos, complexidade da informação de controle e possibilidade de *checkpoints* inúteis.

3 Visões Progressivas de Computações Distibuídas

Uma visão progressiva de uma computação distribuída é formada por uma seqüência de (*checkpoints*) globais consistentes que poderia ter ocorrido nesta ordem durante a computação [22]. Em contraste, as abordagens exploradas tradicionalmente pelos algoritmos existentes na literatura fazem uso de uma das seguintes técnicas: (i) computam um *checkpoint* global consistente utilizando um conjunto inicial de *checkpoints* e o transformam em um *checkpoint* global consistente através da inclusão de *checkpoints* de outros processos [36] ou (ii) computam um *checkpoint* global consistente retrocedendo de um estado global inconsistente [17]. Acreditamos que a abordagem progressiva pode contribuir para um melhor entendimento e para a obtenção de soluções melhores para problemas ligados a estados globais consistentes e protocolos quase-síncronos para *checkpointing*.

Z-Precedência entre *Checkpoints* Um dos resultados teóricos mais importantes na área de *checkpointing* foi a determinação por Netzer e Xu das condições necessárias e suficientes para a participação de um *checkpoint* em um *checkpoint* global consistente [38].

Obtivemos uma re-interpretação deste resultado com a determinação da *Z-precedência* entre *checkpoints* [22]. Esta relação determina uma ordem para a observação de *checkpoints* em visões progressivas e é aplicável a vários modelos computacionais. Em particular, aplicamos este conceito ao modelo de objetos e ações atômicas [54], estendendo nosso trabalho de construção assíncrona de estados globais consistentes neste modelo [19, 20].

Desenvolvimento e Avaliação de Protocolos Existem três abordagens na literatura para se garantir que um protocolo não gera *checkpoints inúteis*: (i) demonstrar a possibilidade de se construir um *checkpoint* global consistente a partir de qualquer *checkpoint* do padrão [2], (ii) garantir que o padrão de *checkpoints* gerado possui a propriedade de *precedência virtual* [29] ou (iii) garantir que determinados padrões de comunicação que poderiam ocasionar *checkpoints* inúteis não ocorrem [8].

Podemos provar que um protocolo não gera *checkpoints* inúteis garantindo que dado um *checkpoint* global consistente e os *checkpoints* locais que o sucedem imediatamente, é sempre possível construir um outro *checkpoint* global consistente sem descartes de *checkpoints* inúteis [19]. Este cenário engloba características das três abordagens anteriores, contribuindo para a proposta de novos protocolos e para o estabelecimento de um arcabouço único para avaliação dos protocolos quase-síncronos presentes na literatura.

Avaliação de Predicados Instáveis Predicados instáveis, de maneira geral, só podem ser avaliados na presença de um *reticulado* da computação que contém todos os estados globais pelos quais a computação pode ter passado [16]. Acreditamos ser possível a modificação dos nossos algoritmos para a construção de visões progressivas para que se possa obter um reticulado formado apenas por *checkpoints* globais consistentes. Com a colaboração dos componentes da aplicação para a seleção de *checkpoints* adequados, este reticulado poderá ser útil para avaliação eficiente de predicados pertencentes a algumas classes específicas de predicados instáveis. Uma classe de provável aplicação seria a de predicados globais instáveis formados pela conjunção de predicados locais.

4 Monitorização e Recuperação de Falhas

Monitorização utilizando Visão Progressiva Consideramos que um *programa distribuído* é composto pela superposição de dois sistemas reativos: (i) o *programa da aplicação*, que implementa os aspectos funcionais do programa distribuído e (ii) o *programa de controle*, que implementa os aspectos de gerência e reconfiguração. O programa de controle, por sua vez, também está dividido em dois módulos (Figura 1): (i) o *fotógrafo*,

responsável pela construção progressiva de *checkpoints* globais consistentes [22] e (ii) o *monitor*, responsável pela avaliação de predicados globais [16] e atuação sobre a aplicação. Os componentes do programa da aplicação podem cooperar com o programa de controle através da seleção de *checkpoints*. Um protótipo desta arquitetura para o modelo de processos e mensagens foi implementado em Java^{TM} por Gustavo Maciel Dias Vieira [49].

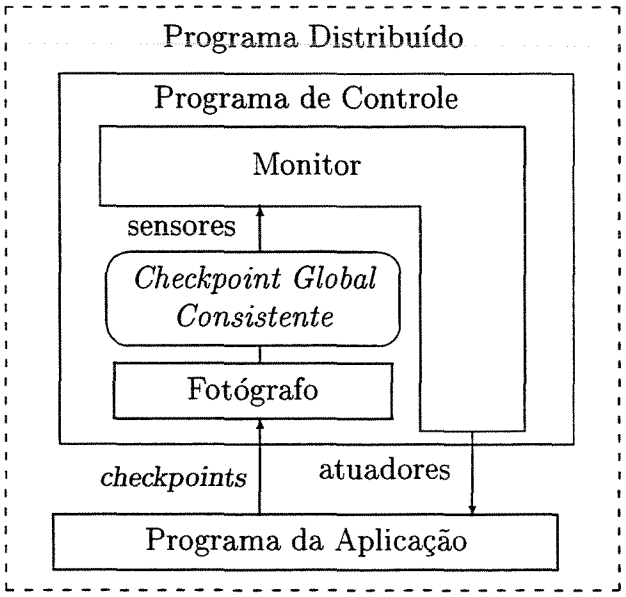


Figura 1: Arquitetura de *software* para monitorização utilizando visão progressiva

Integração de Recuperação de Erros e Monitorização A Figura 2 apresenta uma arquitetura de *software* que ilustra a integração entre os mecanismos. O fotógrafo constrói *checkpoints* globais consistentes e os envia para o programa de controle, que é composto pelos sistemas de monitorização e reconfiguração. Quando o sistema de recuperação é notificado de alguma falha, ele propaga as linhas de recuperação para que os componentes da aplicação possam fazer o retrocesso. Linhas de recuperação também são propagadas periodicamente para que possa ser feita a coleta de lixo de *checkpoints* obsoletos.

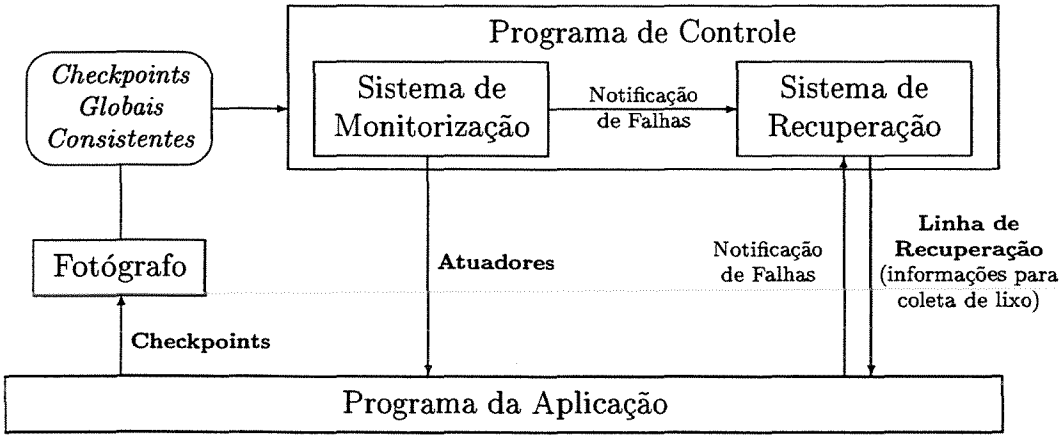


Figura 2: Integração entre monitorização e recuperação com retrocesso.

5 Conclusão

Uma visão progressiva de uma computação distribuída é formada por uma seqüência de fotografias do estado da aplicação que poderia ter ocorrido nesta ordem durante a computação. Visões progressivas podem ser construídas a partir de *checkpoints* (estados selecionados pelos componentes da aplicação), formando uma abstração da computação executada. Argumentamos que o conceito de visões progressivas pode permitir uma avaliação uniforme dos protocolos existentes e contribuir para o desenvolvimento de novos protocolos. Visões progressivas também podem contribuir para a implementação de mecanismos para tolerância a falhas. Uma das nossas propostas é integrar sistemas de monitorização e sistemas de recuperação de falhas por retrocesso de estado, aproveitando as similaridades entre estes dois sistemas de modo a oferecer uma solução única, mais uniforme e eficiente.

Agradecimentos

Nossos especias agradecimentos a Gustavo Maciel Dias Vieira por ter implementado e testado nossas idéias e por ter contribuído com vários comentários interessantes.

Capítulo 5

Using Common Knowledge to Improve Fixed-Dependency-After-Send*

Islene Calciolari Garcia

Luiz Eduardo Buzato

Abstract

Checkpoint patterns that enforce the rollback-dependency trackability (RDT) property allow efficient solutions to the determination of consistent global checkpoints that include a given set of checkpoints. Fixed-Dependency-After-Send (FDAS) is a well-known RDT protocol that forces the dependency vector of a process to remain unchanged during a checkpoint interval after the first message-send event. In this paper, we explore processes' common knowledge about their behavior to derive a more efficient condition to induce checkpoints under FDAS. We consider that our approach can be used to improve other RDT checkpointing protocols.

Keywords: Distributed systems; Fault-tolerance; Distributed checkpointing; Rollback-dependency trackability

*Este artigo foi publicado no *II Workshop de Testes e Tolerância a Falhas*, em Curitiba, Paraná, em julho de 2000.

1 Introduction

A checkpoint is a stable memory record of a process' state. A consistent global checkpoint is a set of checkpoints, one per process, that could have been seen by an idealized observer external to the computation [14]. The set of all checkpoints taken by a distributed computation forms a checkpoint pattern. Checkpoint patterns that enforce the rollback-dependency trackability (RDT) property allow efficient solutions to the determination of the maximum and minimum consistent global checkpoints that include a set of checkpoints [50]. Many applications can benefit from these algorithms: rollback recovery, software error recovery, deadlock recovery, mobile computing and distributed debugging [50].

In order to enforce the RDT property, an RDT checkpointing protocol [3, 50] allows processes to take checkpoints asynchronously (basic checkpoints), but they may be induced by the protocol to take additional checkpoints (forced checkpoints). Fixed-Dependency-After-Send (FDAS) is a well-known RDT protocol that forces the dependency vector of a process to remain unchanged during a checkpoint interval after the first message-send event [50]. Upon the reception of a message, a process must take a forced checkpoint if an entry of its dependency vector is about to change [3, 17, 36, 50].

The usual approach to implement FDAS does not take advantage of the processes' common knowledge about their behavior. Consider a scenario where a process p_i receives a message from p_j sent during an interval already known by p_i. Since p_i knows that p_j cannot increase its dependency vector after a send, it can skip the verification of checkpoint dependencies. This simple observation reduces the complexity of the decision to take a forced checkpoint from $O(n)$ to $O(1)$, where n is the number of processes in the computation.

The paper is structured as follows. Section 2 describes the computational model adopted. Section 3 introduces rollback-dependency trackability. Section 4 describes Fixed-Dependency-After-Send. Section 5 presents and proves the correction of the proposed optimization. Section 6 summarizes the paper.

2 Computational Model

A distributed computation is composed of n sequential processes $(p_0, \ldots, p_{n-1})$ that communicate only by exchanging messages. Messages cannot be corrupted, but can be delivered out of order or lost. The activity of a process is modeled as a sequence of *events* that can be divided into internal events and communication events realized through the sending and the reception of messages. Checkpoints are internal events; each process takes an initial checkpoint (immediately after execution begins) and a final checkpoint (imme-

diately before execution ends). Figure 1 illustrates a space-time diagram [33] augmented with checkpoints (black squares).

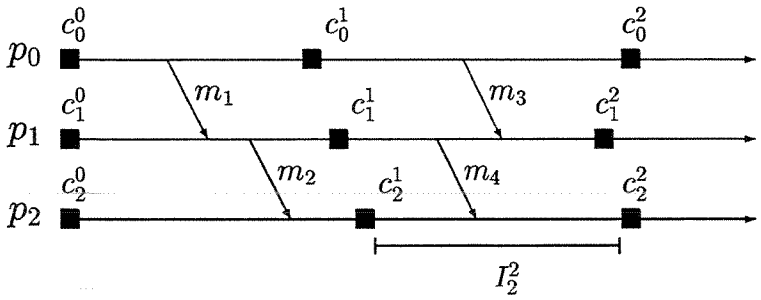


Figure 1: A distributed computation

Let c_i^γ denote the γth checkpoint taken by p_i. Checkpoint $c_i^{\gamma-1}$, $\gamma > 0$, and its immediate successor c_i^γ define a checkpoint interval I_i^γ. This interval represents the set of events produced by p_i between $c_i^{\gamma-1}$ and c_i^γ.

3 Rollback-Dependency Trackability

This Section introduces the concept of rollback-dependency trackability as defined by Wang [50], beginning with the definition of an R-graph, a digraph used to capture dependencies among checkpoints [50].

Definition 3.1 R-graph—*In an R-graph, each node represents a checkpoint and a directed edge is drawn from c_a^α to c_b^β if (i) $a = b$ and $\beta = \alpha + 1$ or (ii) $a \neq b$, and a message m is sent in I_a^α and received in I_b^β.*

Figure 2 shows the R-graph correspondent to the distributed computation depicted in Figure 1. The name R-graph (rollback-dependency graph) comes from the observation that if there is a path in the R-graph from c_a^α to c_b^β and I_a^α is rolled back, I_b^β must also be rolled back [50].

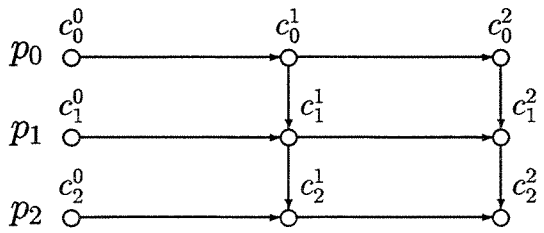


Figure 2: R-graph

Dependency Vector A dependency tracking mechanism can be used to capture causal dependencies among checkpoints. Each process p_i maintains and propagates a size-n dependency vector dv_i, that is initially $(0, \ldots, 0)$. The entry $dv_i[i]$ represents the current interval of p_i and it is incremented immediately after a new checkpoint is taken. Every other entry $dv_i[j]$, $j \neq i$, represents the highest interval index of p_j upon which p_i is dependent; it is updated every time a message m with a greater value of $dv_m[j]$ arrives at p_i.

Figure 3 depicts the dependency vectors associated to checkpoints of the distributed computation presented in Figure 1. Note that the dependency vector associated to checkpoint c_2^1 is (1, 1, 1) and it correctly represents the nodes that can reach this checkpoint in the R-graph (Figure 2). Unfortunately, not all checkpoint dependencies can be tracked on-line. For example, the dependency vector associated to checkpoint c_2^2 does not capture that c_0^2 can reach c_2^2 in the R-graph, since the edge from c_0^2 to c_1^2 was established after m_4 was sent.

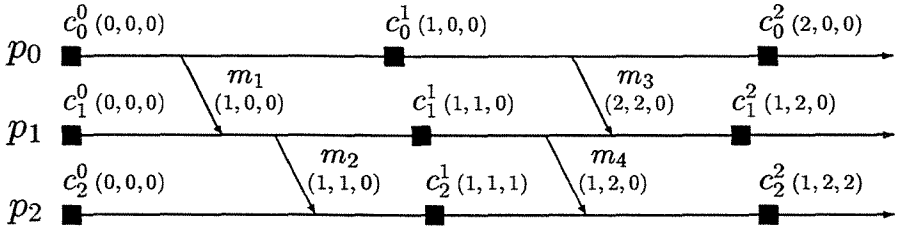


Figure 3: A distributed computation with dependency vectors

Rollback-Dependency Trackability Wang established a property in a checkpoint pattern that allows dependency vectors to carry all information needed to perform reachability analysis in its correspondent R-graph [50].

Definition 3.2 Rollback-Dependency Trackability—*A checkpoint pattern satisfies rollback-dependency trackability if the following property holds:*

> *For any two checkpoints $c_a^\alpha (\alpha \neq 0)$ and c_b^β of the pattern, there is a path from c_a^α to c_b^β in the R-graph if, and only if, $dv(c_b^\beta)[a] \geq \alpha$.*

4 Fixed-Dependency-After-Send

One way to enforce RDT is to consider Fixed-Dependency-After-Send (FDAS): in any interval, after the first message has been sent, the dependency vector remains unchanged until the next checkpoint [50]. Thus, upon the reception of a message, a forced checkpoint is induced if any entry of the dependency vector is about to be changed. The descriptions of FDAS presented in the literature always compare all entries of the dependency vector to induce a forced checkpoint [3, 17, 36, 50].

An implementation of FDAS is described in class `FDAS` (Class 1), using Java* [26]. Each process p_i maintains and propagates a dependency vector (Section 3). Process p_i also maintains a flag *afterSend* that captures whether a message has been sent or not during the current interval. The flag is reset after a checkpoint is taken and it is set after a message is sent.

The method `receiveMessage` contains the part of the FDAS that enforces RDT. Upon the reception of a message m, the dependency vector of the message is scanned. If a new dependency is established, say at $dv[k]$, and at least one message was sent in the current interval a forced checkpoint is taken. The dependency vector of the process is updated from $dv[k]$ to $dv[n]$, to register the new dependencies. The complexity of this method is $O(n)$.

*We have chosen Java because it is easy to read and has a widely known specification. Java is a trademark of Sun Microsystems, Inc.

Class 1 FDAS.java

```
public class FDAS {
    public static final int N = 100; // Number of processes in the computation
    public int pid; // A process unique identifier in the range 0..N-1
    protected int [ ] DV = new int [N]; // Automatically initialized to (0,...,0)
    protected boolean afterSend;

    public void takeCheckpoint() {
        // Save state to stable memory
        DV[pid]++;
        afterSend = false;
    }

    public FDAS(int pid) { this.pid = pid; } // Constructor

    public void run() { takeCheckpoint(); } // Initiate execution

    public void finalize() { takeCheckpoint(); } // Finish execution

    public void sendMessage(Message m) {
        m.DV = (int [ ]) DV.clone(); // Piggyback DV onto the message
        afterSend = true;
        // Send message
    }

    public void receiveMessage(Message m) {
        int k;
        for (k = 0; k < N && m.DV[k] ≤ this.DV[k]; k++)
            ; // Stop at the first new dependency
        if (k < N) { // New dependency
            if (afterSend)
                takeCheckpoint();
            for (; k < N; k++) // Update DV starting at the first new dependency
                if (m.DV[k] > DV[k]) DV[k] = m.DV[k];
        }
        // Message is processed by the application
    }
}
```

5 An Optimization based on Common Knowledge

The approach presented in the previous Section does not take advantage of the processes' common knowledge about their behavior. Consider a scenario where a process p_i receives a message m from p_j which has been sent during an interval already known by p_i due to the causal sequence of messages μ (Figure 4). Since p_j is not allowed to increase its dependency vector after a message-send event, process p_i can verify if a new dependency is established based solely on $dv_m[j]$. This observation takes us to an optimized version of the method `receiveMessage`.

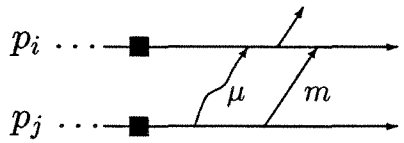


Figure 4: Process p_i already knows the interval in which m was sent

Class 2 FDAS.java (Optimized version of receiveMessage)

```
public class FDAS {
    /* ... */ // Same as in Class 4.1
    public void receiveMessage(Message m) {
        if (m.DV[m.sender] > DV[m.sender]) { // New dependency
            if (afterSend) takeCheckpoint();
            for (int k=0; k < N; k++) // Update DV
                if (m.DV[k] > DV[k]) DV[k] = m.DV[k];
        }
        // Message is processed by the application
    }
}
```

Theorem 5.1 *The optimized version of* `receiveMessage` *correctly implements FDAS.*

Proof: Consider the sequence of messages $\mu = (m_1, \ldots, m_\ell)$ from I_j^t to p_i, such that each message m_k, $1 \leq k < \ell$, is prime (it is the first message that carries information about I_j^t to the process that receives it). Consider that the last message of μ is received by p_i after a message-send event (Figure 5 (a)). We prove that p_i cannot have changed $dv_i[j]$ regardless the number ℓ of messages in μ.

Base: $\ell = 1$ In this case, μ is formed by a single message m (Figure 5 (b)). If $dv_m[j] > dv_i[j]$, process p_i would have taken a forced checkpoint before processing m.

Step: $\ell > 1$ Consider that no process is allowed to increase an entry of its dependency vector due to a sequence of $\ell - 1$ messages. We prove that this behavior also holds for a sequence of ℓ messages. Let m be the last message of μ, sent by process p_k during I_k^κ and let μ' be the first $\ell - 1$ messages of μ from I_j^ι to p_k (Figure 5 (c)). Since p_i does not take a checkpoint upon the reception of m, there must exist a sequence of messages μ'' from I_k^κ that arrives at p_i before m. Since m is the first message that brings information about I_j^ι to p_i, the last message of μ' must arrive at p_k after it has sent the first message of μ''. Thus, p_k increases the jth entry of its dependency vector during I_k^κ after a message-send event. □

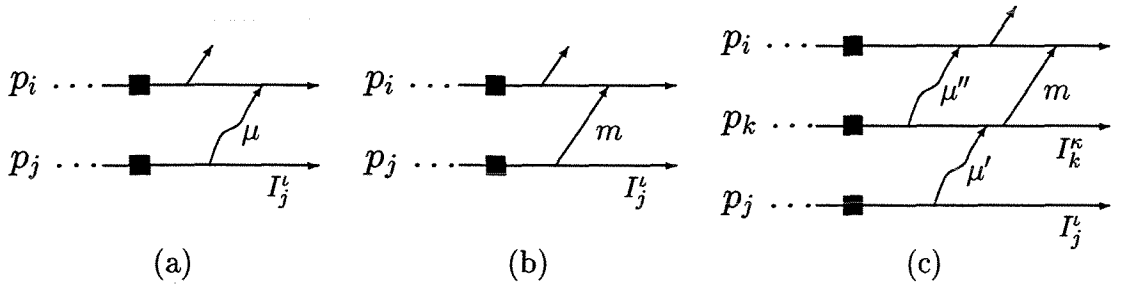


Figure 5: Contradiction hypothesis (a), base (b), and induction step (c) of Theorem 5.1

The observation of the knowledge shared by the processes has allowed us to get a reduction from $O(n)$ to $O(1)$ on the complexity of the decision to take a forced checkpoint. Besides that, the total complexity of the method `receiveMessage` is reduced to $O(1)$ when no new dependency is established. However, the sender of the message must be identified.

6 Conclusion

Fixed-Dependency-After-Send (FDAS) is an RDT protocol that forces the dependency vector of a process to remain unchanged during a checkpoint interval after the first message-send event [50]. In this paper, we have explored processes' common knowledge about their behavior to derive a simpler condition to induce checkpoints under FDAS. We have obtained a reduction from $O(n)$ to $O(1)$, where n is the number of processes in the computation, on the complexity to check if a new dependency is about to be established.

Our improvement can be directed applied to Fixed-Dependency-Interval, a previous version of FDAS that forces the dependency vector of a process to remain unchanged during a checkpoint interval [31, 50]. At the moment, we are investigating whether a similar improvement can be applied to the RDT protocol proposed by Baldoni, Helary, Mostefaoui, and Raynal [3]. A more general result would indicate that this approach to detect new dependencies can be used in all RDT checkpointing protocols.

Capítulo 6

On the Minimal Characterization of the Rollback-Dependency Trackability Property*

Islene Calciolari Garcia

Luiz Eduardo Buzato

Abstract

Checkpoint and communication patterns that enforce rollback-dependency trackability (RDT) have only on-line trackable checkpoint dependencies and allow efficient solutions to the determination of consistent global checkpoints.

Baldoni, Helary, and Raynal have explored RDT at the message level, in which checkpoint dependencies are represented by zigzag paths. They have presented many characterizations of RDT and conjectured that a certain communication pattern characterizes the minimal set of zigzag paths that must be tested *on-line* by a checkpointing protocol in order to enforce RDT. The contributions of this work are (i) a proof that their conjecture is false, (ii) a minimal characterization of RDT, and (iii) introduction of an original approach to analyze RDT checkpointing protocols.

Keywords: Distributed algorithms, fault-tolerance, distributed checkpointing, rollback recovery, zigzag paths.

*Este artigo foi publicado na *21th IEEE International Conference on Distributed Computing Systems*, em Phoenix, Arizona, Estados Unidos, em abril de 2001

1 Introduction

This paper is about the problem of finding the minimal characterization of the rollback-dependency trackability (RDT) property defined by Wang [50]. The search for the minimal characterization of RDT has led Baldoni, Helary, and Raynal [6] to state an interesting conjecture concerning the conditions that have to be tested in order to ensure *on-line* the RDT property. The contributions of this work are (i) a proof that their conjecture is false, (ii) a minimal characterization of the RDT property, and (iii) introduction of an original approach to analyze checkpointing protocols that enforce RDT.

A checkpoint is a recording in stable memory of a process' state. The set of all checkpoints taken by a distributed computation and the dependencies established among these checkpoints due to message exchanges form a checkpoint and communication pattern (CCP). CCPs that enforce RDT present only checkpoint dependencies that can be on-line trackable using dependency vectors, and allow efficient solutions to the determination of the maximum and minimum consistent global checkpoints that include a set of checkpoints [50]. Many applications can benefit from these algorithms: rollback recovery, software error recovery, and distributed debugging [50].

Netzer and Xu have determined that checkpoint dependencies are created by sequences of messages called *zigzag paths* [38]. Two types of zigzag paths can be identified: (i) causal paths (C-paths) and (ii) non-causal paths (Z-paths). C-paths can be on-line trackable through the use of dependency vectors; Z-paths, on the contrary, cannot be on-line trackable. However, a CCP may present Z-paths and still enforce RDT. In this case, all Z-paths must be *doubled* by a causal path; a Z-path is doubled by a causal one if the pair of checkpoints related by that Z-path is also related by a C-path [6].

The notion of RDT-compliance was introduced by Baldoni, Helary, and Raynal in order to establish properties that could reduce the set of Z-paths that must be doubled to guarantee RDT. This concept can be summarized as follows: "In a given CCP, an X-path is a Z-path that satisfies a property X. The property X is RDT-compliant if every CCP without X-paths satisfies the RDT property" [6]. In their work, they conjecture that a specific property X, named "non-visibly-doubled-EPSCM", determines the smallest set of Z-paths that must be tested on-line by a protocol that enforces RDT. This paper presents a property Y, named "non-visibly-doubled-PMM", that further reduces the set of Z-paths that must be tested.

The proof that property Y is RDT-compliant is based on the observation that a protocol that enforces RDT must enforce it in every consistent cut of the computation. During the progress of the computation, the most recent consistent cut can be used to

generate a consistent global checkpoint. This approach has its roots on the notion of progressive view, a sequence of consistent global checkpoints that could have occurred in this order during the computation, introduced in [22]. As far as we know, this approach to the analysis of protocol behavior is new, having helped us to introduce the concept of *left-doubling*: a Z-path is left-doubled in relation to a consistent cut if both, this Z-path and a C-path that doubles it, belong to the left (past) of the consistent cut. Left-doubling and consistent cuts have been important to construct the proof that the conjecture raised by Baldoni, Helary, and Raynal is false.

This paper is structured as follows. Section 2 introduces the computational model. Section 3 describes the RDT-compliance notion [6]. Section 4 presents the minimal characterization of RDT. Section 5 concludes the paper.

2 Computational model

A distributed computation is composed of n sequential processes $\{p_0, \ldots, p_{n-1}\}$ that communicate only by exchanging messages. Messages cannot be corrupted, but can be delivered out of order or lost. A checkpoint is an internal event that records the process' state in stable memory. Each process takes an initial checkpoint immediately after execution begins and a final checkpoint immediately before execution ends. The set of all checkpoints taken by a distributed computation and the dependencies established among these checkpoints due to message exchanges form a checkpoint and communication pattern (CCP).

2.1 Consistent cuts

The local history of a process p_i is modeled as a possibly infinite sequence of events $(e_i^1, e_i^2, \ldots)$. These events can be divided into internal events and communication events realized through the sending and receiving of messages. The analysis of causal precedence between events is fundamental for a better understanding of consistency [33].

Definition 2.1 Causal precedence—*Event* e_a^α *causally precedes* e_b^β $(e_a^\alpha \rightarrow e_b^\beta)$ *if (i)* $a = b$ *and* $\beta = \alpha + 1$*; (ii)* $\exists m : e_a^\alpha = send(m)$ *and* $e_b^\beta = receive(m)$*; or (iii)* $\exists e_c^\gamma : e_a^\alpha \rightarrow e_c^\gamma \wedge e_c^\gamma \rightarrow e_b^\beta$.

A cut of a distributed computation contains an initial prefix of each of the proceses' local histories. A consistent cut is left-closed under causal precedence and defines an

instant in a distributed computation [16]. If a cut $C' \subset C$, we can say that C' is in the past of C (Figure 1).

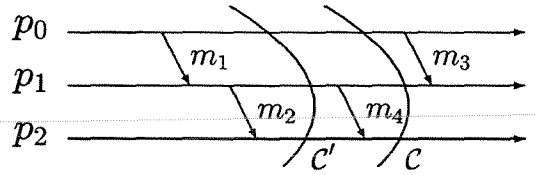


Figure 1: Consistent cuts

Definition 2.2 Consistent Cut—*A cut C is consistent if, and only if,*

$$e \in C \wedge e' \rightarrow e \Rightarrow e' \in C$$

2.2 Zigzag paths

Let c_i^γ denote the γth checkpoint taken by p_i. Two successive checkpoints $c_i^{\gamma-1}$ and c_i^γ define a checkpoint interval I_i^γ that represents the set of events produced by p_i between $c_i^{\gamma-1}$ and c_i^γ. Figure 2 illustrates a space-time diagram [33] augmented with checkpoints (black squares).

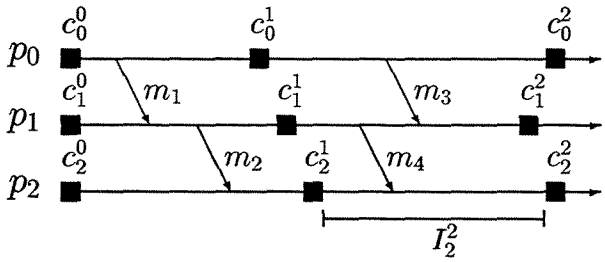


Figure 2: Checkpoints

Checkpoint dependencies are created by sequences of messages called *zigzag paths* [38]. Two types of zigzag paths can be identified: (i) causal paths (C-paths) and (ii) non-causal paths (Z-paths). A zigzag path is causal if the reception of each message but the last one causally precedes the send event of the next one in the sequence. In Figure 2, $[m_1, m_2]$ is a C-path that connects c_0^0 to c_2^1 and $[m_3, m_4]$ is a Z-path that connects c_0^1 to c_2^2.

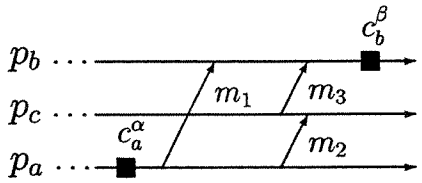
Definition 2.3 Zigzag path—*A sequence of messages $\mu = [m_1, \ldots, m_k]$ is a zigzag path that connects c_a^α to c_b^β if (i) p_a sends m_1 after c_a^α; (ii) if m_i, $1 \leq i < k$, is received by p_c, then m_{i+1} is sent by p_c in the same or a later checkpoint interval; (iii) m_k is received by p_b before c_b^β.*

Let $\zeta = [m_1]$ and $\zeta' = [m_2, m_3]$ be two Z-paths; the concatenation of ζ and ζ' will be denoted by $\zeta \cdot \zeta'$, or $\zeta \cdot [m_2, m_3]$, or $[m_1] \cdot \zeta'$, or $[m_1, m_2, m_3]$ [6].

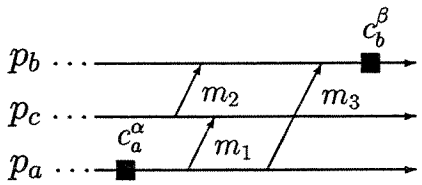
2.3 Rollback-dependency trackability

Definition 2.4 Rollback-dependency trackability—*A checkpoint pattern enforces the RDT property if all Z-paths are causally doubled.*

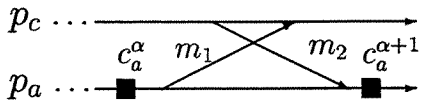
A Z-path is doubled by a causal one if the pair of checkpoints related by that Z-path is also related by a C-path [6]. In Figure 3 (a), the Z-path $[m_2, m_3]$ that connects c_a^α to c_b^β is causally doubled by m_1. In Figure 3 (b), $[m_1, m_2]$ is causally doubled by m_3. In Figure 3 (c), $[m_1, m_2]$ is trivially doubled due to the process execution.



(a) $[m_2, m_3]$ is doubled by m_1



(b) $[m_1, m_2]$ is doubled by m_3



(c) $[m_1, m_2]$ is trivially doubled

Figure 3: Causal doubling

Definition 2.5 Causal doubling—*A Z-path that connects c_a^α to c_b^β is* causally doubled *if $a = b$ and $\alpha < \beta$ or there exists a C-path μ that connects c_a^α to c_b^β.*

A zigzag path that starts in an interval I_b^β and finishes in an interval $I_b^{\beta'}$ such that $\beta' < \beta$ cannot be causally doubled. Such checkpoint pattern is a Z-cycle and identifies a useless checkpoint, that is, a checkpoint that cannot be part of any consistent global checkpoint [38]. Figure 4 presents a Z-path $[m_1, m_2]$ that cannot be causally doubled. A CCP that enforces RDT does not contain useless checkpoints [50].

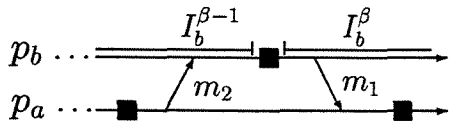


Figure 4: A Z-cycle cannot be doubled

2.4 Communication-induced protocols

A communication-induced checkpointing protocol allows processes to take checkpoints asynchronously (basic checkpoints), but they may be induced by the protocol to take additional checkpoints (forced checkpoints) [17, 36]. Forced checkpoints can be taken upon the arrival of a message, but before this message is processed by the computation. The decision to take a forced checkpoint must be based only on the local knowledge of a process; there is no global knowledge or knowledge about the future of the computation. Figure 5 presents a protocol that direct processes to take a forced checkpoint if at least one message has been sent in the current interval (No-Receive-After-Send) [36, 50].

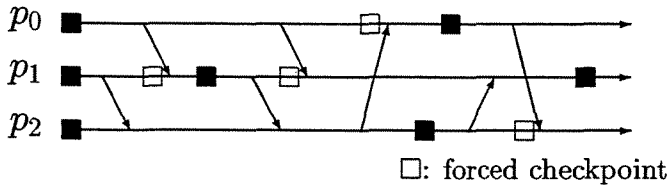


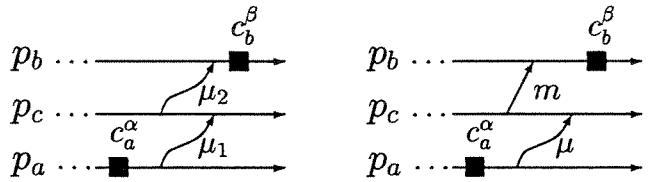
Figure 5: A communication-induced protocol

3 RDT-compliant properties

The notion of RDT-compliance was introduced by Baldoni, Helary, and Raynal in order to establish properties that could reduce the set of Z-paths that must be doubled in order to guarantee RDT, and, consequently produce more efficient protocols. The RDT-compliance notion can be summarized as follows: "In a given CCP, an X-path is a Z-path that satisfies a property X. The property X is RDT-compliant if every CCP without X-paths satisfies the RDT property" [6].

3.1 A Hierarchy of Z-Paths

A Z-path can be seen as a concatenation of C-paths $\mu_1 \cdot \mu_2 \cdot \ldots \cdot \mu_k$. The number of C-paths in a Z-path is the *order* of this path. A Z-path of order 2 (CC-path) is composed of exactly two causal paths μ_1 and μ_2 (Figure 6 (a)). A CM-path is a Z-path of order 2 composed of a causal path μ and a single message m (Figure 6 (b)).



(a) CC-path (b) CM-path

Figure 6: Z-paths of order 2

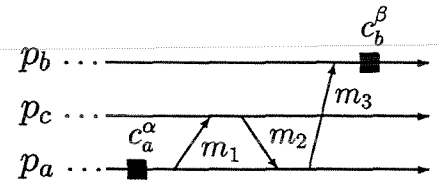
Following, we define a hierarchy of paths based on constraints that can be applied to a causal path μ.

Definition 3.1 Elementary path—*The sequence of processes traversed by the C-path has no repetition.*

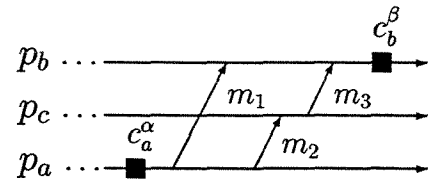
Definition 3.2 Prime path—*A C-path μ that connects c_a^α to c_b^β is prime if the last message of μ is the first message that brings to p_b the knowledge about c_a^α.*

Definition 3.3 Simple path—*A C-path $\mu = [m_1, \ldots, m_k]$ is simple if for every message m_i, $1 \leq i < k$, receive(m_i) and send(m_{i+1}) occur in the same checkpoint interval.*

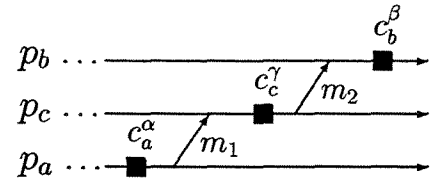
Let us consider the C-paths that connect c_a^α to c_b^β as shown in Figures 7 (a,b,c). In Figure 7 (a), $[m_1, m_2, m_3]$ is not elementary, because the same process sends m_1 and m_3; message m_3 is an elementary C-path. In Figure 7 (b), $[m_2, m_3]$ is not prime due to m_1. In Figure 7 (c), $[m_1, m_2]$ is not simple due to c_c^γ. These examples help us with the introduction of EPCM-paths and EPSCM-paths.



(a) $[m_1, m_2, m_3]$ is not elementary



(b) $[m_2, m_3]$ is not prime



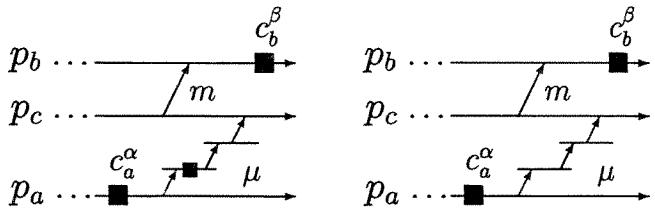
(c) $[m_1, m_2]$ is not simple

Figure 7: Constraints on C-paths

Definition 3.4 EPCM-path—*An EPCM-path $\zeta = \mu \cdot [m]$ is a CM-path composed of a C-path μ and a single message m such that μ is elementary and prime.*

Definition 3.5 EPSCM-path—*An EPSCM-path $\zeta = \mu \cdot [m]$ is an EPCM-path composed of a C-path μ and a single message m such that μ is simple.*

Figure 8 presents an EPCM-path and an EPSCM-path. Informally, the main difference between these paths is that the C-path of an EPSCM-path does not contain checkpoints.



(a) EPCM-path (b) EPSCM-path

Figure 8: Constrained paths

In this work, we analyze more restricted Z-paths. An MM-path is a CM-path composed of two single messages m_1 and m_2. A PMM-path is an MM-path such that m_1 is prime (Figure 9); since m_1 is trivially elementary and simple, a PMM-path is an EPSCM-path.

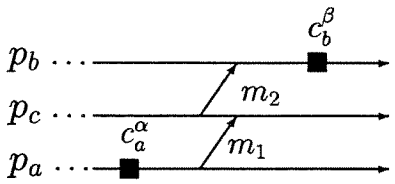


Figure 9: A PMM-path

Definition 3.6 PMM-path—*A PMM-path* $\zeta = [m_1] \cdot [m_2]$ *is an EPSCM-path composed of two messages* m_1 *and* m_2.

3.2 Non-doubled PMM is not RDT-compliant

Baldoni, Helary, and Raynal have proved that non-doubled CC, non-doubled CM and non-doubled EPSCM are RDT-compliant properties. The CCP presented in Figure 10, taken from [6], was introduced to show that the restriction on the length of the causal component of an EPSCM-path does not lead to an RDT-compliant property. We can use this CCP to show that non-doubled PMM is not an RDT-compliant property: all PMM-paths are causally doubled, but the CCP does not satisfy RDT, since the EPSCM-path $[m_1, m_2, m_3].[m]$ is not causally doubled.

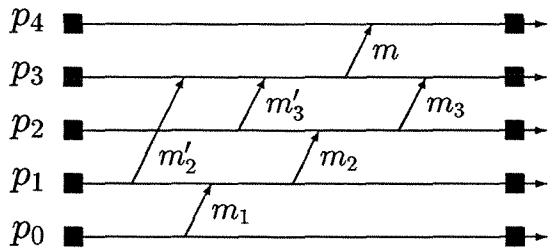


Figure 10: No non-doubled PMM, but not RDT

3.3 Visible properties

A property X is *visible* if it can be tested on-line upon the arrival of a message. On the contrary, a *non-visible* property cannot be tracked on-line and may depend on the future of the computation. Non-doubled EPCM, for example, is a non-visible property. However, we can derive two communication-induced protocols from this property: a protocol that breaks all EPCM-paths [50] and a protocol that breaks just the EPCM-paths that are not perceived as causally doubled [3]. Figure 11 shows an EPCM-path $\mu \cdot [m]$ that is causally doubled by a C-path ν. Process p_c is able to detect this doubling due to the C-path ν'. In this case, $\mu \cdot [m]$ is *visibly doubled* by ν.

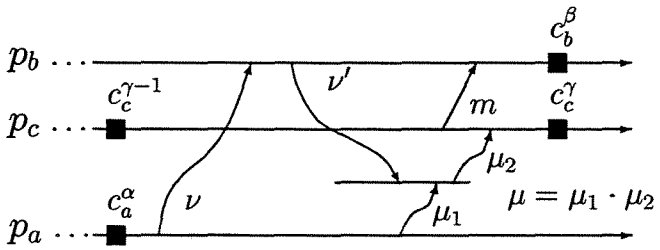


Figure 11: A visibly doubled EPCM-path

Definition 3.7 Visibly Doubled EPCM-path—*An EPCM-path $\mu \cdot [m]$ is visibly doubled if (i) is causally doubled by a C-path ν and (ii) the reception of the last message of ν causally precedes the sending of the last message of μ.*

Based on the concept of visible doubling and in the results discussed in the last Section, Baldoni, Helary, and Raynal have proposed the following conjecture [6].

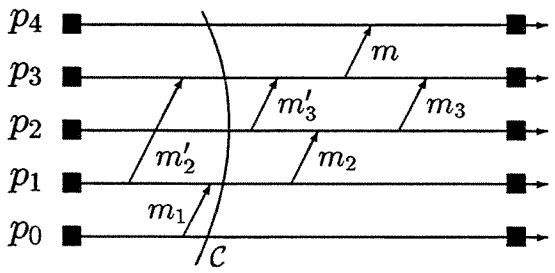
Conjecture 3.1 *The set of non-visibly-doubled-EPSCM-paths is the smallest one to test for breaking in order to ensure* on-line *the RDT property.*

In the next Section, we prove that this conjecture does not hold. Although the property non-doubled PMM is not RDT-compliant, the property non-visibly-doubled-PMM is RDT-compliant.

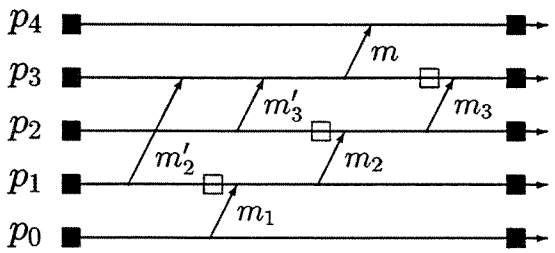
4 The minimal characterization of RDT

Let us analyze the counter-example presented in [6], in the usual context of communication-induced checkpointing protocols (Section 2.4). The processes must enforce RDT without (i) global knowledge or (ii) knowledge about the future of the computation. Figure 12 (a) shows a consistent cut $\mathcal{C}$ such that the PMM-path $[m_1] \cdot [m_2']$ is in the past of $\mathcal{C}$ and the C-path $[m_1, m_2, m_3]$ that causally doubles it is in the future of $\mathcal{C}$. This behavior contradicts (i) and (ii) above.

The CCP depicted in Figure 12 (b) should be observed. Process p_1 must take a forced checkpoint upon the arrival of m_1, because it is not able to know that the PMM-path $[m_1] \cdot [m_2']$ will be causally doubled by the C-path $[m_1, m_2, m_3]$. Similarly, process p_2 must take a forced checkpoint because it does not know that the PMM-path $[m_2] \cdot [m_3']$ is already causally doubled by $[m_2']$. Finally, process p_3 must take a forced checkpoint because, due to the forced checkpoint taken by p_2, $[m_3] \cdot [m]$ is a PMM-path.



(a) $[m_1, m_2']$ is doubled in the future of $\mathcal{C}$



(b) Forced checkpoints

Figure 12: The behavior of RDT protocols

An RDT protocol must enforce RDT in every consistent cut of the computation. Thus, we introduce the concept of *left-doubling*: a Z-path is left-doubled in relation to a consistent cut if both, this Z-path and a C-path that doubles it, belong to the left (past) of the consistent cut.

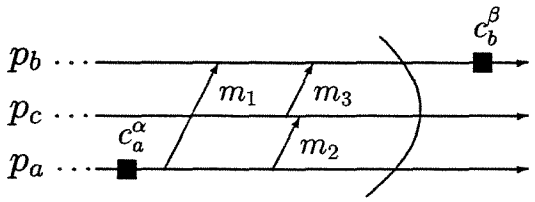
Definition 4.1 *A C-path μ belongs to a consistent cut $\mathcal{C}$ if the reception of the last message of μ belongs to $\mathcal{C}$.*

Definition 4.2 *A Z-path $\zeta = \mu_1 \cdot \mu_2 \cdot \ldots \cdot \mu_\ell$ belongs to a consistent cut $\mathcal{C}$ if all causal components of ζ belong to $\mathcal{C}$.*

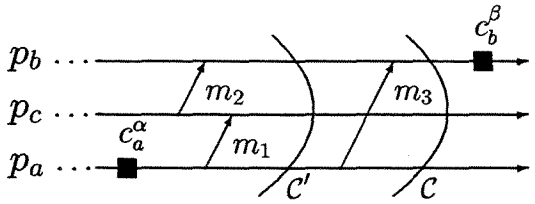
Definition 4.3 Left-doubling—*A Z-path ζ is left-doubled in relation to a consistent cut $\mathcal{C}$ if, and only if*

- *ζ belongs to $\mathcal{C}$;*
- *ζ is doubled by a C-path μ that also belongs to $\mathcal{C}$.*

A Z-path $\zeta = \mu_1 \cdot \mu_2 \cdot \ldots \cdot \mu_\ell$ may be left-doubled in relation to any consistent cut that contains ζ; in Figure 13 (a) this happens because the reception of m_1 causally precedes the reception of m_3. However, a Z-path ζ can be left-doubled in relation to a consistent cut but not in relation to another one (Figure 13 (b)).



(a) $[m_2, m_3]$ is left-doubled in relation to any consistent cut that contains it



(b) $[m_1, m_2]$ is left-doubled in relation to $\mathcal{C}$, but not in relation to $\mathcal{C}'$

Figure 13: Left-doubling

Let us consider the case in which a Z-path ζ is from a process p_b to itself, say ζ is from I_b^{β} to $I_b^{\beta'}$. As described in Section 2.3, if $\beta' < \beta$ this Z-path is a Z-cycle and cannot be causally doubled. If $\beta \geq \beta'$, this Z-path is trivially doubled due to the process execution. In order to simplify the presentation of the results that follow, we are going to consider all trivially doubled Z-paths as left-doubled by a *fictitious* C-path μ (Figure 14).

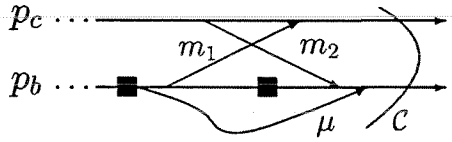


Figure 14: A trivially left-doubled Z-path

Definition 4.4 *RDT(C)—A consistent cut C satisfies the RDT property if, and only if, all Z-paths that belong to C are left-doubled.*

A consistent cut C does not satisfy RDT ($\overline{RDT}(C)$) if at least one Z-path that belongs to C is not left-doubled.

4.1 Left-doubled CC-paths and CM-paths

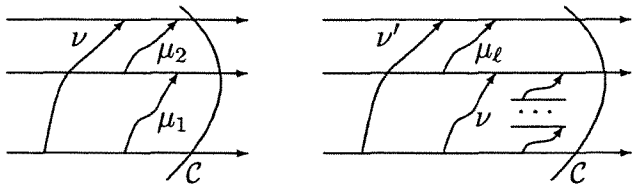
The following lemmas are similar to the ones presented in [6], but they are based on the left-doubling approach.

Lemma 4.1 *Given a consistent cut C, if all CC-paths are left-doubled, all Z-paths are left-doubled.*

Proof:

Base: A Z-path of order 2 $\zeta = \mu_1 \cdot \mu_2$ must be left-doubled by a C-path ν (Figure 15 (a)).

Step: Let us assume that all Z-paths of order $\ell - 1$ are left-doubled. We are going to prove that all Z-paths of order ℓ are left-doubled. Let $\mu_1 \cdot \mu_2 \cdot \ldots \cdot \mu_\ell$ be a Z-path of order ℓ. According to the induction hypothesis, the Z-path $\mu_1 \cdot \mu_2 \cdot \ldots \cdot \mu_{\ell-1}$ must be left-doubled by a causal path ν. The zigzag path $\nu \cdot \mu_\ell$ can be a C-path or a CC-path that must be left-doubled by a C-path ν' (Figure 15 (b)). □



(a) Base (b) Induction step

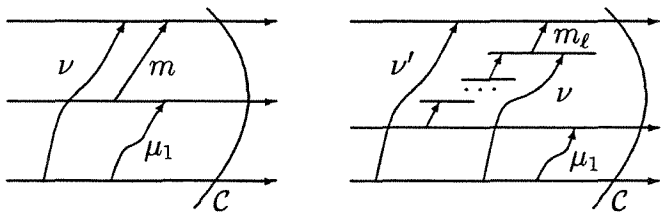
Figure 15: Proof of Lemma 4.1

Lemma 4.2 *Given a consistent cut C, if all CM-paths are left-doubled, all CC-paths are left-doubled.*

Proof: Let $\mu_1 \cdot \mu_2$ be a non-left-doubled CC-path. The proof uses induction on the number of messages of μ_2.

Base: $|\mu_2| = 1$ The CC-path $\mu_1 \cdot \mu_2$ is a CM-path and must be left-doubled by a C-path ν (Figure 16 (a)).

Step: Let us assume that all CC-paths $\mu_1 \cdot \mu_2$ such that $|\mu_2| \leq \ell - 1$ are left-doubled. Let us consider the case in which $\mu_2 = [m_1 \cdot m_2 \cdot \ldots \cdot m_\ell]$. According to the induction hypothesis, the CC-path $\mu_1 \cdot [m_1 \cdot m_2 \cdot \ldots \cdot m_{\ell-1}]$ must be left-doubled by a C-path ν. The zigzag path $\nu \cdot [m_\ell]$ is C-path or a CM-path that must be left-doubled by a C-path ν' (Figure 16 (b)). □



(a) Base (b) Induction step

Figure 16: Proof of Lemma 4.2

4.2 Non-visibly-doubled PMM is RDT-compliant

In this Section, we prove that a protocol that breaks all non-visibly-doubled PMM-paths enforces RDT. Informally, our proof is based on the observation that if such a protocol has failed to enforce RDT at a consistent cut C, it must have failed in the past. Since the

initial consistent global checkpoint defines a consistent cut that trivially enforces RDT, the protocol must be correct. The "first instant" that a system fails to guarantee RDT is a minimum consistent cut such that $\overline{RDT}(\mathcal{C})$, that is, there is no consistent cut $\mathcal{C}'$ in the past of $\mathcal{C}$ such that $\overline{RDT}(\mathcal{C}')$.

Definition 4.5 Minimum $\overline{RDT}(\mathcal{C})$—$\mathcal{C}$ *is a* minimum *consistent cut that does not satisfy RDT if, and only if,*

$$\forall \text{ consistent cut } \mathcal{C}' \mid \mathcal{C}' \subset \mathcal{C} \Rightarrow RDT(\mathcal{C}')$$

Let us assume a checkpoint pattern $\mathcal{P}$ that does not enforce RDT. We can easily construct an algorithm to determine a minimum consistent cut $\mathcal{C}$ such that $\overline{RDT}(\mathcal{C})$. Since $\mathcal{P}$ does not enforce RDT, there must exist at least one consistent cut $\mathcal{C}$ such that $\overline{RDT}(\mathcal{C})$. Let us consider the case in which $\mathcal{C}$ is not minimum. Thus, there must exist a consistent cut $\mathcal{C}'$ such that $\mathcal{C}' \subset \mathcal{C}$ and $\overline{RDT}(\mathcal{C}')$. Analogously, we can repeat this process until we find a minimum consistent cut $\mathcal{C}''$ such that $\overline{RDT}(\mathcal{C}'')$.

Theorem 4.1 *Non-visibly-doubled PMM is an RDT-compliant property.*

Proof: Let us consider a checkpoint pattern $\mathcal{P}$ such that $\mathcal{P}$ has no non-visibly-doubled PMM-path, but it does not satisfy RDT. Let $\mathcal{C}$ be a consistent cut in $\mathcal{P}$ such that $\mathcal{C}$ is a minimum $\overline{RDT}(\mathcal{C})$. Thus, according to Lemma 4.2, $\mathcal{C}$ must contain at least one non-left doubled CM-path $\mu \cdot [m]$. Let us assume that μ is from p_a to p_c and m is from p_c to p_b (Figure 17 (a)).

Let m' be the last message of μ, sent by process p_d, such that μ is the concatenation of a C-path μ' and the message m' (Figure 17 (b)). In the special case that $\mu = [m']$, ν is empty and $p_d = p_c$; this case is also covered. We have two possibilities for the MM-path $[m'] \cdot [m]$.

- $[m'] \cdot [m]$ is not a PMM-path (Figure 17 (c)), or
- $[m'] \cdot [m]$ is visibly doubled (Figure 17 (d)).

$[m'] \cdot [m]$ is not a PMM-path There exists a C-path ν from p_d to p_c that arrives at p_c before m' (Figure 17 (c)).

It is possible to construct a consistent cut $\mathcal{C}'$ in the past of $\mathcal{C}$ including μ', ν, and m but not m' (Figure 17 (e)). For the sake of contradiction, let us assume that it is not possible to construct such a consistent cut. In this case, there must exist an event $e \notin \mathcal{C}$ such that e precedes the reception of the last message of μ', or the reception of the last

message of ν, or the reception of m. This contradicts the hypothesis that $\mathcal{C}$ is a consistent cut. Let us consider the case in which *receive*(m') causally precedes *receive*(m) due to a C-path ν'. A C-path $\mu \cdot \nu'$ would left double $\mu \cdot [m]$, contradicting the hypothesis that $\mu \cdot [m]$ is a non-left-doubled CM-path.

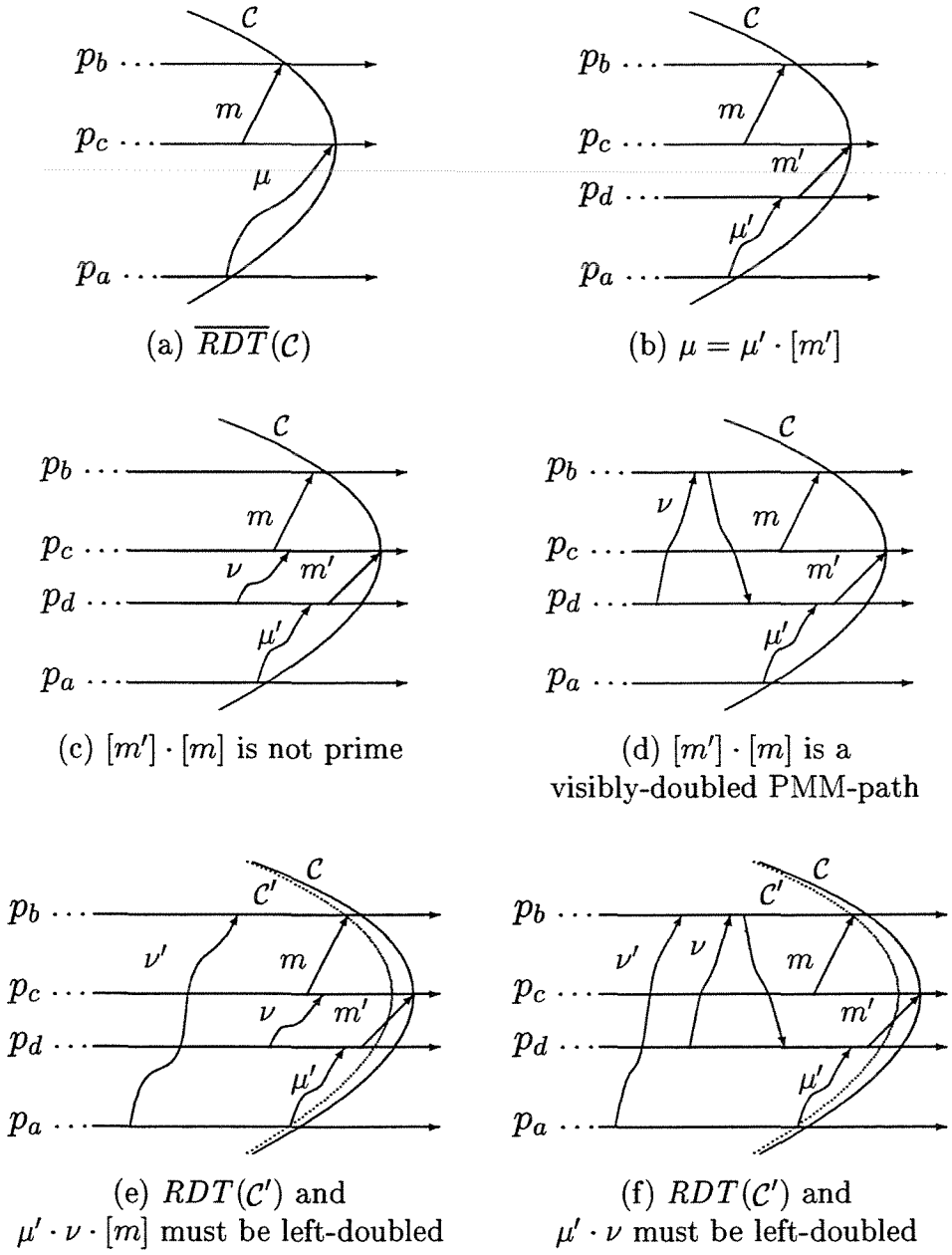
We know that $RDT(\mathcal{C}')$, otherwise $\mathcal{C}$ would not be a minimum $\overline{RDT}(\mathcal{C})$. Thus, every Z-path that belongs to $\mathcal{C}'$ is left-doubled. In particular, the Z-path $\mu' \cdot \nu \cdot [m]$ must be left-doubled by a C-path ν' (Figure 17 (e)). The existence of ν' contradicts the hypothesis that $\mu \cdot [m]$ is a non-left-doubled CM-path.

$[m'] \cdot [m]$ is a visibly-doubled PMM-path Let us consider the case that $m' \cdot m$ is visibly-doubled by a C-path ν. By definition, the reception of the last message of ν causally precedes the sending of m' and ν must belong to $\mathcal{C}$. It is possible to construct a consistent cut $\mathcal{C}'$ in the past of $\mathcal{C}$ including μ', ν and m, but not m' (Figure 17 (f)). Since $RDT(\mathcal{C}')$, the Z-path $\mu' \cdot \nu$ must be left-doubled by a C-path ν'. The existence of ν' contradicts the hypothesis that $\mu \cdot [m]$ is a non-left-doubled CM-path. □

4.3 Minimality of the characterization

A visible RDT-compliant property is minimal if it cannot be implied by any other visible RDT-compliant property. In this Section, we discuss why the presented characterization is minimal in the computational model described.

Since a PMM-path is composed by two messages, this is the minimal Z-path allowed in the computational model described; we cannot reduce the number of messages, otherwise we obtain a C-path. If a non-visibly-doubled PMM-path is formed, there is a risk that the resulting checkpoint pattern does not enforce RDT. In the absence of global information or knowledge about the future of the computation, such risk cannot be taken.



(a) $\overline{RDT}(\mathcal{C})$

(b) $\mu = \mu' \cdot [m']$

(c) $[m'] \cdot [m]$ is not prime

(d) $[m'] \cdot [m]$ is a visibly-doubled PMM-path

(e) $RDT(\mathcal{C}')$ and $\mu' \cdot \nu \cdot [m]$ must be left-doubled

(f) $RDT(\mathcal{C}')$ and $\mu' \cdot \nu$ must be left-doubled

Figure 17: Non-Visibly-Doubled PMM is an RDT-compliant property

5 Conclusion

Research on the minimality of RDT characterization is important fundamentally because it can lead to checkpointing protocols that exhibit the following attributes: (i) take less forced checkpoints and/or (ii) are more elegant in terms of the control structures maintained and propagated by processes and the condition tested to induce forced checkpoints. The first attribute leads to a performance measure where a checkpoint protocol, say CP1, is considered better than a protocol CP2 if the number of forced checkpoints induced by CP1 is smaller than the number of forced checkpoints induced by CP2. In [45], Tsai, Kuo, and Wang have shown that there is no optimal on-line RDT protocol that takes fewer forced checkpoints than any other RDT protocol for all possible communication patterns. The second attribute motivates an alternative performance measure, that is, a protocol CP1, based on a condition C1, is considered better than CP2, based on condition C2, if, for every message m, C1 $\Rightarrow$ C2. In [5], Baldoni, Helary, and Raynal propose such alternative measure to rank RDT protocols.

This paper has provided the smallest characterization of RDT, with the implication that it is now finally possible to check whether a checkpointing protocol is optimal with regard to the performance measure proposed by Baldoni, Helary, and Raynal [5]. Our result has been obtained by looking at checkpoint patterns from a distinct point of view, through the simple observation that a protocol that enforces RDT must enforce it in every consistent cut of the computation. During the progress of the computation, the most recent consistent cut can be used to generate a consistent global checkpoint. This approach has its roots on the notion of progressive view, a sequence of consistent global checkpoints that could have occurred in this order during the computation, introduced in [22].

Capítulo 7

RDT-Partner: An Efficient Checkpointing Protocol that Enforces Rollback-Dependency Trackability*

Islene Calciolari Garcia

Gustavo Maciel Dias Vieira

Luiz Eduardo Buzato

Abstract

Checkpoint patterns that enforce rollback-dependency trackability (RDT) have only on-line trackable checkpoint dependencies and allow efficient solutions to the determination of consistent global checkpoints. The design of RDT checkpointing protocols that are efficient both in terms of the number of forced checkpoints and in terms of the data structures propagated by the processes is a very interesting research topic. Fixed-Dependency-After-Send (FDAS) is an RDT protocol based only on vector clocks, but that takes a high number of forced checkpoints. The protocol proposed by Baldoni, Helary, Mostefaoui and Raynal (BHMR) takes less forced checkpoints than FDAS, but requires the propagation of an $O(n^2)$ matrix of booleans.

In this paper, we introduce a new RDT protocol, called RDT-Partner, in which a process can save forced checkpoints in comparison to FDAS during checkpoint intervals in which the communication is bound to a pair of processes; a very interesting optimization in the context of client-server applications. Although the data structures required by the proposed protocol maintain the $O(n)$ complexity of FDAS, theoretical and simulation studies show that it takes virtually the same number of forced checkpoints than BHMR.

Keywords: distributed algorithms, fault-tolerance, checkpointing, rollback recovery, rollback-dependency trackability

*Este artigo foi publicado no 19º Simpósio Brasileiro de Redes de Computadores, em Florianópolis, Santa Catarina, em maio de 2001.

1 Introduction

A checkpoint is a recording in stable memory of a process' state. The set of all checkpoints taken by a distributed computation and the dependencies established among these checkpoints due to message exchanges form a checkpoint pattern. Checkpoint patterns that enforce rollback-dependency trackability (RDT) present only checkpoint dependencies that can be on-line trackable using vector clocks or dependency vectors, and allow efficient solutions to the determination of the maximum and minimum consistent global checkpoints that include a set of checkpoints [50]. Many applications can benefit from these algorithms: rollback recovery, software error recovery, and distributed debugging [50].

In order to enforce the RDT property, an RDT checkpointing protocol [3, 50] allows processes to take checkpoints asynchronously (basic checkpoints), but they may be induced by the protocol to take additional checkpoints (forced checkpoints). Some RDT checkpointing protocols presented in the literature are based only on checkpoints, message-send, and message-receive events: No-Receive-After-Send, Checkpoint-After-Send, Checkpoint-Before Receive, and Checkpoint-After-Send-Before-Receive [50]. These protocols are instantiations of the Mark-Receive-Send model [42] and are prone to induce a large number of forced checkpoints.

An effort to reduce the number of forced checkpoints can be done through the collaboration of the processes to maintain and propagate control structures. The decision to take a forced checkpoint must be based on information piggybacked on application messages; there are no control messages and no knowledge about the future of the computation. An important goal is to develop an efficient protocol both in terms of the number of forced checkpoints and in terms of the complexity of the required data structures.

Fixed-Dependency-Interval (FDI) [31, 50] and Fixed-Dependency-After-Send (FDAS) [50] maintain and propagate vector clocks. They force the vector clock of a process to remain unchanged during an entire checkpoint interval (FDI) or after the first message-send event of an interval (FDAS). Trying to reduce even more the number of forced checkpoints, Baldoni, Helary, Mostefaoui, and Raynal have explored the RDT property at the message level [3, 5, 6]. A protocol proposed by them, called BHMR, never takes more forced checkpoints than FDAS [45]. However, the more elaborated condition used by BHMR requires the propagation of an additional $O(n^2)$ matrix of booleans [3].

In this paper, we introduce a new RDT protocol, called RDT-Partner, in which a process can save forced checkpoints in comparison to FDAS during checkpoint intervals in which the communication is bound to a pair of processes; a very interesting optimization in the context of client-server applications. This protocol is based on a recent result that characterized the strongest condition that can be used on-line by an RDT checkpointing protocol [23]. Although the data structures required by the proposed protocol

maintain the $O(n)$ complexity of FDAS, theoretical and simulation studies show that it takes virtually the same number of forced checkpoint than BHMR.

The paper is structured as follows. Section 2 introduces rollback-dependency trackability. Section 3 describes the RDT-Partner protocol. Section 4 compares the proposed protocol with FDAS and BHMR. Section 5 summarizes the paper.

2 Rollback-Dependency Trackability

The literature presents two approaches to define rollback-dependency trackability. The first one is based on the study of on-line trackable dependencies, implemented through the use of vector clocks or dependency vectors [50]. The other approach is based on the study of sequence of messages [3, 5, 6, 23].

2.1 Computational model

A distributed computation is composed of n sequential processes $(p_1, \ldots, p_n)$ that communicate only by exchanging messages. Messages cannot be corrupted, but can be delivered out of order or lost. The activity of a process is modeled as a sequence of events that can be divided into internal events and communication events realized through the sending and the receiving of messages. Checkpoints are internal events; each process takes an initial checkpoint (immediately after execution begins) and a final checkpoint (immediately before execution ends). Figure 1 illustrates a space-time diagram [33] augmented with checkpoints (black squares).

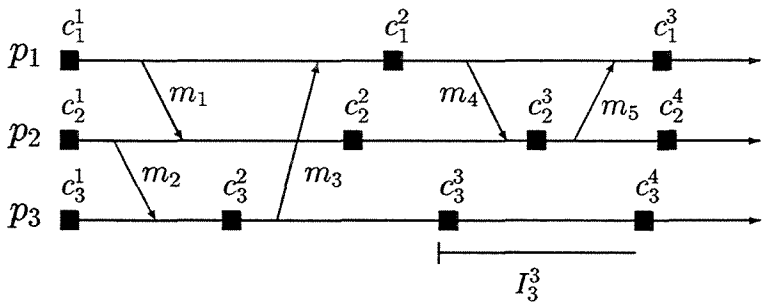


Figure 1: A distributed computation

Let c_i^γ denote the γth checkpoint taken by p_i. A non-final checkpoint c_i^γ, $\gamma \geq 1$, and its immediate successor $c_i^{\gamma+1}$ define a checkpoint interval I_i^γ. This interval represents the set of events produced by p_i between c_i^γ and $c_i^{\gamma+1}$, including c_i^γ and excluding $c_i^{\gamma+1}$.

2.2 Checkpoint dependencies

A consistent global checkpoint is a set of checkpoints, one per process, that could have been seen by an idealized external observer [14]. Checkpoints that are part of the same consistent global checkpoint cannot be related by checkpoint dependencies. Netzer and Xu have determined that checkpoint dependencies are created by sequences of messages called *zigzag paths* [38].

Definition 2.1 Zigzag path—*A sequence of messages $\mu = [m_1, \ldots, m_k]$ is a zigzag path that connects c_a^α to c_b^β if (i) p_a sends m_1 after c_a^α; (ii) if m_i, $1 \leq i < k$, is received by p_c, then m_{i+1} is sent by p_c in the same or a later checkpoint interval; (iii) m_k is received by p_b before c_b^β.*

Two types of zigzag paths can be identified: (i) causal paths and (ii) non-causal paths. A zigzag path is causal if the reception of each message but the last one causally precedes the send event of the next one in the sequence. In Figure 1, $[m_2, m_3]$ is a causal path from c_2^1 to c_1^2 and $[m_1, m_2]$ is a non-causal zigzag path from c_1^1 to c_3^2.

A zigzag path that connects a checkpoint to itself is called a Z-cycle and identifies a useless checkpoint, that is, a checkpoint that cannot be part of any consistent global checkpoint [38]. In Figure 1, $[m_3, m_1, m_2]$ and $[m_5, m_4]$ are examples of Z-cycles; c_3^2 and c_2^3 are useless checkpoints.

Let $\zeta = [m_1]$ and $\zeta' = [m_2, m_3]$ be two zigzag paths; the concatenation of ζ and ζ' will be denoted by $\zeta \cdot \zeta'$, or $\zeta \cdot [m_2, m_3]$, or $[m_1] \cdot \zeta'$, or $[m_1, m_2, m_3]$ [6].

2.3 Vector clocks

A transitive dependency tracking mechanism can be used to capture causal dependencies among checkpoints. Each process p_i maintains and propagates a size-n vector clock vc_i, such that $vc_i[i]$ is initialized to 1 and all other entries to 0. The entry $vc_i[i]$ represents the current interval of p_i and it is incremented immediately before a new checkpoint is taken. Every other entry $vc_i[j]$, $j \neq i$, represents the highest checkpoint index of p_j that p_i causally depends and it is updated every time a message m with a greater value of $vc_m[j]$ arrives to p_i.

Figure 2 depicts the vector clocks established during a distributed computation. Note that the vector clock associated to checkpoint c_3^2 is (1, 1, 2) and it correctly represents the dependencies of this checkpoint. Unfortunately, not all dependencies can be tracked on-line. For example, the vector clock associated to checkpoint c_3^3 does not capture the existence of a zigzag path that connects c_1^2 to c_3^3.

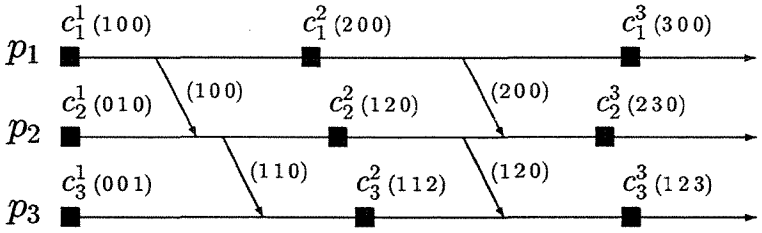


Figure 2: A distributed computation with vector clocks

Definition 2.2 Rollback-dependency trackability (vector clock characterization) *A checkpoint pattern enforces RDT if all checkpoint dependencies can be on-line trackable through the use of vector clocks.*

RDT is a desirable property because when all dependencies are causal dependencies, efficient algorithms can be used to construct consistent global checkpoints. Also, an RDT checkpoint pattern does not admit useless checkpoints [50].

When a communication-induced checkpointing protocol is used to enforce RDT, processes take checkpoints asynchronously (basic checkpoints), but they may be induced by the protocol to take additional checkpoints (forced checkpoints) [17, 36]. Forced checkpoints can be taken upon the arrival of a message, but before this message is processed by the computation.

In the FDAS protocol, a forced checkpoint is taken upon the reception of a message if (i) at least one message has been sent during the current interval, and (ii) at least one entry of the vector clock is about to be changed [50]. Figure 3 shows the same scenario of Figure 2 under FDAS; the resulted checkpoint pattern enforces RDT.

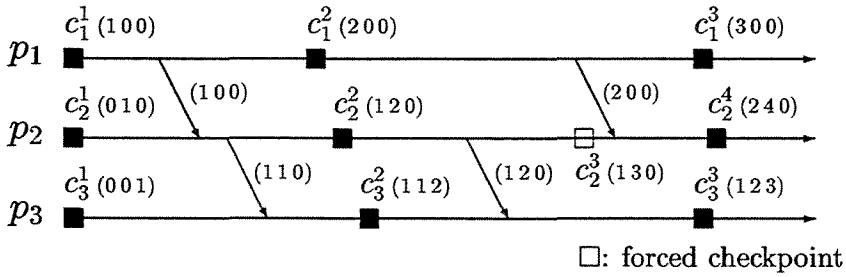


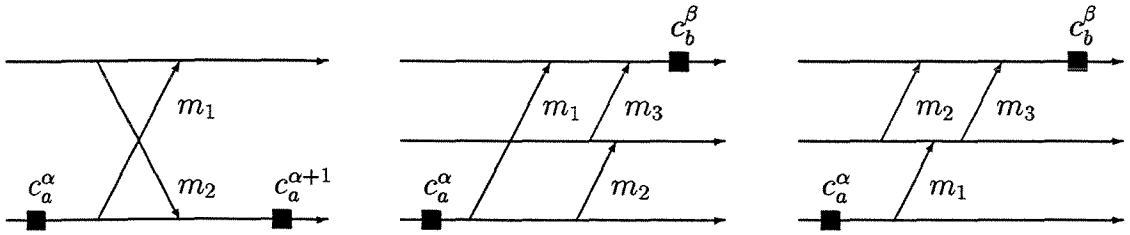
Figure 3: A distributed computation under FDAS

2.4 A message-based characterization of RDT

Trying to design more efficient protocols, Baldoni, Helary, Mostefaoui, and Raynal have explored the RDT property in the message level [3, 5, 6].

Causal doubling

A non-causal zigzag path is doubled by a causal one if the pair of checkpoints related by that zigzag path is also related by a causal dependency [6]. In Figure 4 (a) $[m_1, m_2]$ connects c_a^α to $c_a^{\alpha+1}$ and it is trivially doubled by the execution flow of p_a. In Figure 4 (b), the non-causal zigzag path $[m_2, m_3]$ that connects c_a^α to c_b^β is causally doubled by m_1. In Figure 4 (c), $[m_1, m_2]$ is causally doubled by $[m_1, m_3]$.



(a) $[m_1, m_2]$ is trivially doubled

(b) $[m_2, m_3]$ is doubled by m_1

(c) $[m_1, m_2]$ is doubled by $[m_1, m_3]$

Figure 4: Causal doubling

Definition 2.3 Trivial doubling—*A non-causal zigzag path from c_a^α to c_b^β is trivially doubled if $a = b$ and $\alpha < \beta$.*

Definition 2.4 Causal doubling—*A non-causal zigzag path from c_a^α to c_b^β is* causally doubled *if it is trivially doubled or there exists a causal path μ from c_a^α to c_b^β.*

A Z-cycle cannot be causally doubled. In Figure 5, the zigzag path $[m_1, m_2]$ cannot be doubled by the process execution because a causal dependency from c_b^β to c_b^β cannot exist.

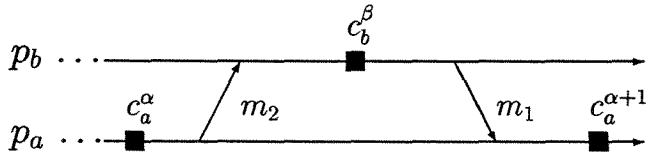


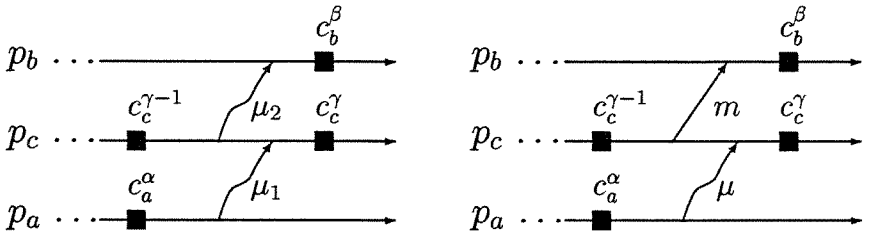
Figure 5: Z-cycle: $[m_1, m_2]$ cannot be causally doubled

Definition 2.5 Rollback-Dependency Trackability (zigzag path characterization) *A checkpoint pattern enforces the RDT property if all zigzag paths are causally doubled.*

Since it appears to be very hard to track non-causally doubled zigzag paths, Baldoni, Helary and Raynal have suggested an approach that tries to minimize the number of non-causal zigzag paths that must be causally doubled to enforce RDT [6].

PCM-paths

A non-causal zigzag path can be seen as a concatenation of causal paths $\mu_1.\mu_2.\cdots.\mu_k$. The number of causal paths in a non-causal zigzag path is the *order* of this path. A non-causal zigzag path of order 2 (CC-path) is composed of exactly two causal paths μ_1 and μ_2 (Figure 6 (a)). A CM-path is a non-causal zigzag path of order 2 composed of a causal path μ and a single message m (Figure 6 (b)).



(a) CC-path (b) CM-path

Figure 6: Non-causal zigzag paths of order 2

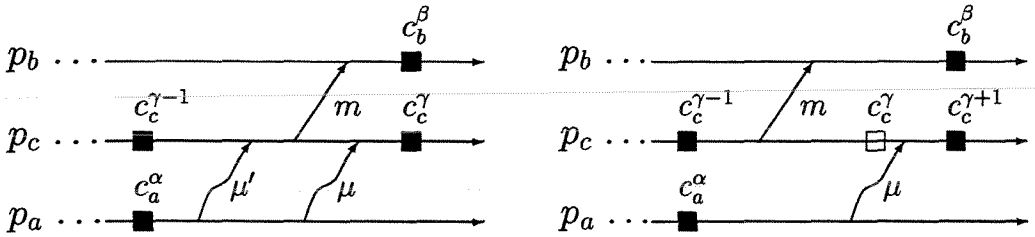
In order to further reduce the set of zigzag paths that must be doubled, let us consider an additional constraint on the causal path μ of a CM-path $\mu.[m]$. Let μ be a *prime* path from c_a^α to c_c^γ, that is, the first path that brings to p_c the knowledge about c_a^α. In Figure 7 (a), μ is not prime due to existence of μ'; in Figure 7 (b), μ is a prime path.

Definition 2.6 Prime path—*A causal path μ from c_a^α to c_c^γ is prime if the last message of μ is the first message that brings to p_c the knowledge about c_a^α.*

Definition 2.7 PCM-path—*A PCM-path is a non-causal zigzag path composed of a **p**rime **c**ausal path μ and a single **m**essage m.*

Definition 2.8 Rollback-Dependency Trackability (PCM-path characterization) *A checkpoint pattern enforces the RDT property if all PCM-paths are causally doubled.*

This characterization leads to the development of a simple RDT protocol that *breaks* all PCM-paths, that is, induces a process to take a forced checkpoint upon the establishment of a PCM-path. This behavior is illustrated in Figure 7 and can be seen as a reinterpretation of the FDAS protocol [5, 6].



(a) $\mu.[m]$ is not a PCM-path (b) p_c must *break* $\mu.[m]$

Figure 7: FDAS can be seen as a protocol that breaks PCM-paths

Visibly doubling

A PCM-path need not to be broken by a process p_c if, upon the establishment of this path, p_c is able to detect that it is already causally doubled [3, 5, 6]. Figure 8 shows a PCM-path $\mu \cdot [m]$ that is causally doubled by a causal path ν; process p_c is able to detect this doubling due to the causal path ν'. In this case, $\mu \cdot [m]$ is *visibly doubled* by ν.

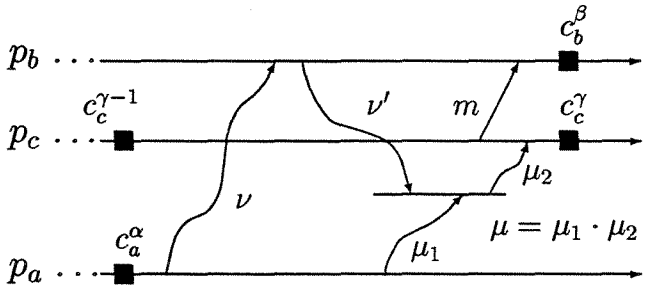


Figure 8: A visibly doubled PCM-path

Definition 2.9 Visibly Doubled PCM-path—*A PCM-path $\mu \cdot [m]$ is visibly doubled if (i) is causally doubled by a causal path ν and (ii) the reception of the last message of ν causally precedes the sending of the last message of μ.*

Definition 2.10 Rollback-Dependency Trackability (Visibly doubled characterization) *A checkpoint pattern enforces the RDT property if all PCM-paths are visibly doubled.*

BHMR is a protocol that breaks all non-visibly doubled PCM-paths. Unfortunately, this more efficient strategy in terms of forced checkpoints is less efficient in terms of data structures [5]. Each process must maintain and propagate information regarding other processes' knowledge about causal relantionships, requiring an $O(n^2)$ data structure. Indeed, in the implementation of BHMR [3] each process maintains and propagates an $O(n)$ vector clock, an $O(n)$ vector of booleans, and an $O(n^2)$ matrix of booleans.

2.5 The minimal characterization of RDT

Recently, we have determined the minimal (strongest) condition that can be used to enforce RDT [23]. A PMM-path is a non-causal zigzag path composed of two single messages m_1 and m_2, such that m_1 is prime (Figure 9). We have proved that a protocol that breaks all non-visibly doubled PMM-paths must enforce RDT [23]. In the next Section, we are going to explore this characterization to propose a protocol that is efficient both in terms of forced checkpoints and in terms of data structures.

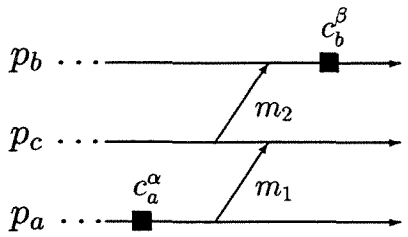


Figure 9: A PMM-path

Definition 2.11 PMM-path—*A PMM-path is a non-causal zigzag-path composed of a* ***p****rime single* ***m****essage* m_1 *and a single* ***m****essage* m_2.

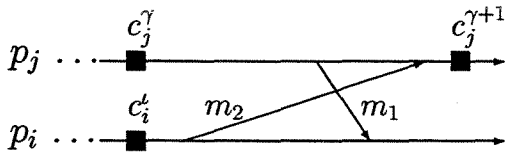
Definition 2.12 Rollback-Dependency Trackability (minimal characterization) *A checkpoint pattern enforces the RDT property if all PMM-paths are visibly doubled.*

3 RDT-Partner protocol

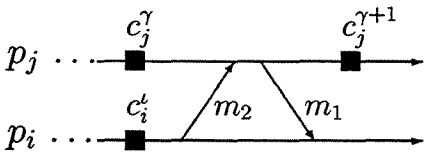
The implementation of a protocol that keeps track of all visibly-doubled paths seems to require $O(n^2)$ information [5]. In this Section, we introduce an $O(n)$ protocol, called RDT-Partner, that keeps track of only trivially-doubled PMM-paths.

3.1 PMM-cycles

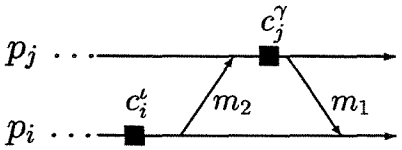
A PMM-path $[m_1, m_2]$ that starts and finishes in the same process is a PMM-cycle. Some PMM-cycles are trivially doubled by the process execution; others cannot be causally doubled because they form Z-cycles. Figure 10 illustrates the possibilities of PMM-cycles from the perspective of a process p_i. In Figure 10 (a) and (b), $[m_1, m_2]$ is trivially doubled due to the p_j's execution flow from c_j^γ to $c_j^{\gamma+1}$. In Figure 10 (c), $[m_1, m_2]$ is a Z-cycle and cannot be causally doubled (a causal dependency from c_j^γ to c_j^γ cannot exist).



(a) When m_1 is sent, process p_j has no knowledge about c_i^ι ($[m_1, m_2]$ is trivially doubled)



(b) When m_1 is sent, process p_j has knowledge about c_i^ι ($[m_1, m_2]$ is trivially doubled)



(c) When m_1 is sent, process p_j has knowledge about c_i^ι ($[m_1, m_2]$ cannot be causally doubled)

Figure 10: PMM-cycles

Let us assume that a process p_i maintains and propagates a vector clock vc_i and let us analyze the complexity required to distinguish trivially doubled PMM-cycles from Z-cycles. First, let us consider the scenario depicted in Figure 10 (a), in which upon the sending of m_1, p_j has no knowledge about c_i^ι. Process p_i can deduce, upon the reception of m_1, that m_2 will be received during I_j^γ or a later checkpoint interval and that $[m_1, m_2]$ will be trivially doubled by the p_j's execution flow. To identify this scenario, p_i must evaluate the following condition: $vc_{m_1}[i] < vc_i[i]$.

Let us consider the other scenarios, in which upon the sending of m_1, p_j has knowledge about c_i^t. In both cases, upon the reception of m_1, p_i would detect that $vc_{m_1}[i] = vc_i[i]$. Thus, p_i needs additional information to distinguish the scenario depicted in Figure 10 (b) from the scenario depicted in Figure 10 (c). Message m_1 can carry a boolean value to indicate whether p_j has taken a forced checkpoint after the last time it received knowledge about a new checkpoint index in p_i.

3.2 Partner relationships

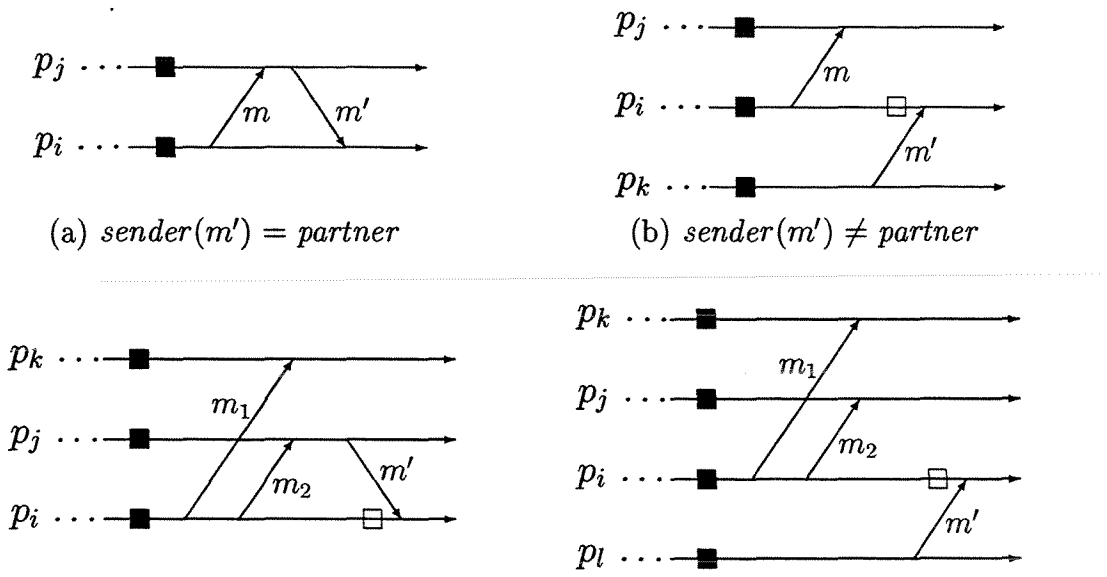
In the previous section, we have shown that a process can efficiently detect trivially doubled PMM-cycles. The detection of visibly doubled PMM-paths is similar to the detection of visibly doubled PCM-paths and seems to require $O(n^2)$ information [5]. This means that a process p_i can efficiently save a forced checkpoint only in a constrained situation, in which p_i sends a message to p_j and receives another message from p_j.

Definition 3.1 Partner—*Process p_j is a partner of process p_i if p_i has sent a message to p_j during the current interval.*

Let us consider that p_i has just one partner, say p_j, in the current interval. If p_i receives a message m' from p_j (Figure 11 (a)) it can save a forced checkpoint if a Z-cycle is not established, as described in Section 3.1. If p_i receives a message m' from another process, say p_k (Figure 11 (b)), that forms a PMM-path, it must take a forced checkpoint before processing m'. Process p_i will take this forced checkpoint even if this PMM-path is visibly doubled, because it will not be able to efficiently detect this doubling.

Let us consider that p_i has more than one partner, say p_k and p_j, and it receives a message m' from p_j (Figure 11 (c)). Process p_i can efficiently detect that $[m', m_2]$ is not a Z-cycle, but it cannot efficiently detect whether the PMM-path $[m', m_1]$ is visibly doubled. Thus, process p_i must take a forced checkpoint before processing m'. A similar situation occurs when p_i receives a message from another process, say p_l (Figure 11 (d)). In this case, two PMM-paths $[m', m_1]$ and $[m', m_2]$ are formed, and p_i must take a forced checkpoint before processing m'.

Finally, Figure 12 illustrates that forced checkpoints can be saved in a nested sequence of partner interactions.



(a) $sender(m') = partner$

(b) $sender(m') \neq partner$

(c) $sender(m')$ is one of the partners

(d) $sender(m')$ is not one of the partners

Figure 11: The behavior of the RDT-Partner protocol

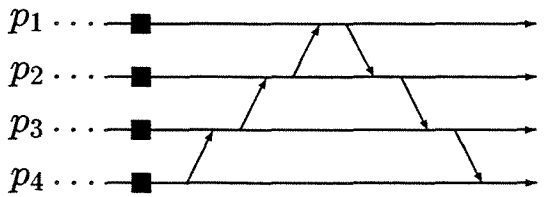


Figure 12: A nested sequence of partner interactions

3.3 RDT-Partner implementation

An implementation of RDT-Partner is described in class `RDT_Partner` (Class 1), using Java* [26]. Every process, say p_i, maintains and propagates a vector clock vc_i in order to characterize casual precedence among checkpoints. When p_i sends a message, vc_i is piggybacked onto it. Before consuming a message m, process p_i takes a component-wise maximum of vc_i and vc_m. When p_i takes a checkpoint, it increments $vc_i[i]$.

*We have chosen Java because it is easy to read and has a precise description. Java is a trademark of Sun Microsystems, Inc.

Class 1 RDT_Partner.java

```
public class RDT_Partner {
    public static final int N = 100; // Number of processes in the computation
    public int pid; // A process unique identifier in the range 0..N-1
    protected int [ ] vc = new int [N]; // Vector clock, automatically initialized to (0 ... 0)
    protected boolean [ ] simple = new boolean [N]; // Keeps track of simple PMM-cycles
    protected int partner_pid;
    public static final int NO_PARTNER = - 1, MORE_THAN_ONE_PARTNER = N + 1;

    public class Message {
        public int sender, receiver;
        public int [ ] vc;
        public boolean simple;
        // Message body
    }

    public void takeCheckpoint() {
        vc[pid]++; // Increment checkpoint index immediately before the checkpoint
        // Save state to stable memory
        partner_pid = NO_PARTNER;
        for (int i = 0; i < N; i++) simple[i] = i == pid;
    }

    public RDT_Partner(int pid) { this.pid = pid; } // Constructor
    public void run() { takeCheckpoint(); } // Initiate execution
    public void finalize() { takeCheckpoint(); } // Finish execution

    public void sendMessage(Message m) {
        m.vc = (int [ ]) vc.clone(); // Piggyback vc onto the message
        m.simple = simple[m.receiver];
        if (partner_pid == NO_PARTNER) partner_pid = m.receiver;
        else if (partner_pid ≠ m.receiver) partner_pid = MORE_THAN_ONE_PARTNER;
        // Send message
    }

    public void receiveMessage(Message m) {
        if (m.vc[m.sender] > vc[m.sender]) {
            if (partner_pid ≠ NO_PARTNER && // PMM-path
                (partner_pid ≠ m.sender || // Not a PMM-cycle
                 (partner_pid == m.sender && m.vc[pid] == vc[pid] && !m.simple))) // Z-cycle
                takeCheckpoint(); // Forced checkpoint
            simple[m.sender] = true;
        }
        for (int i = 0; i < N; i++) // Update the vector clock
            if (m.vc[i] > vc[i]) vc[i] = m.vc[i];
        // Message is processed by the application
    }
}
```

Process p_i also maintains a variable *partner_pid* that keeps track of partner relationships during checkpoint intervals:

- *partner_pid* = NO_PARTNER indicates that p_i has not sent any message;
- *partner_pid* = j indicates that p_i has sent message(s) only to p_j;
- *partner_pid* = MORE_THAN_ONE_PARTNER indicates that p_i has sent messages for more than one process.

In order to distinguish trivially doubled PMM-cycles from Z-cycles, each process p_i maintains a vector of booleans *simple*, such that *simple*$[j]$, $j \neq i$, indicates that a checkpoint has not been taken after p_i has received knowledge about $c_j^{vc_i[j]}$. When p_i sends a message to p_j it piggybacks *simple*$[j]$ onto the message. When process p_i takes a checkpoint it sets all entries in *simple* to false, except its ith entry.

A checkpoint is induced by p_i before delivering a message m if a PMM-path is detected and one of the following condition holds: (i) it is not a PMM-cycle or (ii) it is a PMM-cycle, but forms a Z-cycle.

4 A comparison with FDAS and BHMR

4.1 Simulation results

Our experimental data was obtained using the simulation toolkit for quasi-synchronous algorithms Metapromela [48]. This toolkit was built atop Spin [30], a tool to simulate and perform consistency analysis of distributed protocols and algorithms. In Metapromela, the processes are asynchronous and the simulated execution of each process is a succession of atomic events of three types: internal, message-send and message-receive. The only type of internal event that is relevant for checkpointing is the occurrence of a basic checkpoint. The environment is controlled by adjusting the distribution of these events and the communication network.

The experiment was performed considering a complete network, i.e., each pair of processes is connected by a bidirectional communication channel. The channels do not lose, corrupt or change the order of messages. For each experimental point it was considered the average of 10 measurements. Each measurement was taken by the execution of each of the studied protocols under the same pattern of messages and basic checkpoints, that is, under exactly the same history of events. We counted the ratio of forced checkpoints per process over a period of 300 basic checkpoints for each process. Figure 13 shows the results obtained for $2 \leq n \leq 20$, where n is the number of processes in the computation.

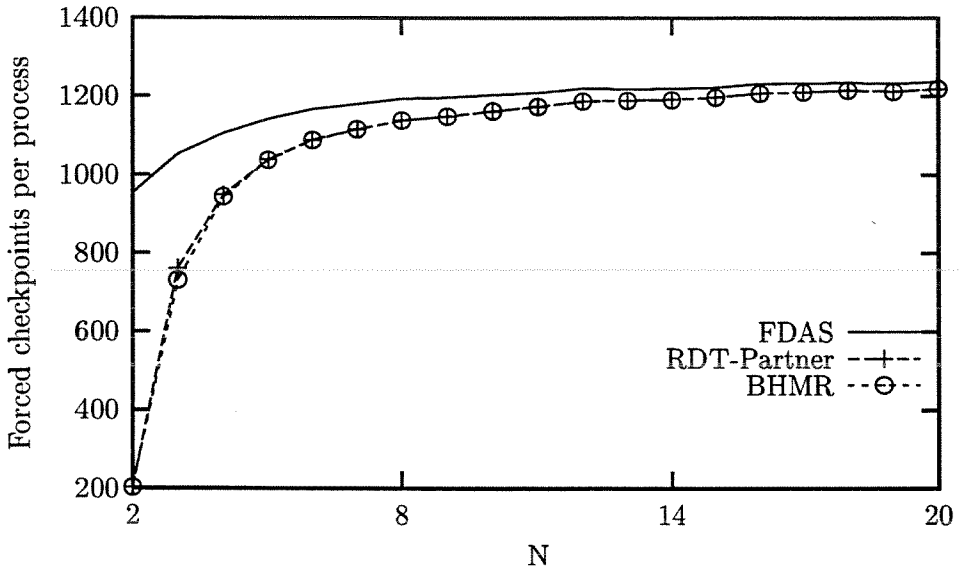


Figure 13: Simulation results

FDAS takes consistently more checkpoints than BHMR and RDT-Partner. For small values of n, say 2 or 3, the difference is large, but the difference becomes less significant as the values of n grow. BHMR and RDT-Partner take almost the same number of forced checkpoints with a very small difference for $3 \leq n \leq 5$. In the following section, we give a theoretical explanation for this behavior.

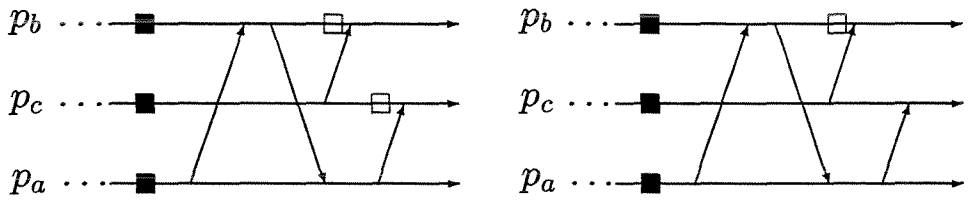
4.2 Theoretical argumentation

The theoretical analysis performed in [45] proved that any protocol that uses a stronger condition than FDAS to take forced checkpoints will outperform FDAS. Thus, FDAS cannot take less checkpoints than BHMR and RDT-Partner, since both protocols break doubled PMM-cycles, and FDAS is not able to track such dependencies. Figure 14 shows two-message scenarios in which BHMR and RDT-Partner can save a checkpoint in comparison to FDAS. Since in our simulation the communication is uniform, the probability of these scenarios is higher for small values of n, explaining the behavior of the curves.

In comparison to RDT-Partner, BHMR can also detect visibly doubled PCM-paths, potentially saving more forced checkpoints. However, the minimum scenario in which BHMR can save a checkpoint in comparison to RDT-Partner requires four messages (Figure 15) and is likely to occur less frequently during a distributed computation. In a system composed of three processes, this situation could be more likely, explaining the slightly difference in the curves around this point ($n = 3$).

(a) FDAS (b) BHMR and RDT-partner

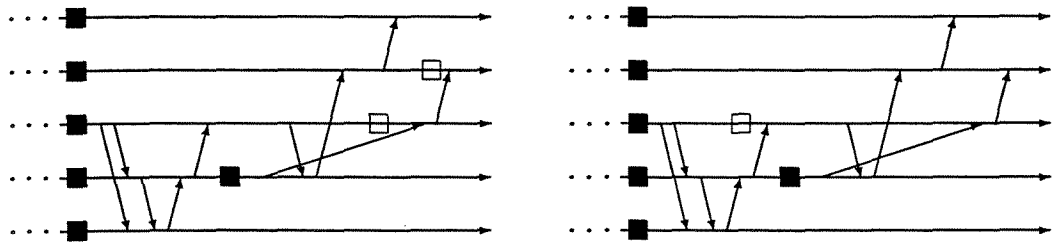
Figure 14: BHMR and RDT-Partner can save forced checkpoints in comparison to FDAS



(a) RDT-partner (b) BHMR

Figure 15: BHMR can save a forced checkpoint in comparison to RDT-Partner

Although this situation cannot occur with FDAS, it is possible for a protocol based on a weaker condition to outperform a protocol based on a stronger condition [45]. BHMR uses a condition stronger than RDT-Partner and Figure 16 illustrates a scenario in which RDT-Partner can save a forced checkpoint in comparison to BHMR. This scenario involves five processes and ten messages. We can consider such scenarios less probable in distributed computations.



(a) BHMR (b) RDT-Partner

Figure 16: RDT-Partner can save a forced checkpoint in comparison to BHMR

5 Conclusion

Checkpoint patterns that enforce the rollback-dependency trackability (RDT) property allow efficient solutions to the determination of consistent global checkpoints [50]. In this paper, we have introduced a new RDT protocol, called RDT-Partner, that is efficient both in terms of the number of forced checkpoints and in terms of the complexity of the required data structures.

We have presented theoretical and simulation studies to show that RDT-Partner presents a very good compromise between the stronger condition of BHMR [3] and the smaller control structure of FDAS [50]. In conclusion, RDT-Partner is so far the best protocol to adopt in practical implementations.

Capítulo 8

Systematic Analysis of Index-Based Checkpointing Algorithms using Simulation*

Gustavo Maciel Dias Vieira

Islene Calciolari Garcia

Luiz Eduardo Buzato

Abstract

Index-based checkpointing allows the use of simple and efficient algorithms for domino-effect free construction of recovery lines. In this paper, we use a simulation toolkit to analyze the behavior of index-based algorithms. We present a performance study of the well-known algorithm proposed by Briatico, Ciuffoletti, and Simoncini and explore the impact of some optimizations of this algorithm presented in the literature. Our results indicate that an expensive and complex optimization may not reduce the number of forced checkpoints in comparison to a simpler optimization.

Keywords: distributed checkpointing, rollback recovery, logical clocks, simulation of distributed systems.

*Este artigo foi publicado no *IX Simpósio de Computação Tolerante a Falhas*, em Florianópolis, Santa Catarina, em março de 2001.

1 Introduction

A checkpoint is a stable memory record of a process state. A consistent global checkpoint is a set of checkpoints, one per process, that could have been seen by an idealized observer external to the computation [14]. A recovery line is a consistent global checkpoint from which a distributed computation can be restarted after a failure. Fault tolerance based on checkpoints and recovery lines can be divided into three autonomous activities. *Checkpointing* is concerned with efficient protocols for the recording of checkpoints. *Recovery* deals with efficient protocols for constructing and rolling processes back to a recovery line [17]. *Garbage collection* removes from stable memory checkpoints that are no longer useful to rollback recovery [51].

Checkpointing strategies are classified as [35]: asynchronous, quasi-synchronous, and synchronous. When processes take checkpoints asynchronously, the distributed computation is subject to the domino effect, that may force all processes to return to their initial state in the worst case [41]. In synchronous checkpointing, processes synchronize their checkpointing activity to construct a recovery line [14, 32]. A quasi-synchronous checkpointing algorithm allows processes to take checkpoints asynchronously (basic checkpoints), but they may be induced by the algorithm to take additional checkpoints (forced checkpoints) [17, 36] in order to avoid the domino effect.

Index-based checkpointing [9, 11, 17, 28, 35, 36] is a quasi-synchronous approach that allows simple and efficient algorithms for rollback recovery and garbage collection. Checkpoints are timestamped with indexes that are similar to Lamport's logical clocks [33] in a way that checkpoints with the same index form a consistent global checkpoint. The first algorithm to use this approach was the one proposed by Briatico, Ciuffoletti, and Simoncini (BCS) [11]. BCS is very simple and efficient. However, its performance is strongly coupled to the policy adopted to take basic checkpoints. For example, if basic checkpoints are taken periodically according to a global clock, no forced checkpoint is ever taken. In contrast, if each process takes basic checkpoints at different rates, many forced checkpoints may be required. In order to overcome this weakness, many optimizations of BCS were proposed [9, 28, 35].

In this paper, we use the simulation toolkit Metapromela [48] to analyze the behavior of BCS and its optimizations. This toolkit was built atop Spin [30] and it makes possible the comparison of the algorithms using a strictly controlled environment, allowing different algorithms to be executed under the same checkpoint and communication pattern.

We have identified and analyzed the individual impact of many optimizations of BCS, including the ones that only appeared combined with other optimizations in the literature. Our results indicate that an expensive and complex optimization [9] may not reduce the number of forced checkpoints in comparison to a simpler optimization [9, 29].

This paper is structured as follows. Section 2 introduces the computational model. Section 3 describes the simulation model. Section 4 analyzes the behavior of index-based checkpointing algorithms. Section 5 concludes the paper.

2 Computational Model

A distributed computation is carried out by n sequential processes $(p_0, \ldots, p_{n-1})$ that communicate only by exchanging messages. Messages cannot be corrupted, but can be delivered out of order or lost. A process is modeled as a sequence of *events* that can be divided into internal events and communication events realized through the sending and the receiving of messages.

Checkpoints are internal events; each process takes an initial checkpoint (immediately after execution begins) and a final checkpoint (immediately before execution ends). A checkpoint and its immediate successor in the same processes define a checkpoint interval that represents the set of events executed by this process between the two checkpoints. A global checkpoint is formed by a set of causally unrelated (concurrent) checkpoints. Figure 1 illustrates a distributed computation as a space-time diagram [33] augmented with checkpoints (black squares); I represents a checkpoint interval and $\hat{\Sigma}$ a consistent global checkpoint.

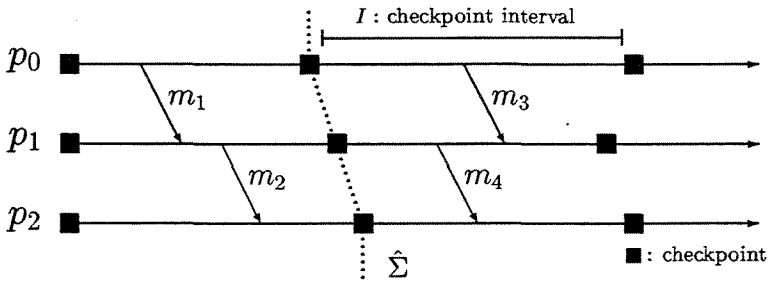


Figure 1: A distributed computation

A quasi-synchronous checkpointing algorithm allows processes to take checkpoints asynchronously (basic checkpoints), but they may be induced by the algorithm to take additional checkpoints (forced checkpoints) [17, 36] in order to avoid the domino effect. Forced checkpoints can be taken upon the arrival of a message, but before this message is processed by the computation. The decision to take a forced checkpoint must be based only on the local knowledge available at a process; there is no global knowledge or knowledge about the future of the computation.

Figure 2 presents a quasi-synchronous checkpointing algorithm that direct processes to take a forced checkpoint if at least one message has been sent in the current interval (No-Receive-After-Send) [36, 50].

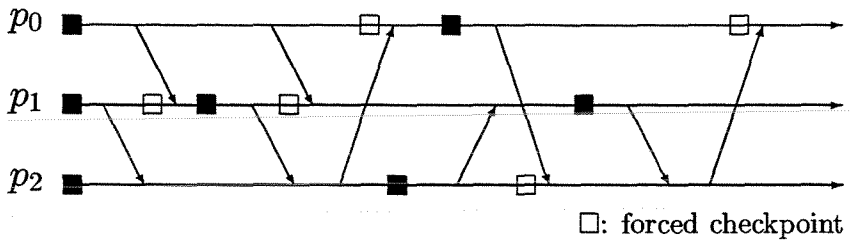


Figure 2: A quasi-synchronous checkpointing algorithm

3 Simulation Environment

Our experimental data was obtained using the simulation toolkit for quasi-synchronous algorithms Metapromela [48]. This toolkit was built atop Spin [30], a tool to simulate and perform consistency analysis of distributed protocols and algorithms.

In Metapromela, the processes are asynchronous and the simulated execution of each process is a succession of atomic events of three types: internal, send-message and receive-message. The only type of internal event that is relevant for checkpointing is the occurrence of a basic checkpoint. The environment is controlled by adjusting the distribution of these events and the communication network.

All of the experiments were performed considering a complete network, i.e., each pair of processes is connected by a bidirectional communication channel. The channels do not lose, corrupt or change the order of messages. For each experimental point it was considered the average of 10 measurements. Each measurement was taken by the execution of each of the studied algorithms under the same pattern of messages and basic checkpoints. We counted the ratio of forced checkpoints per basic checkpoint over a period of 300 basic checkpoints. We considered two scenarios in this study:

None-faster: In the none-faster scenario all processes have, on average, the same number of events between basic checkpoints. The processes do not take basic checkpoints at exactly the same time, just the ratio of basic checkpoints per events is the same. This setting represents a situation where all the processes behave in the same way and have the same execution speed. Particularly, we have an average of 8 communication events

between any two basic checkpoints. We have made the measurements varying the number of processes from 2 to 20.

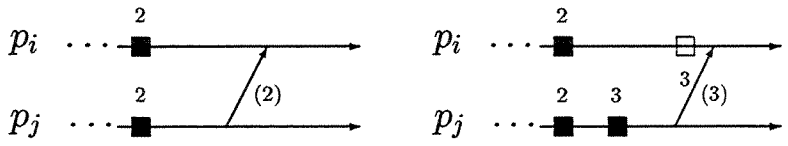
One-faster: In the one-faster scenario all but one process behave as in the none-faster scenario and this process has a smaller number of events between the basic checkpoints. This setting represents a system with a single process that has more important local states and is willing to take more basic checkpoints, introducing asymmetry in the pattern of basic checkpoints. In this scenario we consider a system with 6 processes and vary the difference in basic checkpoint ratio between the faster process and the others. We have made measurements with this difference ranging from 1 to 30 times faster.

4 Index-based Checkpointing

The essence of index-based checkpointing is that checkpoints with the same index form a consistent global checkpoint. The goal is to produce an index-based checkpointing protocol that guarantees this property, with a minimum number of forced checkpoints. In this Section, we use simulation to analyze the performance of some index-based protocols proposed in the literature.

4.1 BCS

In the algorithm proposed by Briatico, Ciuffoletti, and Simoncini (BCS) [11], every process maintains and propagates an index idx that is similar to a logical clock [33]. Process p_i initializes idx_i to 0 and increments it after a basic checkpoint is taken. When p_i sends a message, it piggybacks idx_i onto it. When p_i receives a message m with $idx_m > idx_i$, it takes a forced checkpoint (Figure 3).



(a) p_i does not take a forced checkpoint

(b) p_i must take a forced checkpoint

Figure 3: BCS

BCS is very simple and efficient. However, its performance is strongly coupled to the policy adopted to take basic checkpoints as can be seen in Figure 4. In the none-faster

scenario, BCS takes a low number of checkpoints even when the number of processes increases. In the one-faster scenario, there is a considerable increase in the number of forced checkpoints.

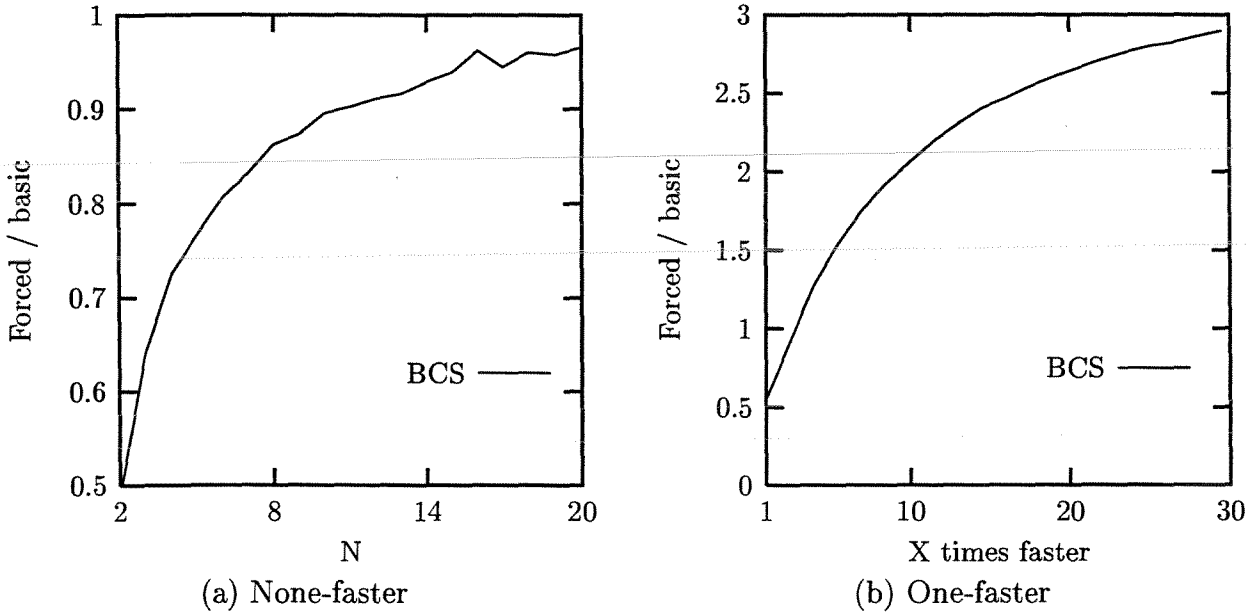


Figure 4: BCS

Figure 5 illustrates the effect of asymmetry; process p_0 takes checkpoints in a higher rate and induces the other processes to take forced checkpoints.

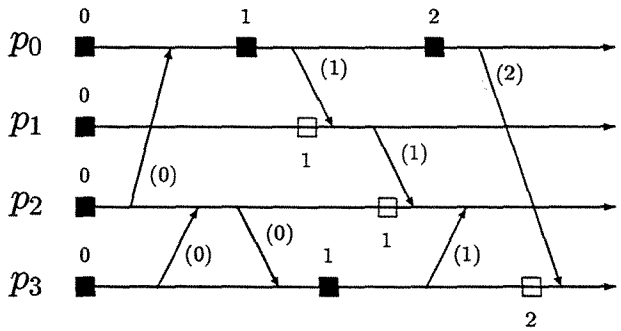
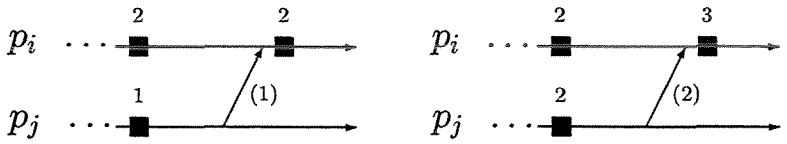


Figure 5: A scenario running BCS

4.2 Dealing with asymmetry

Let us consider a checkpoint interval in which a process p_i has only received messages with indexes that are smaller than its current index (Figure 6 (a)). Process p_i can deduce

that it is ahead and not increment its index when the next checkpoint is taken. On the contrary, if p_i has received at least one message with the same index, p_i must increment its index when it takes the next basic checkpoint (Figure 6 (b)). In order to implement this behavior a process needs to maintain a flag that indicates whether a message with an equal index has arrived in the current checkpoint interval. This optimization, called *Lazy-BCS*, reduces the impact of asymmetry and guarantees the absence of the domino effect [9, 29].

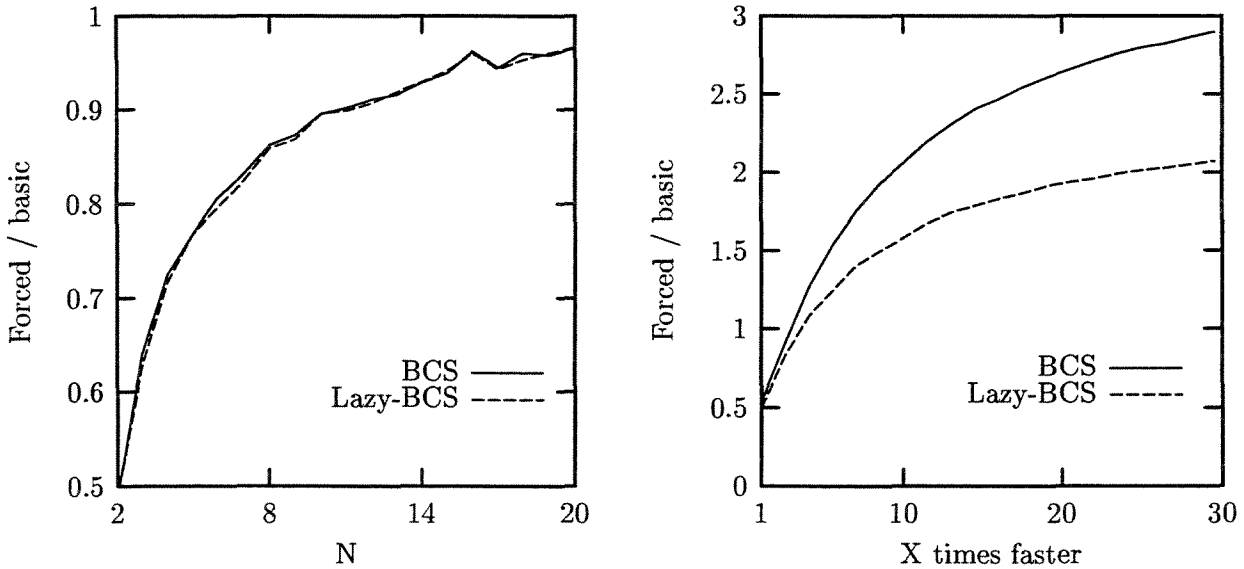


(a) p_i does not increment its index

(b) p_i must increment its index

Figure 6: Lazy-BCS

Figure 7 compares the behavior of BCS and Lazy-BCS. In the none-faster scenario, the difference is very small. Since the processes' indexes are almost synchronized, it is frequent that a process receives a message with an equal index and the Lazy-BCS optimization cannot be applied. In the one-faster scenario, there is a higher chance that a process can apply the Lazy-BCS optimization and we can see a reduction in the number of forced checkpoints.



(a) None-faster

(b) One-faster

Figure 7: Lazy-BCS

Figure 8 presents the same scenario depicted in Figure 5, this time running Lazy-BCS. Now, despite the fact that p_0 is taking basic checkpoints faster than the others, it labels the checkpoints with slower growing indexes. This lazy behavior allows p_3 to save a forced checkpoint.

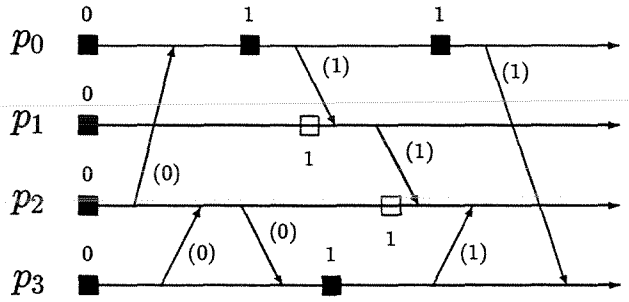
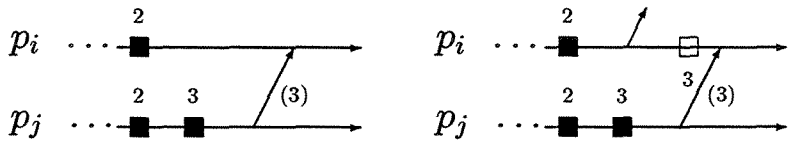


Figure 8: A scenario running Lazy-BCS

4.3 Waiting for the first send event

Let us consider an interval in which a process p_i has not sent any message at the time it receives a message with a greater index (Figure 9). Since p_i has not propagated the index of the current interval, it can increase its index without taking a forced checkpoint. We call this approach *aftersend* and it is domino-effect free. In order to implement this behavior a process needs to maintain a flag that indicates whether a message has been sent. This optimization has been incorporated to many index-based protocols [9, 29] and has also appeared in the context of checkpointing protocols that enforce Rollback-Dependency Trackability [50]. We have applied this optimization to BCS and to Lazy-BCS.



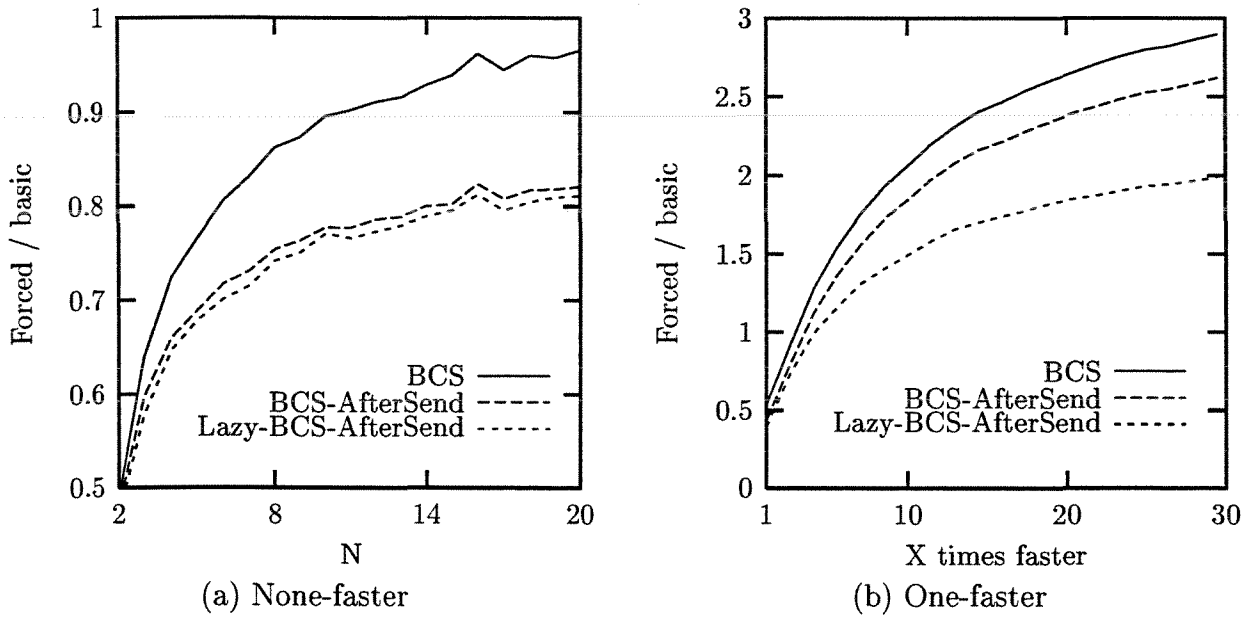
(a) p_i does not take a forced checkpoint

(b) p_i must take a forced checkpoint

Figure 9: BCS-Aftersend

Figure 10 compares the behavior of BCS, BCS-Aftersend and Lazy-BCS-Aftersend. In the none-faster scenario, both BCS-Aftersend and Lazy-BCS-Aftersend present a consid-

erable reduction in the number of forced checkpoint in comparison to BCS. This reduction shows that aftersend is effective even in the best scenario for BCS. In the one-faster scenario, BCS-Aftersend still presents a reduction of forced checkpoints in comparison to BCS, but Lazy-BCS-Aftersend presents a greater reduction.



(a) None-faster (b) One-faster

Figure 10: Waiting for the first send event

Figure 11 presents the same scenario depicted in Figure 5, this time running BCS-Aftersend. Since p_1 has not sent any message when it receives a message with a greater index, it saves a forced checkpoint.

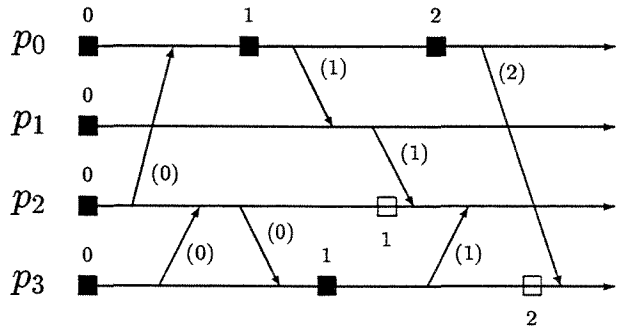


Figure 11: A scenario running BCS-Aftersend

Figure 12 presents the same scenario running Lazy-BCS-Aftersend and both p_1 and p_3 can save a forced checkpoint. This scenario illustrates the fact that Lazy-BCS-Aftersend combines the positive effects of the two presented optimizations.

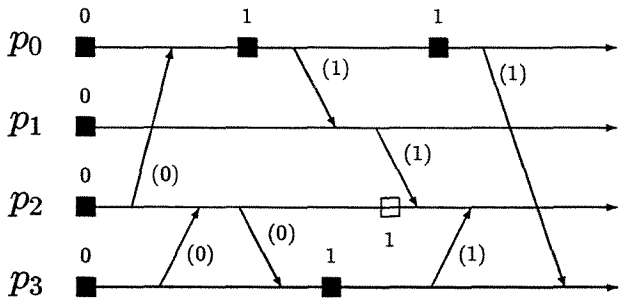


Figure 12: A scenario running Lazy-BCS-Aftersend

4.4 BQF

An optimization proposed by Baldoni, Quaglia and Fornara (BQF) [9] can be seen as a variant of Lazy-BCS-Aftersend. Let us assume that a process p_i has received a message from p_j with an equal index. In this situation, Lazy-BCS induces p_i to increment its index when the next checkpoint is taken. However, BQF postpones the decision to assign an index to a checkpoint (Figure 13 (a)). If before the first send-message event of the next interval p_i receives another message from p_j indicating that p_j has taken another checkpoint without increasing its index, p_i does not need to increase its index (Figure 13 (b)). If p_i sends a message without receiving such a message from p_j, p_i must increase its index (Figure 13 (c)). This optimization requires the propagation of an extra vector of n integers, where n is the number of processes in the computation.

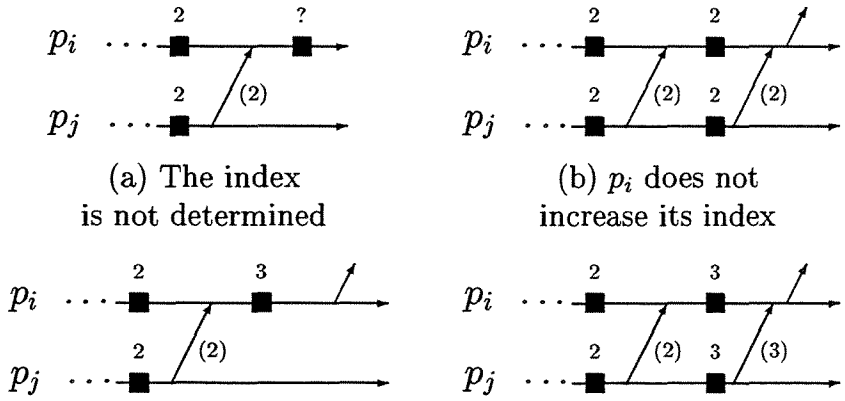
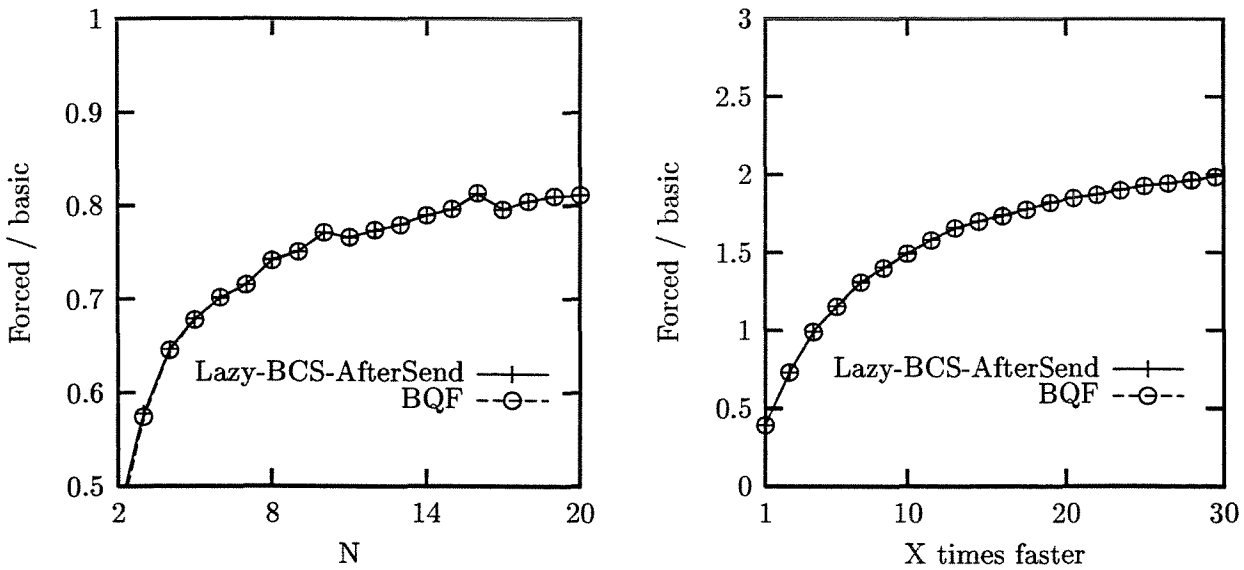


Figure 13: BQF

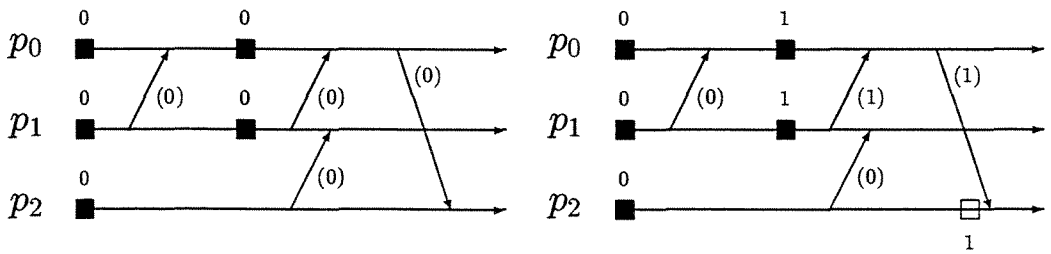
The protocol as described in [9] includes also another optimization of BCS that was first proposed by Manivannan and Singhal [35]. Using this optimization, a process skips a basic checkpoint if at least one forced checkpoint has been taken after its last basic checkpoint. This optimization is intrusive because it interferes with the application's policy of taking basic checkpoints and, thus, it cannot be under imposed to all applications. Thus, we have not considered this optimization in our experiments.

Figure 14 compares the behavior of Lazy-BCS-Aftersend and BQF. Even though BQF uses a much more expensive control information, it does not present a noticeable reduction in the number of forced checkpoints. These experimental results indicate that the configurations in which BQF would save a forced checkpoint in comparison to Lazy-BCS-Aftersend seem to be rare. For example, in the scenario presented in Figure 12 BQF would take the same forced checkpoint and propagate the same indexes. Figure 15 presents a scenario in which BQF can save a forced checkpoint in comparison to Lazy-BCS-Aftersend.



(a) None-faster (b) One-faster

Figure 14: BQF



(a) BQF (b) Lazy-BCS-Aftersend

Figure 15: BQF saves a forced checkpoint in comparison to Lazy-BCS-Aftersend

5 Conclusion

The literature on checkpointing protocols is characterized for: (i) a huge number of protocols and protocol optmizations, and (ii) proliferation of non-systematic approachs to their comparison. The result of this haphazard growth of literature is loss of perspective on the field; it becomes very difficult to assess and understand these protocols one in relation to each other. This paper has taken the problem of assessing index-based checkpointing protocols as an example that the remedy to the situation is based on a very simple recipe: (i) separation of the basic protocol from the optmizations that can be applied onto it and (ii) modular, incremental presentation of each of the protocol variants (produced by modular, incremental application of the optimizations on the basic protocol), supported by simulation results.

Index-based checkpointing [9, 11, 17, 28, 35, 36] allows simple and efficient algorithms for domino-effect free construction of recovery lines. In this paper, we have presented a performance study of the well-known algorithm proposed by Briatico, Ciuffoletti, and Simoncini and have explored the impact of some optimizations of this algorithm presented in the literature.

The ideal result in checkpointing research would be to find the index-based algorithm that requires the minimum number of forced checkpoints, outperforming the other algorithms in all possible scenarios. Tsai, Kuo and Wang [46] have proved that it is impossible to develop such an optimum index-based algorithm. Thus, simulation studies are very important to give a perspective of the performance of these algorithms.

We have used the simulation toolkit Metapromela [48] to analyze the behavior of BCS and its optimizations. We have analyzed two simple optimizations of BCS, Lazy-BCS and BCS-Aftersend, that have exhibited a reduction of the number of forced checkpoints. The combination of these two optimizations produced a very effective algorithm, Lazy-BCS-Aftersend, that does not requires the propagation of extra control information.

We have also analyzed an optimization proposed by Baldoni, Quaglia and Fornara [9], that can be seen as a variant of Lazy-BCS-Aftersend. That optimization requires the propagation of an extra vector of n integers, where n is the number of processes in the computation. Surprisingly enough our systematic approach has indicated that BQF does not present a noticeable reduction in the number of forced checkpoints in comparison to Lazy-BCS-Aftersend. Since Lazy-BCS-Aftersend is very simple and efficient, we conclude that it is the best algorithm among the ones analyzed in this paper.

Capítulo 9

A Linear Approach to Enforce the Minimal Characterization of the Rollback-Dependency Trackability Property*

Islene Calciolari Garcia

Luiz Eduardo Buzato

Abstract

A checkpointing protocol that enforces rollback-dependency trackability (RDT) during the progress of a distributed computation must take forced checkpoints to *break* non-trackable dependencies. Breaking just non-visibly doubled dependencies instead of breaking all non-trackable dependencies leads to fewer forced checkpoints, but seemed to require the processes of a computation to maintain and propagate $O(n^2)$ control information. In this paper, we prove that this hypothesis is false by presenting a protocol that breaks the minimal set of non-visibly doubled dependencies necessary to enforce RDT, called "non-visibly doubled PMM-paths", using only $O(n)$ control information.

Keywords: fault-tolerance, rollback recovery, distributed checkpointing, distributed algorithms, algorithm complexity.

*Este artigo foi enviado para avaliação em uma conferência internacional da área.

1 Introduction

A checkpointing protocol that enforces rollback-dependency trackability (RDT) during the progress of the computation must take forced checkpoints to *break* non-trackable dependencies [3, 50]. Although it is not possible to design an RDT protocol that will take the minimum number of forced checkpoints for all checkpoint and communication patterns [45], RDT protocols based on stronger induction conditions usually take fewer forced checkpoints than RDT protocols based on weaker conditions [7]. The inconvenience of the strongest approach, based on breaking only non-visibly doubled paths, was that it seemed to require $O(n^2)$ control information [7], while a weaker approach based on breaking all non-trackable requires only $O(n)$ control information [50], where n is the number of processes in the computation. The main contribution of this paper is to present a simple RDT protocol that implements the stronger approach in $O(n)$.

A checkpoint is a recording in stable memory of a process' state that can be used for rollback recovery. The set of all checkpoints taken by a distributed computation and the dependencies established among these checkpoints due to message exchanges form a checkpoint and communication pattern (CCP). CCPs that satisfy RDT present only checkpoint dependencies that are on-line trackable using dependency vectors, and allow efficient solutions to the determination of the maximum and minimum consistent global checkpoints that include a set of checkpoints [50]. Many applications can benefit from these algorithms: rollback recovery, software error recovery, and distributed debugging [50].

Netzer and Xu have determined that checkpoint dependencies are created by sequences of messages called *zigzag paths* [38]. Two types of zigzag paths can be identified: causal paths (C-paths) and non-causal paths (Z-paths). C-paths are on-line trackable through the use of dependency vectors; Z-paths, on the contrary, cannot be on-line tracked. However, a CCP may present Z-paths and still satisfy RDT. In this case, all Z-paths must be *doubled* by a C-path; a Z-path is doubled by a causal one if the pair of checkpoints related by that Z-path is also related by a C-path [6, 7].

Baldoni, Helary and Raynal have established properties that could reduce the set of Z-paths that must be doubled in a CCP that satisfies RDT. They have concentrated their study on visible properties, that is, properties that can be tested on-line by an RDT protocol [6]. They have also proved that a process does not need to break a Z-path if it is able to detect that it is already causally doubled (a *visibly doubled* path). Additionally, they have conjectured that a specific set of Z-paths, named "non-visibly-doubled-EPSCM-paths", determines the smallest set of Z-paths that must be tested for breaking by an RDT protocol [6]. Based on this set, they have proposed an RDT protocol that enforces this characterization using $O(n^2)$ control information, claiming that this protocol is optimal with respect to the size of control information [7].

Recently, we have proved that their conjecture was false and the set of Z-paths that must be tested for breaking by an RDT protocol can be further reduced to the set of "non-visibly-doubled-PMM-paths" [23]. In this paper, extending the approach used to prove the conjecture false, we describe a protocol that enforces this minimal characterization of RDT requiring only $O(n)$ control information.

This paper is structured as follows. Section 2 describes the computational model adopted. Section 3 introduces rollback-dependency trackability. Section 4 describes a quadratic approach to enforce the minimal characterization of RDT, similar to the one suggested by Baldoni, Helary, Mostefaoui, and Raynal [3, 7]. Section 5 presents a linear approach to enforce the minimal characterization of RDT. Section 6 concludes the paper.

2 Computational model

A distributed computation is composed of n sequential processes $\{p_0, \ldots, p_{n-1}\}$ that communicate only by exchanging messages. Messages cannot be corrupted, but can be delivered out of order or lost. The local history of a process p_i is modeled as a possibly infinite sequence of events $(e_i^1, e_i^2, \ldots)$, divided into internal events and communication events.

A checkpoint is an internal event that records the process' state in stable memory. Each process takes an initial checkpoint immediately after execution begins and a final checkpoint immediately before execution ends. Let c_i^γ denote the γth checkpoint taken by p_i. Two successive checkpoints $c_i^{\gamma-1}$ and c_i^γ, $\gamma > 0$, define a checkpoint interval I_i^γ. An event e_i^ι belongs to I_i^γ ($e_i^\iota \in I_i^\gamma$) if it occured in p_i after $c_i^{\gamma-1}$, but not after c_i^γ. Figure 1 illustrates a checkpoint interval I_i^γ and an event e_i^ι that belongs to I_i^γ.

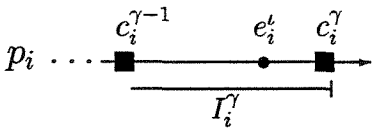


Figure 1: A checkpoint interval

The set of all checkpoints taken by a distributed computation and the dependencies established among these checkpoints due to message exchanges form a checkpoint and communication pattern (CCP). Figure 2 illustrates a CCP using a space-time diagram [33] augmented with checkpoints (black squares).

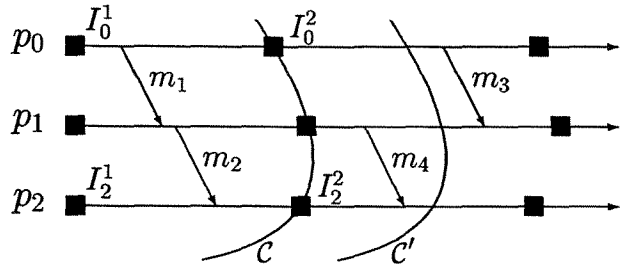


Figure 2: A distributed computation

2.1 Consistency

The concept of causal precedence is fundamental for a better understanding of consistency [33].

Definition 2.1 Causal precedence—*Event* e_a^α *causally precedes* e_b^β ($e_a^\alpha \rightarrow e_b^\beta$) *if (i)* $a = b$ *and* $\beta = \alpha + 1$*; (ii)* $\exists m : e_a^\alpha = send(m)$ *and* $e_b^\beta = receive(m)$*; or (iii)* $\exists e_c^\gamma : e_a^\alpha \rightarrow e_c^\gamma \wedge e_c^\gamma \rightarrow e_b^\beta$.

A cut of a distributed computation contains a prefix of each of the proceses' local histories. A consistent cut is left-closed under causal precedence and defines an instant in a distributed computation [16]. If a cut $\mathcal{C} \subset \mathcal{C}'$, we can say that $\mathcal{C}$ is in the past of $\mathcal{C}'$ (Figure 2).

Definition 2.2 Consistent Cut—*A cut* $\mathcal{C}$ *is consistent if, and only if,*

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

A consistent global state is formed by the states of each process in the fronteir of a consistent cut [16]. The set of consistent global checkpoints is a subset of the set of consistent global states. In Figure 2, $\mathcal{C}$ is related to a consistent global checkpoint, but $\mathcal{C}'$ is not.

2.2 Zigzag paths

Netzer and Xu have determined that checkpoints that are part of the same consistent global checkpoint cannot be related by sequences of messages called *zigzag paths* [38].

Definition 2.3 Zigzag path—*A sequence of messages* $\mu = [m_1, \ldots, m_k]$ *is a zigzag path from* I_a^α *to* I_b^β *if (i)* p_a *sends* m_1 *after* $c_a^{\alpha-1}$*; (ii) if* m_i, $1 \leq i < k$, *is received by* p_c, *then* m_{i+1} *is sent by* p_c *in the same or a later checkpoint interval; (iii)* m_k *is received by* p_b *before* c_b^β.

Two types of zigzag paths can be identified: (i) causal paths (C-paths) and (ii) non-causal paths (Z-paths). A zigzag path is causal if the reception of m_i, $1 \leq i < k$, causally precedes the send event of m_{i+1}. In Figure 2, $[m_1, m_2]$ is a C-path from I_0^1 to I_2^1 and $[m_3, m_4]$ is a Z-path from I_0^2 to I_2^2. A Z-path that starts in a checkpoint interval and finishes in a previous checkpoint interval of the same processes is a Z-cycle and identifies a useless checkpoint, that is, a checkpoint that cannot be part of any consistent global checkpoint [38]. Figure 3 presents a Z-cycle $[m_1, m_2, m_3]$ and a useless checkpoint c_i^γ.

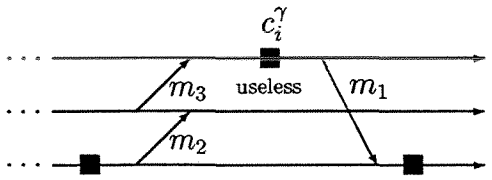


Figure 3: A Z-cycle

3 Rollback-Dependency Trackability

The literature presents two approaches to define RDT. The first one is based on the study of on-line trackable dependencies, implemented through the use of dependency vectors [50]; the other one is based on the study of sequence of messages [3, 6, 7, 23].

3.1 Dependency vectors

A transitive dependency tracking mechanism can be used to capture causal dependencies among checkpoints. Each process maintains and propagates a size-n dependency vector. Let dv_i be the dependency vector of p_i, $m.dv$ be the dependency vector piggybacked on a message m, and $dv(c)$ be the dependency vector associated to a checkpoint c. All entries of dv_i are initialized to 0. The entry $dv_i[i]$ represents the current interval of p_i and it is incremented immediately after a checkpoint (including the initial one). Every other entry $dv_i[j]$, $j \neq i$, represents the highest interval index of p_j that p_i has knowledge about and it is updated using a component-wise maximum every time a message m with a greater value of $m.dv[j]$ arrives to p_i. Figure 4 depicts the dependency vectors established during a distributed computation.

Note in Figure 4 that $dv(c_2^1)$ is (1, 1, 1) and it correctly captures all zigzag paths that reach I_2^1. Unfortunately, not all dependencies can be tracked on-line. For example, $dv(c_2^2)$ is $(1, 2, 2)$ and it does not capture the zigzag path from I_0^2 to I_2^2.

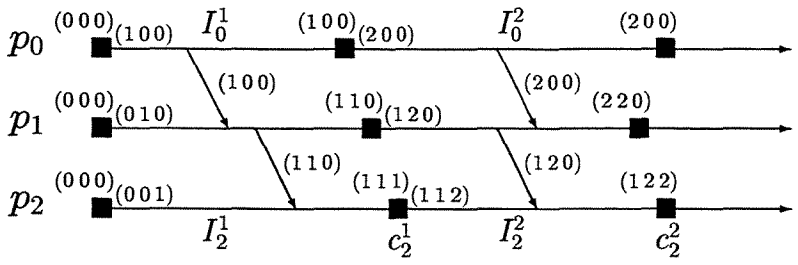


Figure 4: A distributed computation with dependency vectors

Definition 3.1 On-line trackability—*A zigzag path from I_a^α to I_b^β is on-line trackable through the use of dependency vectors if $dv(c_b^\beta)[a] \geq \alpha$.*

Definition 3.2 Dependency vector characterization of RDT—*A checkpoint pattern satisfies RDT if all zigzag paths are on-line trackable.*

RDT is a desirable property because efficient algorithms can be used to construct consistent global checkpoints if all zigzag paths are on-line trackable. Also, an RDT checkpoint pattern does not admit useless checkpoints [50].

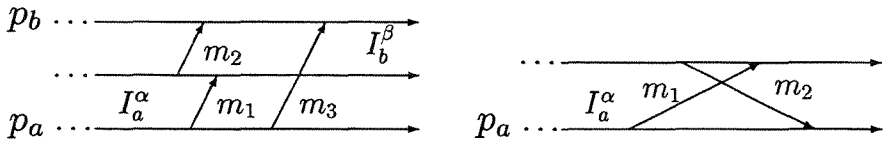
3.2 Causal doubling

A CCP may present Z-paths and satisfy RDT if all Z-paths are *doubled* by a C-path [6, 7].

Definition 3.3 Causal doubling—*A Z-path from I_a^α to I_b^β is* causally doubled *if there is a C-path μ from I_a^α to I_b^β or $a = b$ and $\alpha \leq \beta$.*

Definition 3.4 Message-based characterization of RDT—*A checkpoint pattern satisfies RDT if all Z-paths are causally doubled.*

A Z-path can be doubled by a causal one if the pair of checkpoints related by that Z-path is also related by a C-path [6, 7]. Another possibility for a Z-path from I_a^α to I_b^β to be doubled is if it starts and finishes in the same process and I_b^β does not precede I_a^α. In Figure 5 (a), $[m_1, m_2]$ is causally doubled by m_3 and in Figure 5 (b), $[m_1, m_2]$ is trivially doubled due to the execution of p_a.



(a) $[m_1, m_2]$ is doubled by m_3

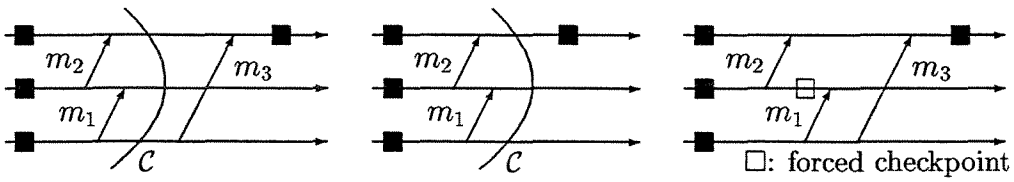
(b) $[m_1, m_2]$ is trivially doubled

Figure 5: Causal doubling

3.3 RDT protocols

A communication-induced checkpointing protocol that enforces RDT allows processes to take checkpoints asynchronously, but they may be induced by the protocol to take forced checkpoints in order to break non-trackable dependencies [3, 50]. Forced checkpoints must be taken upon the arrival of a message, but before this message is processed by the computation. The decision to take a forced checkpoint must be based only on the local knowledge of a process; there are no control messages, no global knowledge or knowledge about the future of the computation. These assumptions impose some restrictions on the set of CCPs that can be produced by RDT protocols.

The CCP depicted in Figure 6 (a), for example, would never have been produced by an RDT procotol. This pattern has a Z-path $[m_1, m_2]$ that is doubled by message m_3 in the future of a consistent cut $\mathcal{C}$. At $\mathcal{C}$, the processes of the computation cannot rely on the existence of m_3, since a scenario such as the one depicted in Figure 6 (b) could have happened, producing a CCP that does not satisfy RDT. Under an RDT protocol, the CCP presented in Figure 6 (a) should present at least one forced checkpoint (Figure 6 (c)).



(a) (b) (c)

Figure 6: The behavior of RDT protocols

The RDT property must be enforced in every consistent cut of a computation that runs an RDT protocol. This observation has lead us to introduce the concept of left-doubling [23].

3.4 Left-doubling

A C-path μ belongs to a consistent cut $\mathcal{C}$ if the reception of the last message of μ belongs to $\mathcal{C}$. A Z-path ζ can be seen as a concatenation of ℓ C-paths $\mu_1 \cdot \mu_2 \cdot \ldots \cdot \mu_\ell$ and ζ belongs to a consistent cut $\mathcal{C}$ if all causal components of ζ belong to $\mathcal{C}$.

Definition 3.5 Left-doubling—*A Z-path ζ is left-doubled in relation to a consistent cut $\mathcal{C}$ if (i) ζ belongs to $\mathcal{C}$ and (ii) ζ is doubled by a C-path μ that also belongs to $\mathcal{C}$.*

A consistent cut $\mathcal{C}$ satisfies the RDT property if, and only if, all Z-paths that belong to $\mathcal{C}$ are left-doubled. Using the concept of left-doubling, we have proved that a protocol that breaks all "non-visibly doubled PMM-paths" must enforce RDT [23].

3.5 The minimal characterization of RDT

Definition 3.6 PMM-path—*A PMM-path is a Z-path composed of a **p**rime **m**essage m_1 and a **m**essage m_2.*

A message m from I_k^κ to p_i is *prime* if m is the first message received by p_i that brings information about I_k^κ. Figure 7 presents a PMM-path $[m_1] \cdot [m_2]$ from I_k^κ to I_j^γ.

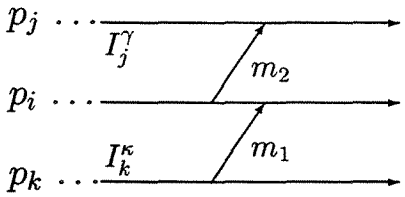


Figure 7: A PMM-path

Definition 3.7 Visibly Doubled PMM-path—*A PMM-path $[m_1] \cdot [m_2]$ is visibly doubled if (i) is causally doubled by a C-path μ and (ii) the reception of the last message of μ causally precedes the sending of m_1.*

Figure 8 presents a visibly doubled PMM-path $[m_1] \cdot [m_2]$ from I_k^κ to I_j^γ. We should note that $[m_1] \cdot [m_2]$ is left-doubled in relation to any consistent cut of the computation, since any consistent cut that contains m_1 should also contain μ. The set of "non-visibly-doubled PMM-paths" characterizes the minimal set of Z-paths that be must tested for breaking by an RDT protocol [23].

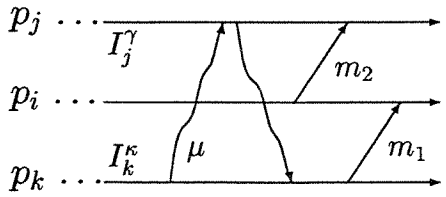


Figure 8: A visibly doubled PMM-path

Definition 3.8 The minimal characterization of RDT—*A CCP satisfies the RDT property if all PMM-paths are visibly doubled.*

In the following sections, we are going to focus on the problem of implementing an RDT protocol that enforces this minimal characterization.

4 A quadratic approach

The core of a protocol that enforces the minimal characterization of RDT lies on the detection of non-visibly doubled PMM-paths by a process p_i. Let us consider a PMM-path $[m_1] \cdot [m_2]$ from I_k^κ to I_j^γ such that m_1 is received by p_i after the sending of m_2 (Figure 7). Before processing m_1, p_i must detect the establishment of this PMM-path and verify whether it is visibly doubled (Figure 8). If $[m_1] \cdot [m_2]$ is visibly doubled, m_1 can be processed immediately. Otherwise, p_i must take a forced checkpoint before processing m_1.

4.1 Detecting PMM-paths

In order to detect all PMM-paths formed upon the reception of a message, process p_i must record for what processes it has sent messages during the current interval. To do this, p_i maintains a vector of booleans $sent\_to_i$, such that all entries of $sent\_to_i$ are set to `false` when p_i takes a checkpoint, and an entry $sent\_to_i[j]$ is set to `true` when p_i sends a message to p_j. A PMM-path is detected by p_i upon the reception of a message m from p_k when the following condition holds:

$$\exists\, j : sent\_to_i[j] \wedge m.\, dv[k] > dv_i[k]$$

4.2 Detecting non-visibly doubled PMM-paths

The detection of whether $[m_1] \cdot [m_2]$ from I_k^κ to I_j^γ is visibly doubled by a C-path μ can be divided into two cases:

I. From the point of view of p_i, the interval I_j^γ is in the past of p_j

Figure 9 shows a scenario in which p_i receives knowledge that m_2 was received during I_j^γ, but μ was received during $I_j^{\gamma+1}$. Since a C-path μ from I_k^κ to $I_j^{\gamma+1}$ does not double a PMM-path from I_k^κ to I_j^γ, process p_i must take a forced checkpoint before processing m_1.

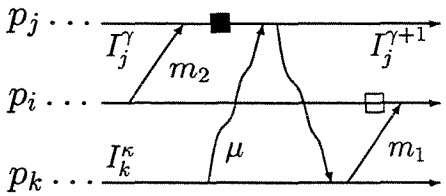


Figure 9: I_j^γ is in the past of p_j

To detect visibly doubled PMM-paths, p_i must evaluate (i) whether m_2 was received by p_j and in which checkpoint interval, and (ii) whether p_j has received knowledge about I_k^κ and in which checkpoint interval. For p_i to be able to answer these questions, the processes of the computation would have to maintain and propagate an **unbounded** amount of control information, proportional to the number of messages and checkpoint intervals.

II. From the point of view of p_i, the interval I_j^γ is not in the past of p_j

Figure 10 presents three scenarios to show that the problem of detecting visibly doubled paths is much easier when in p_i's view I_j^γ is not in the past of p_j. In Figure 10 (a), p_i receives knowledge that both m_2 and μ were received during I_j^γ. In Figures 10 (b, c), p_i receives knowledge that μ was received by p_j, but p_i does not receive knowledge about the reception of m_2. In these cases, the existence of a C-path μ from I_k^κ to p_j guarantees to p_i that $[m_1] \cdot [m_2]$ is causally doubled.

Fortunately, there is an approach to handle case **I** that requires only $O(n)$ control information, as explained in next Section. Section 4.4 shows an $O(n^2)$ approach to handle case **II**. In Section 5, we are going to prove that case **II** can also be handled in $O(n)$.

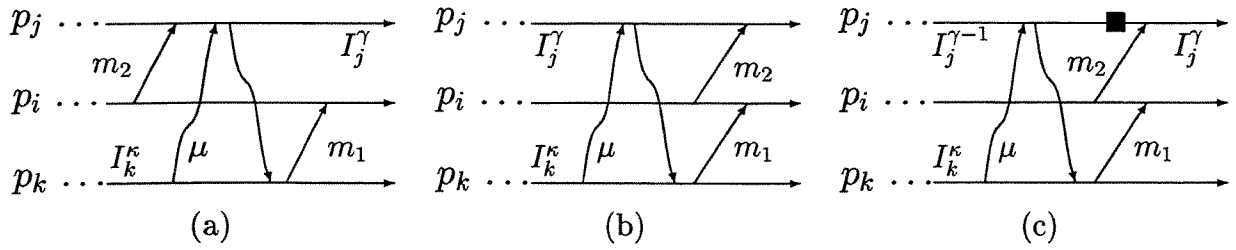


Figure 10: I_j^γ is not in the past of p_j

4.3 Process p_i knows that I_j^γ is in the past

Let us assume that, upon the reception of m_1, p_i knows that m_2 was received in an interval that is on the past of p_j. In Figure 11 (a), p_i receives knowledge about $I_j^{\gamma+1}$ due a C-path ν that arrives to p_i before m_1. The concatenation of ν and m_2 forms a Z-cycle. Since the RDT property does not allow Z-cycles, p_i should have taken a forced checkpoint before processing the last message of ν. The information about $I_j^{\gamma+1}$ could have arrived with m_1. In Figure 11 (b), p_k receives knowledge about $I_j^{\gamma+1}$ due a C-path ν that arrives to p_k before the sending of m_1. The concatenation of ν, m_1 and m_2 also forms a Z-cycle and p_i should have taken a forced checkpoint before processing m_1.

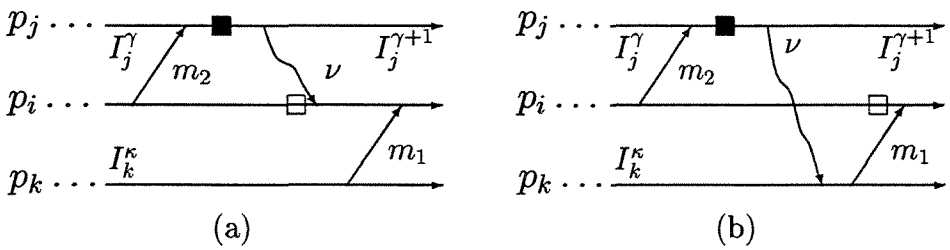


Figure 11: Process p_i knows that I_j^γ is in the past

The Z-cycle $[\nu] \cdot [m_2]$ of Figure 11 (a) and the Z-cycle $[\nu \cdot m_1] \cdot [m_2]$ of Figure 11 have only two causal components. Z-cycles formed by two causal components $[\nu_1] \cdot [\nu_2]$ and are called CC-cycles (Figure 12). CC-cycles can be easily detected and breaked on-line if p_i takes a forced checkpoint before processing the last message of ν_1. Process p_i must take the forced checkpoint only if $[\nu_2] \cdot [\nu_1]$ "contains" a checkpoint, that is, it is not *simple* [3, 7].

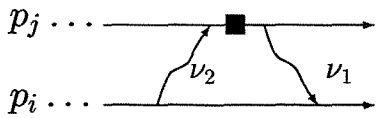


Figure 12: A CC-cycle $[\nu_1] \cdot [\nu_2]$

To keep track of simple paths, each process maintains and propagates a size-n boolean vector *simple*. Let $simple_i$ be the vector maintained by p_i, and $m.simple$ be the vector piggybacked on a message m. The entry $simple_i[i]$ is always `true`, and the entries $simple_i[k]$, $k \neq i$, are reset to `false` when p_i takes a checkpoint. When p_i receives a message m, each entry $simple_i[k]$ is updated as follows:

$$\texttt{if } m.dv[k] > dv_i[k] \texttt{ then } simple_i[k] \leftarrow m.simple[k]$$

$$\texttt{if } m.dv[k] = dv_i[k] \texttt{ then } simple_i[k] \leftarrow simple[k] \wedge m.simple[k]$$

Process p_i detects a CC-cycle upon the reception of m using the following condition:

$$m.dv[i] = dv_i[i] \wedge m.simple[i] = \texttt{false}$$

In the scenarios of Figure 11, the second causal component of the CC-cycle is represented by a single message m_2. However, keeping track of only CC-cycles of the form $[\nu][m]$ would increase the complexity of the required control information due to the propagation of knowledge about single messages. Also, as an RDT protocol must break all CC-cycles, using the above condition does not increase the number of forced checkpoints.

4.4 Tracking C-paths from I_k^κ to p_j

Even if a process p_i breaks CC-cycles, PMM-paths such as the ones described in case **II** of Section 4.2 (Figure 10) needed to be tested for breaking. Thus, if p_i has sent a message m_2 to p_j and receives a prime message m_1 from I_k^κ, p_i must verify whether there is a C-path μ from I_k^κ to p_j. We should note that before the reception of m_1, process p_i cannot have information about μ, otherwise m_1 would not be prime (Figure 13). Thus, the information about μ can only arrive on the control information piggybacked on m_1.

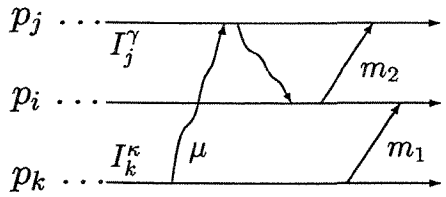


Figure 13: Message m_1 is not prime

According to the definition of dependency vectors, $dv_i[k]$ is the greatest interval index of p_k that p_i has knowledge about. Let us consider a matrix of booleans $causal_i$ such that each entry $causal_i[k][j]$ indicates whether, up to the knowledge of p_i, there is a C-path from $I_k^{dv_i[k]}$ to p_j. The entries on the diagonal of $causal_i$ are always `true`, since there is a trivial causal flow from a process to itself. Every other entry $causal_i[k][j], k \neq j$ is initialized to `false`. When p_i takes a checkpoint, all entries $causal_i[i][j], i \neq j$ are reset to `false`, indicating that there is no C-path from this new interval. Let $m.causal$ be a matrix piggybacked on a message m. When p_i receives m from p_s, $causal_i$ is updated as follows:

$\forall k$, if $m.dv[k] > dv_i[k]$ then $\forall l : causal_i[k][l] \leftarrow m.causal[k][l]$

$\forall k$, if $m.dv[k] = dv_i[k]$ then $\forall l : causal_i[k][l] \leftarrow causal_i[k][l] \vee m.causal[k][l]$

$causal_i[s][i] \leftarrow$ `true`

$\forall l : causal_i[l][i] \leftarrow causal_i[l][i] \vee causal_i[l][s]$

4.5 Checkpoint induction condition

A forced checkpoint is induced by p_i upon the reception of a prime message m from I_k^κ if (i) there is a CC-cycle or (ii) there is a PMM-path from p_k to p_j, but there is no C-path from I_k^κ to p_j:

$$\begin{array}{ll} \text{(i)} & (m.dv[i] = dv_i[i] \wedge m.simple[i] = \texttt{false}) \vee \\ \text{(ii)} & (\exists j : sent\_to_i[j] \wedge m.dv[k] > dv_i[k] \wedge \neg m.causal_i[k][j]) \end{array}$$

The approach presented in this Section is similar to the one presented by Baldoni, Helary, Mostefaoui and Raynal, although their protocols break more complex Z-paths [3, 7]. Their approach requires $O(n^2)$ control information since it tracks the existence of C-paths from every process p_k to every process p_j of the computation. Baldoni, Helary, and

Raynal claim that $O(n^2)$ is optimal with respect to the size of the control information [7]. In the next Section, we are going to show an $O(n)$ RDT protocol that breaks visibly-doubled Z-paths.

5 A linear approach

Linearity of the control information comes as a result of two observations: (i) we do not need to keep track of C-paths from I_k^κ to p_j, instead, we are going to take advantage of dependency vector restrictions imposed by an RDT protocol and (ii) we can perform comparison operations on dependency vectors as a whole instead of keeping track of single entries, as in Definition 3.1. This alternative approach is the key to the complexity reduction. Thus, let us define the following comparison operations:

$$dv(c_j^\gamma) \geq dv(c_k^\kappa) \quad \Leftrightarrow \quad \forall i, 0 \leq i < N, \quad dv(c_j^\gamma)[i] \geq dv(c_k^\kappa)[i]$$
$$dv(c_j^\gamma) = dv(c_k^\kappa) \quad \Leftrightarrow \quad \forall i, 0 \leq i < N, \quad dv(c_j^\gamma)[i] = dv(c_k^\kappa)[i]$$

5.1 Dependency vector restrictions under RDT

Let us begin with dependency vector restrictions that must hold for all CCPs, not only on CCPs produced by RDT protocols.

Theorem 5.1 *Under RDT, the existence of a C-path μ from I_k^κ to I_j^γ guarantees that $dv(c_j^\gamma) \geq dv(c_k^\kappa)$.*

Proof: For the sake of contradiction, let us assume the existence of an entry l of $dv(c_j^\gamma)$ such that $dv(c_j^\gamma)[l] < dv(c_k^\kappa)[l] = \lambda$ (Figure 14). The information about I_l^λ must have arrived at p_k due to a C-path μ' and the last message of μ' must have been received after the sending of the first message of μ. The concatenation of μ and μ' forms a Z-path from I_l^λ to I_j^γ that is not on-line trackable (a non-causally doubled Z-path). □

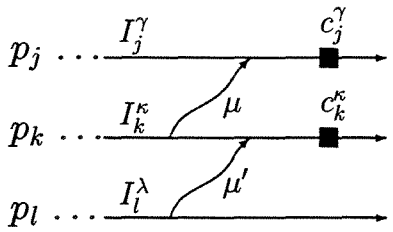


Figure 14: Contradiction hypothesis of Theorem 5.1

Corollary 5.2 *Under RDT, the existence of a C-path μ from I_k^κ to I_j^γ and of a C-path μ' from I_j^γ to I_k^κ guarantees that $dv(c_j^\gamma) = dv(c_k^\kappa)$.*

Proof: Due to μ, $dv(c_j^\gamma) \geq dv(c_k^\kappa)$. Due to μ', $dv(c_k^\kappa) \geq dv(c_j^\gamma)$. Thus, $dv(c_j^\gamma) = dv(c_k^\kappa)$. □

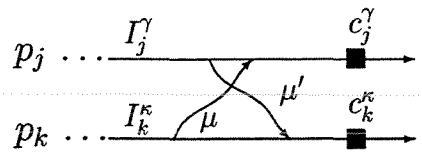


Figure 15: The existence of μ and μ' guarantees that $dv(c_j^\gamma) = dv(c_k^\kappa)$

5.2 Dependency vector restrictions under an RDT protocol

Since an RDT protocol must enforce RDT in every consistent cut of the computation, let us explore dependency vector restrictions during the progress of checkpoint intervals.

Theorem 5.3 *Let μ be a C-path μ from I_k^κ to I_j^γ and let $\mathcal{C}$ be a consistent cut that contains μ. Let $e_j^{\gamma'} \in I_j^\gamma$ and $e_k^{\kappa'} \in I_k^\kappa$ be the events of p_j and p_k that belong to the fronteir of $\mathcal{C}$. Under an RDT protocol, the following restriction should hold: $dv(e_j^{\gamma'}) \geq dv(e_k^{\kappa'})$.*

Proof: For the sake of contradiction, let us assume the existence of an entry l of $dv(e_j^{\gamma'})$ such that $dv(e_j^{\gamma'})[l] < dv(e_k^{\kappa'})[l] = \lambda$ (Figure 16). The information about I_l^λ must have arrived at p_k due to a C-path μ' and the last message of μ' must have been received after the sending of the first message of μ. The concatenation of μ and μ' forms a Z-path from I_l^λ to I_j^γ that is not left-doubled in relation to $\mathcal{C}$. □

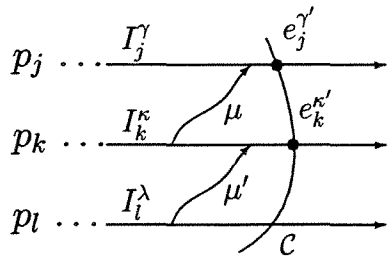


Figure 16: Contradiction hypothesis of Theorem 5.3

Corollary 5.4 *Let μ be a C-path μ from I_k^κ to I_j^γ and μ' be a C-path from I_j^γ to I_k^κ. Let $\mathcal{C}$ be a consistent cut that contains μ and μ'. Let $e_j^{\gamma'} \in I_j^\gamma$ and $e_k^{\kappa'} \in I_k^\kappa$ be the events of p_j and p_k that belong to the frontier of $\mathcal{C}$. Under an RDT protocol, the following restriction should hold: $dv(e_j^{\gamma'}) = dv(e_k^{\kappa'})$.*

Proof: Due to μ, $dv(e_j^{\gamma'}) \geq dv(e_k^{\kappa'})$. Due to μ', $dv(e_k^{\kappa'}) \geq dv(e_j^{\gamma'})$. Thus, $dv(e_j^{\gamma'}) = dv(e_k^{\kappa'})$. □

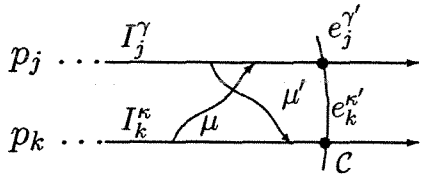


Figure 17: The existence of μ and μ' guarantees that $dv(e_j^{\gamma'}) = dv(e_k^{\kappa'})$

5.3 Process p_k knows that $dv_j = dv_k$

Let μ be a C-path from I_k^κ to I_j^γ and μ' be a C-path from I_j^γ to I_k^κ such that the last message of μ is received before the first message of μ' is sent (Figure 18). Upon the arrival of μ', p_k receives knowledge about μ and, according to Corollary 5.4, it is also able to conclude that $dv_j = dv_k$.

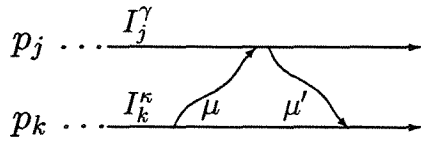


Figure 18: Process p_k knows that $dv_j = dv_k$

The following theorem shows that the verification of equal dependency vectors can replace the verification of the existence of C-paths.

Theorem 5.5 *Let I_k^κ be the current interval of a process p_k. Under an RDT protocol, p_k knows the existence of a C-path μ from I_k^κ to p_j if, and only if, to the knowledge of p_k, $dv_j = dv_k$.*

Proof:
(i) $dv_j = dv_k \Rightarrow$ a C-path μ from I_k^κ to p_j

Since $dv_j = dv_k$, $dv_j[k] = dv_k[k] = \kappa$ and there must be a C-path μ from I_k^κ to p_j.
(ii) a C-path μ from I_k^κ to $p_j \Rightarrow dv_j = dv_k$

Process p_k must have received knowledge about μ due to a C-path μ' from p_j to p_k. The first message of μ' must have been sent after the reception of the last of μ. Since an RDT protocol does not allow CC-cycles, there cannot be a checkpoint between the reception of the last of μ and the sending of first message of μ'. Thus, these two events occured in the same checkpoint interval and, according to Corollary 5.4, upon the reception of the last message of μ', p_k knows that $dv_j = dv_k$ (Figure 18).

Also, p_k will not be able to increase its dependency vector till the end of I_k^κ. For the sake of contradiction, let us assume that p_k receives information about I_l^λ through a C-path ν after the reception of the last message of μ'. According to Corollary 5.2, $dv(c_j^\gamma)$ should be equal to $dv(c_k^\kappa)$, and p_j must also receive information about I_l^λ through a C-path ν'. Let $\mathcal{C}$ be the minimum consistent cut that cointains μ', that is, the cut formed by the the reception of the last message of μ' and all the events that causally precede this reception (Figure 19). Thus, at $\mathcal{C}$, neither p_j nor p_k have knowledge about I_l^λ. From $\mathcal{C}$ is possible to construct a sequence of consistent cuts that reflect the progress of the computation, adding one event at a time. Either ν or ν' is going to be included first during the sequence. If ν is included first, we would have a consistent cut, say $\mathcal{C}'$, such that $[\nu] \cdot [\mu]$ is not left-doubled in relation to $\mathcal{C}'$ (Figure 19). Analogously, if ν' is included first, we would have a consistent cut such that $[\nu'] \cdot [\mu']$ is not left-doubled in relation to it. □

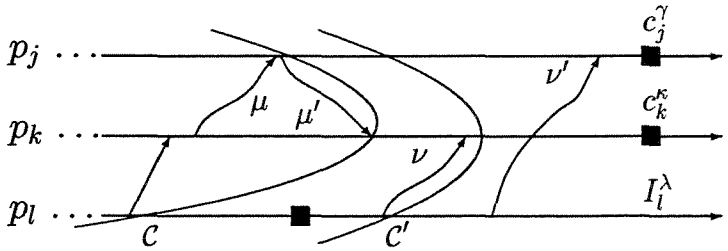


Figure 19: Contradiction hypothesis of Theorem 5.5

Corollary 5.6 *Under an RDT protocol, p_k knows that $dv_k = dv_j$ if $dv_k[k] = dv_j[k]$.*

Proof: Since $dv_k[k] = dv_j[k]$, there must exist a C-path from the current interval of p_k to p_j and, according to Theorem 5.5, $dv_k = dv_j$. □

5.4 Keeping track of equal dependency vectors

Each process maintains and propagates a size-n boolean vector *equal*. Let $equal_i$ be the vector maintained by p_i, and $m.equal$ be the vector piggybacked on a message m. The entry $equal_i[i]$ is always `true`, and the entries $equal_i[k]$, $k \neq i$, are reset to `false` when p_i takes a checkpoint. When p_i receives a message m from p_k without taking a forced checkpoint, it must update $equal_i$. If $m.dv[i] = dv_i[i]$, p_i learns that $dv_i = dv_k$ (Figure 20 (a)). If up to the knowledge of p_k when it sent m there is a process p_j such that $dv_k = dv_j$, p_i also learns that $dv_i = dv_j$ (Figure 20 (b)). This behavior can be summarized as follows:

$$\texttt{if } m.dv[i] = dv_i[i] \texttt{ then } \forall j : equal_i[j] \leftarrow equal_i[j] \vee m.equal[j]$$

There is no need for p_k to propagate additional information about dependency vectors, because if, up to the knowledge of p_k, $dv_j \neq dv_k$, p_i cannot derive from any information contained in m that $dv_i = dv_j$. According to Corollary 5.6, to the knowledge of p_k, $dv_j[k] \neq dv_k[k]$. When p_i receives m, $dv_i[k] = dv_k[k] \neq dv_j[k]$ and $dv_i \neq dv_j$ (Figure 20 (c)). Thus, keeping track of equal dependency vectors requires only $O(n)$ control information.

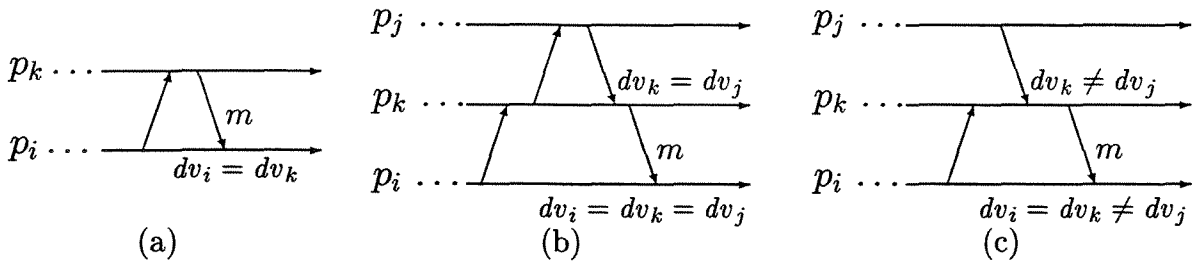


Figure 20: Updating $equal_i$

5.5 Checkpoint induction condition

A forced checkpoint is induced by p_i upon the reception of a prime message m from p_k if (i) there is a CC-cycle or (ii) there is a PMM-path from p_k to p_j, but $dv_k \neq dv_j$:

$$\begin{array}{ll} \text{(i)} & (m.dv[i] = dv_i[i] \wedge m.simple[i] = \texttt{false}) \vee \\ \text{(ii)} & (\exists j : sent\_to_i[j] \wedge m.dv[k] > dv_i[k] \wedge \neg m.equal[j]) \end{array}$$

We should note the above checkpoint induction condition is analogous to the one presented in Section 4.5. The only difference is the replacement of the test $\neg m.causal[k][j]$ for $\neg m.equal[j]$.

5.6 Optimizations

Let us continue to explore properties of the processes' behavior under an RDT protocol to simplify an implementation of the minimal characterization of RDT.

Theorem 5.7 *If p_i receives a non-prime message m, all entries of $m.dv$ are known by p_i.*

Proof: Assume that m was sent by p_k during I_k^κ, and there is a C-path μ from I_k^κ to p_i such that μ arrived to p_i before m. Assume that $m.dv[l] = \lambda > dv_i[l]$ and let μ' be a C-path from p_l to p_k that arrived to p_k after the sending of the first message of μ and before the sending of m. It is possible to construct a consistent cut C such that $\mu'.\mu$ is not left-doubled in relation to C (Figure 21). □

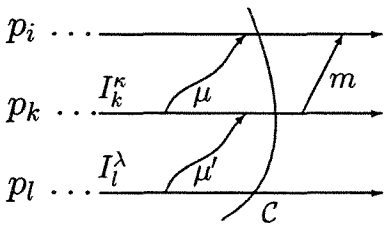


Figure 21: Contradiction hypothesis of Theorem 5.7

Due to Theorem 5.7, upon the reception of a non-prime message m, there is no need to check and update dv_i. Also, since an entry of $simple_i$ can change only if at least one entry of dv_i has changed, the updating of $simple_i$ can be skipped.

According to the second part of the proof of Theorem 5.5, when p_k knows that p_j knows its current interval, say I_k^κ, p_k cannot increase dv_k till the end of I_k^κ. Thus, we can divide a checkpoint interval of any process p_i into three phases:

Phase 0: while no message has been sent, no PMM-path can be formed, and dv_i can incorporate new dependencies without restrictions.

Phase 1: after at least one message has been sent, dv_i can change according to the induction condition presented in Section 5.5.

Phase 2: after p_i has received knowledge about other process with an equal dependency vector, no new dependency can be incorporated into dv_i.

Unfortunately, the updating of vector *equal* cannot benefit from the described optimizations. Figure 22 illustrates that vector *equal* must be updated even if no new dependency is established. When p_1 receives the second message from p_0, it does not change dv_1. However, p_1 receives knowledge that $dv_1 = dv_0$. Using this information, p_2 will be able to save a forced checkpoint when it receives a message from p_1.

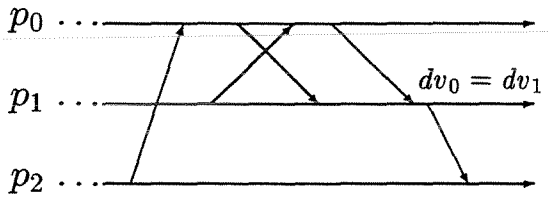


Figure 22: Vector *equal* must be updated even if no new dependency is established

Figure 23 illustrates that vector *equal* must be updated during phase 2. Process p_2 starts phase 2 when it receives a message from p_1 and it learns that $dv_1 = dv_2$. When p_0 receives a message from p_1, it learns that $dv_0 = dv_1$. Also, when p_0 sends a message to p_2, p_2 learns that $dv_0 = dv_2$. Using this information p_3 will be able to save a forced checkpoint when it receives a message from p_2. Thus, even if a process is in the phase 2 of the algorithm, it must continue to update vector *equal* because the collected information may help other processes to save checkpoints.

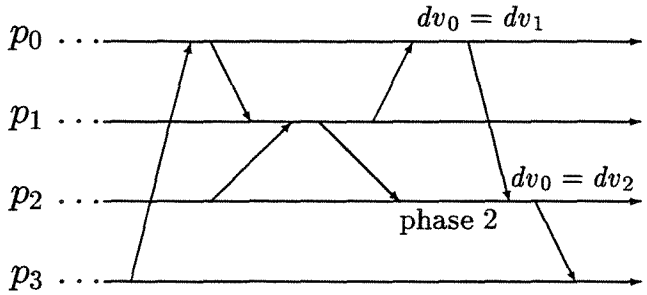


Figure 23: Vector *equal* must be updated during phase 2

An implementation of the minimal characterization of RDT including all optimizations is described in Class `RDT_Minimal`, using Java* [26].

*Java is a trademark of Sun Microsystems, Inc.

Class 1 RDT_Minimal.java (first part)

```
public class RDT_Minimal {
    public static int N = 100; // Number of processes in the application
    public int pid; // Unique process' identifier
    protected int [ ] dv = new int[N]; // Dependency vector, automatically initialezed to (0,...,0)
    protected boolean [ ] equal = new boolean [N]; // Keeps track of equal dependency vectors
    protected boolean [ ] simple = new boolean [N]; // Keeps track of simple paths
    protected boolean [ ] sent_to = new boolean [N]; // Keeps track of sent messages
    public int phase; // Keeps track of interval phase

    public class Message {
        public int sender, receiver;
        public int [ ] dv;
        public boolean [ ] equal;
        public boolean [ ] simple;
    }

    public void takeCheckpoint() {
      // Write state into stable memory
        for (int i=0; i < N; i++) { // Reset control vectors
            equal[i] = false;
            simple[i] = false;
            sent_to[i] = false;
        }
      equal[pid] = true;
      simple[pid] = true;
      dv[pid]++; // Increment dependency vector
      phase = 0; // Reset phase counter
    }

    public RDT_Minimal(int pid) { this.pid=pid; } // Constructor

    public void run() { takeCheckpoint(); } // Start execution

    public void finalize() { takeCheckpoint(); } // Finish execution

    public void sendMessage(Message m) {
      m.dv = (int [ ]) dv.clone(); // Piggybacks control information
      m.equal = (boolean [ ]) equal.clone();
      m.simple = (boolean [ ]) simple.clone();
      sent_to[m.receiver] = true;
      if (phase == 0) phase = 1;
      // Send message
    }
```

Class 1 RDT_Minimal.java (second part)

```
    private boolean mustTakeForcedCheckpoint(Message m) {
        if (phase == 0) return false; // Every new dependency can be accepted
        if (phase == 2) return true; // No new dependency can be accepted
        if (m.dv[pid] == dv[pid] && !m.simple[pid]) return true;
// Non causally doubled CC-cycle
        for (int i=0; i < N; i++) // Verify whether all PMM-paths are visibly doubled
            if (sent_to[i] && !m.equal[i]) return true;
      return false;
    }

    public void receiveMessage(Message m) {
      if (m.dv[m.sender] > dv[m.sender]) { // New dependency
        if (mustTakeForcedCheckpoint(m))
            takeCheckpoint();
        for (int i=0; i < N; i++) // Dependency vector update
            if (m.dv[i] > dv[i]) {
                dv[i] = m.dv[i];
                simple[i] = m.simple[i];
            } else if (m.dv[i] == dv[i])
                simple[i] = simple[i] && m.simple[i];
      }
      if (m.dv[pid] == dv[pid]) { // m.dv == dv
          for (int i=0; i < N; i++)
              equal[i] = equal[i] || m.equal[i];
          phase = 2;
      }
      // Message is processed by the application
    }
}
```

6 Conclusion

The simplest RDT protocols are based only on checkpoints, message-send, and message-receive events: No-Receive-After-Send, Checkpoint-After-Send, Checkpoint-Before Receive, and Checkpoint-After-Send-Before-Receive [50]. Clearly, these protocols are prone to induce a very large number of forced checkpoints. Fixed-Dependency-Interval (FDI) [31, 50] and Fixed-Dependency-After-Send (FDAS) [50] maintain and propagate dependency vectors. They force the dependency vector of a process to remain unchanged during an entire checkpoint interval (FDI) or after the first message-send event of an interval (FDAS).

The RDT protocol proposed by Baldoni, Helary, Mostefaoui, and Raynal (BHMR) was the first protocol to consider visibly doubled Z-paths [3]. Afterwards, Baldoni, Helary, and Raynal have presented a family of RDT protocols, including a refined version of BHMR [7]. Tsai, Kuo, and Wang have proved that BHMR never takes more forced checkpoints than FDAS [45]. However, the more elaborated condition used by BHMR requires the propagation of an $O(n^2)$ matrix of booleans, as described in Section 4.

Recently, we have proposed an RDT protocol, called RDT-partner, that breaks only non-trivially doubled PMM-paths [24]. RDT-partner requires only $O(n)$ control information and our simulation results have shown that it takes virtually the same number of forced checkpoints as BHMR. However, given the results presented in this article, it is now also possible to break all non-visibly doubled PMM-paths in $O(n)$.

Capítulo 10

Conclusão

A primeira contribuição desta tese foi a introdução do conceito de *visão progressiva* de uma computação distribuída, definido como uma seqüência de *checkpoints* globais consistentes que reflete o progresso desta computação. A construção de visões progressivas facilita a solução de vários problemas em sistemas distribuídos, como recuperação de falhas, depuração, monitorização e reconfiguração de computações distribuídas.

Os algoritmos propostos no Capítulo 3 para a construção de visões progressivas foram baseados na relação de *Z-precedência* entre *checkpoints*, também definida por nós. O significado da relação de Z-precedência é o seguinte: um *checkpoint a Z-precede* um checkpoint b se, e somente se, a deve ser *observado antes* de b em todas as visões progressivas de uma determinada computação distribuída. A relação de Z-precedência generaliza a relação de precedência causal proposta por Lamport [33] porque um *checkpoint* a pode Z-preceder um *checkpoint* b mesmo que a não preceda causalmente b, ou seja, a pode precisar ser observado antes de b mesmo que a não tenha acontecido antes de b.

A relação de Z-precedência também pode ser vista como uma reinterpretação de um dos resultados mais importantes da teoria sobre *checkpointing*. Netzer e Xu demonstraram que se um par de *checkpoints* for conectado por uma seqüência de mensagens, denominada *zigzag path*, estes *checkpoints* não podem fazer parte de um mesmo *checkpoint* global consistente [38]. Verificamos que um *checkpoint* a Z-precede um *checkpoint* b se, e somente se, existe uma *zigzag path* que conecta a a b. Desta forma, atribuímos um significado intuitivo às *zigzag paths*, que foi fundamental para a elaboração dos algoritmos para a construção de visões progressivas.

A definição da relação de Z-precedência foi baseada na relação de precedência causal e não utiliza seqüências de mensagens. Consideramos esta escolha mais coerente com o nível de abstração proporcionado pelos *checkpoints*, pois eventos e mensagens correspondem a uma abstração de nível mais baixo. A adoção de um nível mais alto de abstração facilitou

a escrita das provas de correção dos algoritmos para a construção de visões progressivas e permitiu a adoção destes conceitos sobre o modelo de objetos e ações atômicas [19].

Visões progressivas podem ser utilizadas para a monitorização de computações distribuídas. No Capítulo 4, apresentamos uma arquitetura de *software* para monitorização na qual a seleção de *checkpoints* pelos processos da computação distribuída é feita de maneira ortogonal à construção de visões progressivas pelo programa de controle. Acreditamos que esta arquitetura pode ser utilizada tanto para a monitorização de predicados estáveis [14], quanto para a monitorização de predicados instáveis formados pela conjunção de predicados locais [16, 25]. Neste último caso, os processos da computação deverão colaborar com o sistema de monitorização, selecionando *checkpoints* quando predicados locais são alterados.

Ainda no Capítulo 4, apresentamos uma arquitetura de *software* que integra os mecanismos de *checkpointing*, monitorização e recuperação por retrocesso de estado. Duas tarefas são necessárias para a implementação do mecanismo de recuperação por retrocesso de estado: (i) construção de linhas de recuperação após a detecção de uma falha e (ii) descarte de *checkpoints* obsoletos, ou seja, que não serão mais necessários para a construção de uma linha de recuperação. Caso a computação sobre a qual se queira adicionar a possibilidade de recuperação por retrocesso também esteja sendo monitorizada, os *checkpoints* globais consistentes obtidos durante a construção da visão progressiva poderão ser utilizados para facilitar as tarefas (i) e (ii) acima.

Outras contribuições desta tese foram baseadas na observação de que algumas propriedades precisam ser válidas ao longo de todo o progresso da computação e não apenas em relação aos *checkpoints*, como é usualmente considerado. Em particular, provamos que algumas suposições feitas na literatura para padrões de *checkpoint* que obedecem à propriedade *Rollback-Dependency Trackability* (RDT) [6, 7] não eram válidas.

A primeira suposição errada era de que a implementação da condição para indução de *checkpoints* forçados utilizada pelo protocolo *Fixed-Dependency-After-Send* (FDAS) [50] dependia da análise de todas as entradas do vetor de dependências agregado a uma mensagem. No Capítulo 5, nós provamos que é necessária apenas a comparação da entrada do vetor correspondente ao remetente da mensagem. Esta otimização diminui parte do processamento local realizado pelos processos, mas não causa nenhuma redução na complexidade das estruturas de dados mantidas e propagadas pelos processos. No entanto, a existência desta otimização foi essencial para a queda da conjetura descrita a seguir.

Baldoni, Helary e Raynal conjeturaram que o conjunto formado por *EPSCM-paths* não duplicadas visivelmente determinaria o menor conjunto de *zigzag paths* não causais que devem ser evitadas em tempo de execução por um protocolo RDT [6]. Esta conjetura estava baseada em duas observações. A primeira é que havia fortes indícios, através da análise de vários exemplos, de que a duplicação causal de *paths* mais restritivas do

que *EPSCM-paths* não garantiria a propriedade RDT. A segunda observação é que um processo só pode admitir a formação de uma *EPSCM-path* se ele obtiver informação que esta *path* já está causalmente duplicada, ou seja, está visivelmente duplicada.

Apesar de parecer bastante coerente, a soma das duas observações não estava correta. O fato de os processos evitarem a formação de todas as *Z-paths* ou apenas as não duplicadas visivelmente durante o progresso da computação deveria ser levado em conta para a determinação de qual é o conjunto mais restritivo de *Z-paths* que deveria ser considerado por um protocolo RDT.

Na otimização do protocolo FDAS descrita no Capítulo 5, os processos evitam a formação de *PMM-paths*, um subconjunto de *EPSCM-paths* não considerado por Baldoni, Helary e Raynal. No Capítulo 6, nós estendemos este resultado e provamos que o conjunto formado por *PMM-paths* não duplicadas visivelmente determina o menor conjunto de *Z-paths* cuja formação deve ser evitada por um protocolo que garante RDT em tempo de execução. Além de provarmos que a conjetura proposta por Baldoni, Helary e Raynal era falsa, este resultado introduziu uma abordagem original para a análise de protocolos RDT, baseada na verificação desta propriedade em todos os cortes consistentes de uma computação distribuída.

Baldoni, Helary e Raynal também afirmaram que a implementação de um protocolo que evitasse a formação de todas as *EPSCM-paths* não duplicadas visivelmente precisaria da manutenção e propagação de informações de controle com complexidade $O(n^2)$, onde n é o número de processos na computação [7]. Por analogia, a complexidade de um protocolo que evitasse a formação de todas as *PMM-paths* não duplicadas visivelmente também teria complexidade quadrática.

Apesar de os protocolos RDT com complexidade quadrática induzirem menos *checkpoints* forçados que o protocolo FDAS, este último era mais atraente na prática devido à sua complexidade linear e à sua simplicidade. Buscamos o desenvolvimento de um protocolo RDT que agrupasse as características positivas de ambas as abordagens, ou seja, que induzisse menos *checkpoints* forçados que o FDAS através da detecção de duplicação visível de *Z-paths*, mas que mantivesse complexidade linear.

No Capítulo 7, descrevemos o protocolo RDT-partner, que evita a formação de todas as *PMM-paths* que começam e terminam em processos distintos, mas admite a formação de *PMM-cycles* trivialmente duplicados, ou seja, *PMM-paths* que começam e terminam em um mesmo processo p_i e estão causalmente duplicadas devido ao fluxo de execução de p_i. A detecção de *PMM-cycles* trivialmente duplicados requer, além da utilização de vetores de dependência, a manutenção de um vetor de booleanos em cada processo e a propagação de um valor booleano em cada mensagem. Desta forma, conseguimos desenvolver um protocolo RDT com complexidade linear e codificação simples.

Para comparar o desempenho do RDT-Partner com outros protocolos RDT, utilizamos uma ferramenta para simulação de protocolos quase-síncronos denominada Metapromela [47, 48]. Tsai, Kuo e Wang provaram que qualquer protocolo RDT que utilize uma condição mais forte que a condição do FDAS induziria menos *checkpoints* forçados que o FDAS; os dados de simulação que encontramos na comparação do RDT-Partner e do FDAS estavam de acordo com esta afirmação.

Comparamos também o desempenho do RDT-Partner com o protocolo proposto por Baldoni, Helary, Mostefaoui e Raynal (BHMR) [3]. O protocolo BHMR é uma versão anterior do protocolo que evita a formação de todas as EPSCM-paths não duplicadas visivelmente. Escolhemos trabalhar com a versão anterior porque a versão mais recente ainda não havia sido publicada na época em que escrevemos o artigo. Para nossa surpresa, o RDT-Partner e o BHMR mostraram um comportamento quase equivalente. A explicação que encontramos para este fato foi de que um cénario em que o BHMR economiza um *checkpoint* forçados em relação ao RDT-Partner deve envolver pelo menos três processos e quatro mensagens, não sendo muito freqüente durante a computação.

Esta observação pode ser generalizada da seguinte forma: a implementação de uma condição mais restritiva C_R pode não ser mais vantajosa que a implementação de uma condição menos restritiva C_r quando as situações em que a condição C_R economiza *checkpoints* em relação à condição C_r acontecem raramente. No Capítulo 8, mostramos um exemplo desta situação no contexto de protocolos para *checkpointing* baseados em índices.

O primeiro protocolo baseado em índices foi proposto por Briatico, Ciuffoletti e Simoncini (BCS) [11] e é extremamente simples e eficiente, mas apresenta uma queda de desempenho quando os processos da computação não selecionam *checkpoints* básicos com freqüência uniforme. A literatura da área apresenta várias otimizações do BCS que tentam reduzir este efeito [9, 28], muitas vezes apresentadas em conjunto em um mesmo protocolo. No Capítulo 8, novamente utilizando Metapromela, analisamos o impacto individual e da composição de várias otimizações do protocolo BCS. Concluímos que o protocolo proposto por Baldoni, Quaglia e Fornara [9], apesar de ter complexidade linear no número de processos da computação e ser de difícil compreensão, não economiza *checkpoints* forçados em relação a um conjunto de otimizações mais simples e de complexidade constante.

A ferramenta Metapromela, que utilizamos para as simulações dos protocolos quase-síncronos, foi desenvolvida por Gustavo Maciel Dias Vieira, como parte de seu projeto de mestrado; comparações adicionais do desempenho de protocolos RDT e protocolos baseados em índices podem ser encontradas nesta dissertação [47].

Como última contribuição desta tese, o Capítulo 9 apresenta um protocolo que implementa exatamente a caracterização minimal, mas com complexidade linear. Esta redução de complexidade só foi possível graças a um estudo detalhado da variação dos vetores de dependências durante o progresso de uma computação que garante a propriedade RDT

em tempo de execução. A redução de complexidade partiu de uma abordagem de mais baixo nível, baseada em seqüências de mensagens, para uma abordagem de mais alto nível, baseada na variação dos vetores de dependências ao longo da computação. Uma questão em aberto é se este protocolo poderia ser descrito desde o princípio com uma abordagem de mais alto nível, por exemplo, utilizando Z-precedência.

Helary, Mostefaoui e Raynal, ao proporem o conceito de precedência virtual, mostraram que os protocolos quase-síncronos para *checkpointing* têm uma raiz comum [29], sendo possível codificar de maneira semelhante protocolos baseados em índices e protocolos RDT. No entanto, as regras para o desenvolvimento de protocolos utilizando precedência virtual são muito complexas e não levaram a uma visão única de todos os protocolos propostos na literatura.

Dados os resultados apresentados nesta tese, gostaríamos de estender e simplificar a apresentação dessas famílias de protocolos quase-síncronos. Como forte indício da possibilidade desta extensão temos as seguintes verificações: (i) os protocolos baseados em índices também podem se beneficiar da otimização do protocolo RDT-Partner [47] e (ii) o protocolo que implementa a caracterização minimal para RDT com complexidade linear é semelhante ao protocolo baseado em índices proposto por Helary, Mostefaoui, Netzer e Raynal [27, 28]. Novamente, gostaríamos de descrever estas famílias de protocolos utilizando Z-precedência.

Referências

[1] L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain e A. D. Mel. An Analysis of Communication-Induced Checkpointing. Em *29th International Symposium on Fault-Tolerant Computing*, pp. 242–249, Madison, Wisconsin, Estados Unidos, junho 1999.

[2] R. Baldoni, J. M. Helary, A. Mostefaoui e M. Raynal. Consistent Checkpointing in Message Passing Distributed Systems. Relatório Técnico 2564, INRIA, junho 1995.

[3] R. Baldoni, J. M. Helary, A. Mostefaoui e M. Raynal. A Communication-Induced Checkpoint Protocol that Ensures Rollback Dependency Trackability. Em *IEEE Symposium on Fault Tolerant Computing*, pp. 68–77, Seatle, Estados Unidos, julho 1997.

[4] R. Baldoni, J. M. Helary e M. Raynal. Recording in Asynchronous Computations. *Acta Informatica*, 35:441–455, 1998.

[5] R. Baldoni, J. M. Helary e M. Raynal. Rollback-Dependency Trackability: A Minimal Characterization and its Protocol. Relatório Técnico 1173, IRISA, março 1998.

[6] R. Baldoni, J. M. Helary e M. Raynal. Rollback-Dependency Trackability: Visible Characterizations. Em *18th ACM Symposium on the Principles of Distributed Computing*, pp. 33–42, Atlanta, Estados Unidos, maio 1999.

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

[8] R. Baldoni, F. Quaglia e B. Ciciani. A VP-accordant Checkpointing Protocol Preventing Useless Checkpoints. Em *17-th Symposium on Reliable Distributed Systems*, pp. 61–67, Purdue University, West Lafayette, Indiana, Estados Unidos, outubro 1998.

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

[10] J. A. Bondy e U. S. R. Murty. *Graph Theory with Applications.* Elsevier North-Holland, 1976. 264 páginas.

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

[12] G. Cao e M. Singhal. On Coordinated Checkpointing in Distributed Systems. *IEEE Trans. on Parallel and Distributed Systems*, 9(12):1213–1225, dezembro 1998.

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

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

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

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

[17] E. N. Elnozahy, D. Johnson e Y.M.Yang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Relatório Técnico CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, outubro 1996.

[18] M. J. Fischer, N. D. Griffeth e N. A. Lynch. Global States of a Distributed System. *IEEE Trans. on Software Engineering*, 8(3):198–202, maio 1982.

[19] I. C. Garcia. Estados Globais Consistentes em Sistemas Distribuídos. Dissertação de Mestrado, Instituto de Computação, Universidade Estadual de Campinas, julho 1998.

[20] I. C. Garcia e L. E. Buzato. Asynchronous Construction of Consistent Global Snapshots in the Object and Action Model. Em *4th IEEE Int. Conference on Configurable Distributed Systems*, pp. 215–223, Annapolis, Maryland, EUA, maio 1998.

[21] I. C. Garcia e L. E. Buzato. Cortes Consistentes em Aplicações Distribuídas. Em *Segundo Workshop em Sistemas Distribuídos*, pp. 9–16, Curitiba, Paraná, junho 1998.

[22] I. C. Garcia e L. E. Buzato. Progressive Construction of Consistent Global Checkpoints. Em *19th IEEE Int. Conf. on Distributed Computing Systems*, pp. 55–62, Austin, Texas, EUA, junho 1999.

[23] I. C. Garcia e L. E. Buzato. On the minimal characterization of rollback-dependency trackability property. Em *Proceedings of the 21th IEEE Int. Conf. on Distributed Computing Systems*, pp. 342–349, Phoenix, Arizona, EUA, abril 2001.

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

[25] V. K. Garg. Observation of Global Properties in Distributed Systems. Em *IEEE Int. Conference on Software and Knowledge Engineering*, pp. 418–425, Lake Tahoe, Nevada, junho 1996.

[26] J. Gosling, B. Joy e G. L. Steele. *The Java Language Specification.* Java Series. Addison–Wesley, setembro 1996.

[27] J. M. Helary, A. Mostefaoui, R. Netzer e M. Raynal. Communication-Based Prevention of Useless Checkpoints in Distributed Computations. Relatório Técnico 1105, IRISA, maio 1997.

[28] J. M. Helary, A. Mostefaoui, R. Netzer e M. Raynal. Preventing Useless Checkpoints in Distributed Computations. Em *16th Symposium on Reliable Distributed Systems*, Durham, North Carolina, USA, outubro 1997.

[29] J. M. Helary, A. Mostefaoui e M. Raynal. Virtual Precedence in Asynchronous Systems: Concept and Applications. Em *11th Int. Workshop on Distributed Algorithms*, pp. 170–184, Saarbrücken, Germany, setembro 1997.

[30] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. on Software Engineering*, 23(5), maio 1997.

[31] T. R. K. Venkatesh e H. F. Li. Optimal Checkpointing and Local Recording for Domino-Free Rollback Recovery. *Information Processing Letters*, 25(5):295–303, julho 1987.

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

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

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

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

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

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

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

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

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

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

[42] D. L. Russell. State Restoration in Systems of Communicating Processes. *IEEE Trans. on Software Engineering*, 6(2):183–194, março 1980.

[43] R. Schwarz e F. Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*, 7(3):149–174, março 1994.

[44] Sun Microsystems Computer Corporation, Mountain View, CA, USA. *Java API Documentation*, dezembro 1996.

[45] J. Tsai, S. Y. Kuo e Y. M. Wang. Theoretical Analysis for Communication-Induced Checkpointing Protocols with Rollback-Dependency Trackability. *IEEE Trans. on Parallel and Distributed Systems*, outubro 1998.

[46] J. Tsai, S. Y. Kuo e Y. M. Wang. Evaluations on Domino-Free Communication-Induced Checkpointing Protocols. *Information Processing Letters*, 69(1):31–37, janeiro 1999.

[47] G. M. D. Vieira. Estudo comparativo de algoritmos para *Checkpointing*. Dissertação de Mestrado, Instituto de Computação—Universidade Estadual de Campinas, dezembro 2001.

[48] G. M. D. Vieira. Metapromela: A toolkit for simulation of checkpointing algorithms. Em *Students Forum of the IX Brazilian Symposium on Fault-Tolerant Computers*, março 2001.

[49] G. M. D. Vieira e L. E. Buzato. Determinação de Estados Globais Consistentes em Sistemas Distribuídos. Relatório Técnico IC-99-09, Instituto de Computação, Universidade Estadual de Campinas, março 1999.

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

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

[52] Y. M. Wang e W. K. Fuchs. Lazy Checkpoint Coordination for Bounding Rollback Propagation. Em *IEEE Symp. on Reliable Distributed Systems*, pp. 78–85, outubro 1993.

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

[54] S. M. Wheater e D. L. McCue. Configuring Distributed Applications using Object Decomposition in an Atomic Action Environment. Em J. Kramer, editor, *Int. Workshop on Configurable Distributed Systems*, pp. 33–44. IEE (UK), Imperial College of Science, março 1992.

[55] J. Xu e R. H. B. Netzer. Adaptive Independent Checkpointing for Reducing Rollback Propagation. Em *IEEE Symp. on Parallel and Distributed Processing*, pp. 754–761, dezembro 1993.