Estudo Comparativo de Algoritmos para *Checkpointing*

Gustavo Maciel Dias Vieira

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Estudo Comparativo de Algoritmos para *Checkpointing*

Gustavo Maciel Dias Vieira[1]

18 de Dezembro de 2001

Banca Examinadora:

- Prof. Dr. Luiz Eduardo Buzato
  IC—UNICAMP (Orientador)
- Profª Drª Regina Helena Carlucci Santana
  ICMC—USP
- Profª Drª Maria Beatriz Felgar de Toledo
  IC—UNICAMP
- Profª Drª Eliane Martins
  IC—UNICAMP

[1]Apoio financeiro da CAPES, processo número 01P-15081/1997. Apoio financeiro adicional da FAPESP, processo número 96/1532-9 para o Laboratório de Sistemas Distribuídos e processo número 97/10982-0 para o Laboratório de Computação de Alto Desempenho.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Vieira, Gustavo Maciel Dias

V673e Estudo comparativo de algoritmos para checkpointing / Gustavo Maciel Dias Vieira -- Campinas, [S.P. :s.n.], 2001.

Orientador : Luiz Eduardo Buzato

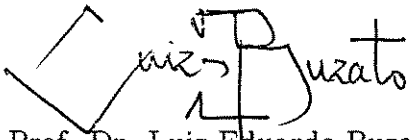
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

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

Estudo Comparativo de Algoritmos para *Checkpointing*

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Gustavo Maciel Dias Vieira e aprovada pela Banca Examinadora.

Campinas, 2 de Abril de 2002.



Prof. Dr. Luiz Eduardo Buzato
IC—UNICAMP (Orientador)

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

TERMO DE APROVAÇÃO

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

Profa. Dra. Regina Helena Carlucci Santana
USP

Profa. Dra. Maria Beatriz Felgar de Toledo
IC - UNICAMP

Profa. Dra. Eliane Martins
IC - UNICAMP

Prof. Dr. Luiz Eduardo Buzato
IC - UNICAMP

Agradecimentos

Gostaria de agradecer a várias pessoas que foram fundamentais nesta minha conquista:

A Candi, pelo amor e companheirismo inabaláveis. Pela força e maturidade em levar a vida. E por sempre ter tido a paciência de me ajudar.

Os meus pais, pelo carinho e confiança sempre presentes. Mesmo quando eu não estava.

O Buzato, meu orientador, pelas idéias, pelo rumo de pesquisa e principalmente pela motivação. Vou guardar para sempre a lembrança das nossas conversas.

A Islene, companheira de pesquisa, pela ajuda e participação na batalha do dia a dia. Os melhores resultados deste trabalho foram fruto da possibilidade de trabalhar com alguém tão competente.

Todos os companheiros de trabalho no IC, pelo excelente ambiente de trabalho e pelas maravilhosas amizades.

Os milhares de programadores que participaram da construção da infraestrutura de *software* livre, pelos brinquedos maravilhosos que são a minha paixão e vocação.

Resumo

Esta dissertação fornece um estudo comparativo abrangente de algoritmos quase-síncronos para *checkpointing*. Para tanto, utilizamos a simulação de sistemas distribuídos que nos oferece liberdade para construirmos modelos de sistemas com grande facilidade.

O estudo comparativo avaliou pela primeira vez de forma uniforme o impacto sobre o desempenho dos algoritmos de fatores como a escala do sistema, a freqüência de *checkpoints* básicos e a diferença na velocidade dos processos da aplicação. Com base nestes dados obtivemos um profundo conhecimento sobre o comportamento destes algoritmos e produzimos um valioso referencial para projetistas de sistemas em busca de algoritmos para *checkpointing* para as suas aplicações distribuídas.

Abstract

This dissertation provides a comprehensive comparative study of the performance of quasi-synchronous checkpointing algorithms. To do so we used the simulation of distributed systems, which provides freedom to build system models easily.

The comparative study assessed for the first time in an uniform environment the impact of the algorithms' performance with respect to factors such as the system's scale, the basic checkpoint rate and the relative processes' speed. By analyzing these data we acquired a deep understanding of the behavior of these algorithms and were able to produce a valuable reference to system architects looking for checkpointing algorithms for their distributed applications.

Conteúdo

Agradecimentos xi

Resumo xiii

Abstract xv

1 Introdução 1

2 *Checkpoints* Globais Consistentes 3
- 2.1 Modelo Computacional e Definições . . . 3
- 2.2 Causalidade e Ordenação de Eventos . . . 5
- 2.3 *Checkpoints* Globais Consistentes . . . 6
  - 2.3.1 Cortes e Estados Globais Consistentes . . . 7
  - 2.3.2 Consistência de *Checkpoints* . . . 8
- 2.4 Algoritmos para *Checkpointing* . . . 10
- 2.5 Comparação de Algoritmos Quase-Síncronos . . . 13

3 Metapromela 15
- 3.1 Requisitos de Simulação . . . 15
- 3.2 Spin . . . 16
- 3.3 Metapromela . . . 20
- 3.4 Construindo um Modelo com Metapromela . . . 21
  - 3.4.1 Definição do Algoritmo . . . 21
  - 3.4.2 Definição da Rede de Comunicação . . . 25
  - 3.4.3 Definição das Prioridades . . . 26
  - 3.4.4 Uso da Ferramenta . . . 27
- 3.5 Executando um Cenário . . . 28
  - 3.5.1 Definição de um Cenário . . . 28
  - 3.5.2 Uso da Ferramenta . . . 32

4 Algoritmos Estudados **33**
4.1 Algoritmos ZPF 34
4.2 Algoritmos ZCF 43
4.3 Algoritmos Não Comparados 56

5 Análise e Conclusão **59**
5.1 Métricas para Comparação 59
5.2 Análise Comparativa 62
5.2.1 Algoritmos ZPF 62
5.2.2 Algoritmos ZCF 69
5.2.3 O Custo de um Padrão ZPF 75
5.3 Conclusão 77

A Médias e Desvios Padrão **81**

Bibliografia **91**

Lista de Figuras

2.1 Diagrama espaço tempo: estados 5
2.2 Diagrama espaço tempo: *checkpoints* 5
2.3 Cortes consistentes e inconsistentes 7
2.4 Consistência de *checkpoints* e efeito dominó 9
2.5 Protocolo quase-síncrono e *checkpoint* forçado. 12

3.1 Exemplo de expressões em Promela 17
3.2 Algoritmo BCS em Promela 18
3.3 Xspin 19
3.4 Simulação de um modelo. 20
3.5 Execução de um cenário. 21
3.6 Exemplo de definição de algoritmo. 23
3.7 Exemplo de tipo de mensagem. 24
3.8 Exemplo de definição de rede de comunicação. 25
3.9 Exemplo de definição de rede de comunicação como grafo completo. 26
3.10 Exemplo de definição de prioridades. 27
3.11 Definição de cenário em função da rede de comunicação. 29
3.12 Definição de cenário em função das prioridades. 31

4.1 Exemplo do padrão gerado pelo algoritmo CASBR. 34
4.2 Exemplo do padrão gerado pelo algoritmo CAS. 35
4.3 Exemplo do padrão gerado pelo algoritmo CBR. 35
4.4 Exemplo do padrão gerado pelo algoritmo NRAS. 37
4.5 Exemplo do padrão gerado pelo algoritmo FDI. 37
4.6 Exemplo do padrão gerado pelo algoritmo FDAS. 39
4.7 Exemplo do padrão gerado pelo algoritmo RDT-Partner. 40
4.8 Exemplo do padrão gerado pelo algoritmo BHMR. 40
4.9 Exemplo do padrão gerado pelo algoritmo BCS. 43
4.10 Exemplo do padrão gerado pelo algoritmo BCS-Aftersend. 45
4.11 Exemplo do padrão gerado pelo algoritmo BCS-Partner. 45

4.12 Exemplo do padrão gerado pelo algoritmo HMNR. 48
4.13 Exemplo do padrão gerado pelo algoritmo Lazy-BCS. 50
4.14 Exemplo do padrão gerado pelo algoritmo Lazy-BCS-Aftersend. 51
4.15 Exemplo do padrão gerado pelo algoritmo Lazy-BCS-Partner. 51
4.16 Exemplo do padrão gerado pelo algoritmo BQF. 54
4.17 Exemplo do padrão gerado pelo algoritmo BQC. 54

5.1 Algoritmos ZPF somente baseados em modelo - 1 63
5.2 Algoritmos ZPF somente baseados em modelo - 2 64
5.3 Algoritmos ZPF que usam relógios simples - 1 65
5.4 Algoritmos ZPF que usam relógios simples - 2 66
5.5 Algoritmos ZPF que usam relógios elaborados - 1 67
5.6 Algoritmos ZPF que usam relógios elaborados - 2 68
5.7 Algoritmos ZCF que evitam *checkpoints* desnecessários - 1 70
5.8 Algoritmos ZCF que evitam *checkpoints* desnecessários - 2 71
5.9 Algoritmos ZCF que evitam incrementar os índices - 1 73
5.10 Algoritmos ZCF que evitam incrementar os índices - 2 74
5.11 O algoritmo BQC - 1 75
5.12 O algoritmo BQC - 2 76

Lista de Algoritmos

4.1 Algoritmo CASBR 35
4.2 Algoritmo CAS 36
4.3 Algoritmo CBR 36
4.4 Algoritmo NRAS 38
4.5 Algoritmo FDI 38
4.6 Algoritmo FDAS 39
4.7 Algoritmo RDT-Partner 41
4.8 Algoritmo BHMR 42
4.9 Algoritmo BCS 44
4.10 Algoritmo BCS-Aftersend 46
4.11 Algoritmo BCS-Partner 47
4.12 Algoritmo HMNR 49
4.13 Algoritmo Lazy-BCS 50
4.14 Algoritmo Lazy-BCS-Aftersend 52
4.15 Algoritmo Lazy-BCS-Partner 53
4.16 Algoritmo BQF 55
4.17 Algoritmo BQC 57

Lista de Tabelas

A.1 Médias e desvios padrão, cenário SP - 1 83
A.2 Médias e desvios padrão, cenário SP - 2 83
A.3 Médias e desvios padrão, cenário SP - 3 84
A.4 Médias e desvios padrão, cenário SI - 1 84
A.5 Médias e desvios padrão, cenário SI - 2 85
A.6 Médias e desvios padrão, cenário SI - 3 85
A.7 Médias e desvios padrão, cenário AV - 1 86
A.8 Médias e desvios padrão, cenário AV - 2 86
A.9 Médias e desvios padrão, cenário AV - 3 87
A.10 Médias e desvios padrão, cenário AP - 1 87
A.11 Médias e desvios padrão, cenário AP - 2 88
A.12 Médias e desvios padrão, cenário AP - 3 88
A.13 Médias e desvios padrão, cenário AI - 1 89
A.14 Médias e desvios padrão, cenário AI - 2 89
A.15 Médias e desvios padrão, cenário AI - 3 90

Capítulo 1

Introdução

Checkpoints são estados de interesse dos processos participantes de uma aplicação distribuída, armazenados em memória persistente de modo que seja possível ter uma abstração do estado da aplicação ao longo de sua execução. Um *checkpoint* global é um conjunto formado pelos *checkpoints* locais dos processos da aplicação, um para cada processo. Nem todos os *checkpoints* globais representam estados que poderiam ser observados na aplicação durante uma possível execução, os *checkpoints* globais que respeitam esta propriedade são chamados *checkpoints* globais consistentes. Usando *checkpoints* globais consistentes podemos observar o progresso da computação em um sistema de depuração distribuído [10, 30] ou recuperar parte da computação já realizada em caso de falha [9], entre outras possíveis aplicações.

Uma aplicação distribuída que use *checkpointing* normalmente é composta por várias partes: um mecanismo de obtenção de estados locais, um mecanismo de armazenamento em memória persistente de *checkpoints*, um algoritmo ou protocolo que decide quais estados devem ser armazenados como *checkpoints*, um algoritmo de coordenação entre os processos para a construção de um *checkpoint* global a partir dos *checkpoints* locais dos processos e mecanismos específicos ao uso destes *checkpoints* globais na aplicação.

Nesta dissertação nós só estamos interessados no problema da seleção de quais estados devem ser *checkpoints*. Mais precisamente, estudamos aqueles algoritmos que usam a abordagem quase-síncrona para tomar esta decisão, ou seja, os processos estão livres para selecionar os seus estados de interesse de acordo com a importância dos seus dados locais, mas o algoritmo pode induzir *checkpoints* extras de modo a permitir que o algoritmo de construção de *checkpoints* globais consiga obter algum *checkpoint* global consistente. Uma visão mais abrangente sobre os detalhes do processo de *checkpointing* pode ser encontrada em [9] e em [11].

A diversidade de aplicações de *checkpointing* tem motivado o desenvolvimento de vários algoritmos quase-síncronos, entretanto há escassez na literatura pertinente de tra-

balhos que permitam a seleção de um ou mais algoritmos para *checkpointing* em função dos requisitos e das características da aplicação distribuída alvo. Os poucos trabalhos que fazem algum tipo de comparação de desempenho entre os algoritmos se limitam a estudar um conjunto bem restrito de algoritmos. Estes estudos se mostram menos úteis na medida que nem sempre é possível utilizar a combinação de seus resultados para ter uma noção do comportamento de vários algoritmos devido a suposições de teste fundamentalmente diferentes.

Esta dissertação tem como objetivo fornecer um estudo abrangente e uniforme do desempenho dos algoritmos para *checkpointing*. Para tanto, utilizamos a simulação de sistemas distribuídos que nos oferece liberdade para construirmos modelos de sistemas com grande facilidade. Temos como conseqüência a possibilidade de testar os algoritmos em uma grande variedade de configurações, utilizando um grande número de testes individuais com um investimento de esforço muito pequeno. A simulação foi feita usando-se uma ferramenta de simulação de algoritmos quase-síncronos para *checkpointing* chamada Metapromela, construída por nós sobre uma outra ferramenta mais geral o Spin [18].

Como resultado deste trabalho fomos capazes de estudar detalhadamente um número considerável de algoritmos quase-síncronos para *checkpointing*, produzindo pela primeira vez dados comparativos sobre estes algoritmos utilizando um ambiente uniforme. Fomos capazes de avaliar o impacto sobre o desempenho dos algoritmos de fatores como a escala do sistema, a freqüência de *checkpoints* básicos e a diferença na velocidade dos processos da aplicação. Com base nestes dados obtivemos um grande conhecimento sobre as características destes algoritmos, sendo este estudo um valioso referencial para projetistas de sistemas em busca de algoritmos para *checkpointing* para as suas aplicações distribuídas.

Esta dissertação está organizada da seguinte forma. No Capítulo 2 é apresentada uma introdução teórica ao problema de consistência de *checkpoints*. O Capítulo 3 apresenta a ferramenta Metapromela e como ela pode ser usada para construir modelos de algoritmos para *checkpointing*. O Capítulo 4 descreve os algoritmos estudados nesta dissertação. Por fim, no Capítulo 5 é feito o estudo comparativo destes algoritmos, usando como base de comparação os resultados obtidos em nossos cenários de simulação.

Capítulo 2

Checkpoints Globais Consistentes

A principal noção associada a consistência de *checkpoints* globais é a de ordenação de eventos. De forma simplificada, podemos observar inconsistência em um *checkpoint* global quando ocorre o registro da recepção de uma mensagem que ainda não foi enviada. Ou seja, ocorreu uma ordenação dos eventos que desrespeita a relação de causa e efeito. É com base nesta relação que Lamport [20] define a dependência causal, um evento depende causalmente de outro se ele "aconteceu depois" do primeiro. Usando esta relação é possível determinar uma ordenação parcial para os eventos do sistema que respeita a causalidade. Podemos então afirmar que um *checkpoint* global é consistente se não existir dependência causal entre os *checkpoints* que o compõem [2].

Porém, não basta evitar a existência de dependência causal entre os *checkpoints* da aplicação, pois nem sempre é possível transformar um subconjunto de *checkpoints* não relacionados pela causalidade em um *checkpoint* global consistente [9, 24]. Uma outra relação de dependência, descoberta por Netzer e Xu [23], é o que caracteriza a condição necessária e suficiente para que um conjunto de *checkpoints* seja estendido à um *checkpoint* global consistente.

Neste Capítulo apresentaremos o modelo computacional adotado em nosso estudo de *checkpoints* globais consistentes, o conceito de precedência causal, a definição de um *checkpoint* global consistente e a condição necessária e suficiente para a consistência de conjuntos de *checkpoints*. Apresentaremos também as principais abordagens para a obtenção de uma seqüência de *checkpoints* que possa ser usada para construir *checkpoints* globais consistentes.

2.1 Modelo Computacional e Definições

Uma *aplicação distribuída* consiste em um conjunto $\{p_0, p_1, \ldots, p_{n-1}\}$ de n *processos* seqüenciais que trabalham para alcançar um objetivo comum. Estes processos são au-

tônomos, eles não compartilham memória ou têm acesso a um relógio global, sendo a comunicação entre os processos realizada inteiramente via troca de mensagens sobre uma *rede de comunicação*.

A rede de comunicação é um grafo orientado fortemente conexo, onde os n vértices representam os processos da aplicação e as arestas orientadas representam canais unidirecionais entre os processos por elas ligados. As mensagens só podem ser enviadas por meio destes canais, mas supõe-se que todos os processos podem se comunicar entre si, mesmo que indiretamente. O mecanismo de troca de mensagens garante que mensagens não são corrompidas, mas o tempo de entrega não é limitado e as mensagens podem ser entregues fora de ordem.

Processos têm a sua execução modelada como uma seqüência de eventos, onde e_i^k representa o k-ésimo evento executado pelo processo p_i. O evento e_i^0 denota o evento inicial da computação realizada pelo processo p_i. Os eventos são classificados em dois tipos: eventos internos e eventos de comunicação. Os eventos de comunicação representam o envio ou a recepção de uma mensagem, todos os outros eventos de um processo são internos. Os processos mantém um conjunto de variáveis locais que constituem o seu *estado*, sendo σ_i^k o estado do processo p_i após a execução do evento e_i^k.

Checkpoints são estados selecionados que formam uma abstração da execução de um processo. Os estados que podem corresponder a *checkpoints* são aqueles associados a execução de eventos internos do processo, assim temos que $\hat{\sigma}_i^k$ representa o k-ésimo *checkpoint* do processo p_i, que corresponde a um estado $\sigma_i^{k'}$, tal que $k \leq k'$ e $e_i^{k'}$ é um evento interno. Chamamos de *padrão de checkpoints* uma escolha específica de quais estados do processo correspondem a *checkpoints*. Os processos tiram um *checkpoint* inicial imediatamente após o início da execução e um *checkpoint* final imediatamente antes do término da execução. Um *intervalo de checkpoints* Δ_i^k é composto pelos estados do processo p_i entre um *checkpoint* $\hat{\sigma}_i^k$ e seu sucessor imediato $\hat{\sigma}_i^{k+1}$, incluindo $\hat{\sigma}_i^k$ e excluindo $\hat{\sigma}_i^{k+1}$.

A maneira usual de se representar graficamente a execução de uma aplicação distribuída é através de um *diagrama espaço-tempo* [20]. Linhas horizontais representam a execução dos processos ao longo do tempo, que progride da esquerda para a direita. Eventos são representados como pontos sobre estas retas e mensagens trocadas por processos são representadas por setas ligando o evento de envio ao evento de recepção da mensagem.

A Figura 2.1 é um exemplo de diagrama espaço-tempo representando os eventos de 3 processos. Os *checkpoints* são representados substituindo-se os eventos que geram o estado de interesse por quadrados preenchidos sobre as retas do diagrama. Na maioria dos casos estaremos interessados apenas nos *checkpoints* de uma execução, omitindo da representação gráfica os eventos internos. Um exemplo de diagrama espaço-tempo representando *checkpoints* está na Figura 2.2.

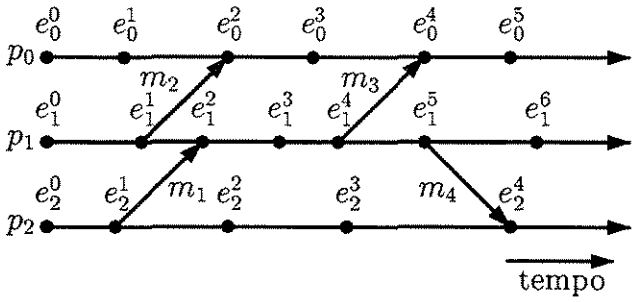


Figura 2.1: Diagrama espaço tempo: estados

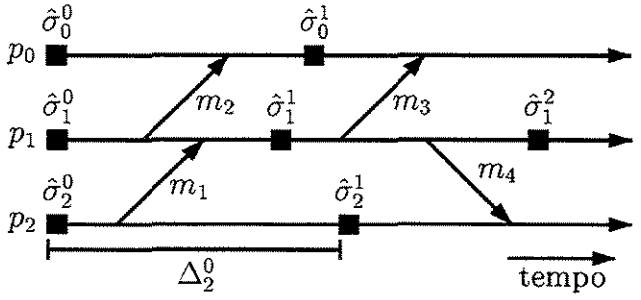


Figura 2.2: Diagrama espaço tempo: *checkpoints*

2.2 Causalidade e Ordenação de Eventos

Como ocorre com outros problemas relacionados a sistemas distribuídos, para obtermos um *checkpoint* global consistente é útil determinarmos uma ordenação entre os eventos observados no sistema. A *relação de precedência causal* [20] é suficiente para estabelecer uma ordenação consistente dos eventos, sendo a base da caracterização de um *checkpoint* global consistente.

A precedência causal captura a dependência entre eventos. Dois eventos ligados por uma relação de causa e efeito tem uma ordem intrínseca que deve ser respeitada em uma ordenação consistente dos eventos do sistema. Um evento e precede causalmente outro evento e' se e pode ter influenciado a ocorrência de e'. A relação de precedência causal ($\rightarrow$) é definida da seguinte maneira [20]:

1. Se e_i^k e e_i^l são eventos de p_i e $k < l$, então $e_i^k \rightarrow e_i^l$;
2. Se para uma mensagem m, $e_i = \text{envio}(m)$ e $e_j = \text{recepção}(m)$, então $e_i \rightarrow e_j$;
3. Se $e \rightarrow e'$ e $e' \rightarrow e''$, então $e \rightarrow e''$.

Se dois eventos ocorrem em um mesmo processo, então a ordem de execução destes eventos neste processo indica a ordem em que estes eventos devem ser observados em uma ordenação consistente de todos os eventos do sistema. Esta é uma regra natural e é proveniente da natureza seqüencial da execução dos processos. Também é natural supor que o evento de envio de uma mensagem deva preceder o evento de recepção desta mesma mensagem. Por último, a relação de precedência causal é transitiva. Se o evento de recepção de uma mensagem é precedido pelo evento de envio, então todos os eventos subseqüentes ao recebimento desta mensagem também dependem causalmente do evento de envio da mensagem.

A relação de precedência causal é não-reflexiva, não existe sentido físico em afirmar que um evento precede a si mesmo. Dois eventos e e e', tais que $e \not\rightarrow e'$ e $e' \not\rightarrow e$, são chamados eventos concorrentes e são denotados por $e \,||\, e'$. Isto implica que não existe relação causal entre estes dois eventos e que não existe uma ordem implícita na qual estes eventos devam aparecer em uma ordenação consistente de todos os eventos do sistema.

A relação de precedência causal não define uma ordenação total dos eventos do sistema. Não existe a necessidade de se impor uma ordem a eventos que sejam concorrentes entre si. De forma geral, se é necessária uma ordenação total dos eventos, uma ordenação arbitrária dos eventos concorrentes é suficiente, mantendo-se consistente com a relação de causalidade.

Nesta dissertação usaremos uma definição e notação estendida da relação de precedência causal entre eventos para englobar estados e *checkpoints*. Existe uma relação de precedência causal entre dois estados σ_i^k e σ_j^l se, e somente se, existir esta relação entre os eventos e_i^k e e_j^l que geraram estes estados, ou seja:

$$\sigma_i^k \rightarrow \sigma_j^l \Leftrightarrow e_i^k \rightarrow e_j^l.$$

Analogamente, existe uma relação de precedência causal entre dois *checkpoints* $\hat{\sigma}_i^k$ e $\hat{\sigma}_j^l$ se, e somente se, existirem entre os estados $\sigma_i^{k'}$ e $\sigma_j^{l'}$ correspondentes a estes *checkpoints* uma relação de precedência:

$$\hat{\sigma}_i^k \rightarrow \hat{\sigma}_j^l \Leftrightarrow \sigma_i^{k'} \rightarrow \sigma_j^{l'}.$$

2.3 *Checkpoints* Globais Consistentes

Uma observação do estado global de uma aplicação distribuída pode ser inconsistente se este estado implica em uma quebra na relação de causa e efeito entre os eventos. Não faz sentido um estado global onde os estados de dois processos, p_i e p_j, mostram que p_i recebeu uma mensagem de p_j, mas p_j não enviou esta mensagem [7].

De forma geral, devemos estabelecer critérios consistentes com a causalidade para determinar quais estados formam um estado global consistente. Nesta Seção descreveremos

o que caracteriza um estado global consistente. Estenderemos esta caracterização para *checkpoints* globais consistentes e veremos quais as outras exigências impostas pela consistência entre *checkpoints*.

2.3.1 Cortes e Estados Globais Consistentes

Um *corte* [2] é um subconjunto dos eventos executados pela aplicação distribuída formado, para cada processo i, por todos os eventos e_i^k onde $0 \leq k \leq c_i$ para um número natural c_i. Um corte pode ser especificado por um conjunto destes n números naturais $\{c_0, c_1, \ldots, c_{n-1}\}$. O conjunto dos últimos eventos de cada processo em um corte $\{e_0^{c_0}, e_1^{c_1}, \ldots, e_{n-1}^{c_{n-1}}\}$ corresponde à *fronteira do corte*. Um *estado global* (Σ) é um conjunto dos estados locais de cada um dos processos, sendo determinado pelos estados dos processos na fronteira de um corte, logo $\Sigma = \{\sigma_0^{c_0}, \sigma_1^{c_1}, \ldots, \sigma_{n-1}^{c_{n-1}}\}$.

Um corte C é consistente se para quaisquer dois eventos e e e' pertencentes ao corte a seguinte relação é válida [2]:

$$(e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C.$$

Esta condição é coerente com a relação de precedência causal. Se um evento faz parte de um corte, então todos os eventos que o precederam causalmente também devem fazer parte do corte. É interessante notar que podem existir relações de precedência entre dois eventos na fronteira do corte sem que este fato torne o corte inconsistente.

A Figura 2.3 mostra dois cortes C e C' como partições do diagrama espaço-tempo. Os eventos que se situam à esquerda da partição fazem parte do corte, sendo os últimos eventos antes da partição a fronteira do corte. Por exemplo, a fronteira do corte C é $\{e_0^1, e_1^2, e_2^2\}$. Nesta figura temos que o corte C é consistente enquanto que o corte C' é inconsistente. O critério de consistência pode ser visto graficamente da seguinte forma: se as setas que representam as mensagens possuem sua origem do lado esquerdo do corte e seu fim do lado direito, então o corte é consistente, caso contrário o corte é inconsistente.

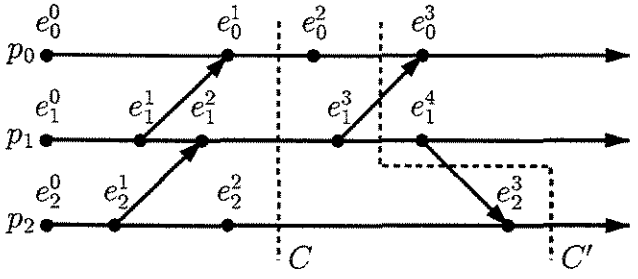


Figura 2.3: Cortes consistentes e inconsistentes

Um estado global é consistente se for gerado pela fronteira de um corte consistente [2]. Desta forma um estado global consistente pode ser visto como uma possível observação do sistema que respeite a relação de precedência causal. Na Figura 2.3 o estado global $\Sigma_C = \{\sigma_0^1, \sigma_1^2, \sigma_2^2\}$ é consistente enquanto o estado global $\Sigma_{C'} = \{\sigma_0^2, \sigma_1^3, \sigma_2^3\}$ não é consistente.

Uma outra forma de se expressar a relação de consistência é em termos dos estados na fronteira do corte. Um estado global $\Sigma = \{\sigma_0^{c_0}, \sigma_1^{c_1}, \ldots, \sigma_{n-1}^{c_{n-1}}\}$ é consistente se, e somente se:

$$\forall i, j : 0 \leq i, j \leq n-1 : \sigma_i^{c_i+1} \not\rightarrow \sigma_j^{c_j}.$$

A relação de causalidade é o único tipo de relação de dependência que afeta a consistência de estados globais. Desta forma, para um subconjunto dos processos do sistema, qualquer conjunto de estados concorrentes destes processos pode tornar-se um estado global consistente, escolhendo-se estados concorrentes apropriados dos outros processos [11].

2.3.2 Consistência de *Checkpoints*

Seria muito dispendioso utilizarmos todos os eventos de uma computação para construirmos estados globais consistentes. Em um sistema distribuído que armazene o seu estado por meio de *checkpointing*, as aplicações registram o seu estado apenas em alguns *checkpoints* selecionados. Um *checkpoint global* ($\hat{\Sigma}$) consiste em um conjunto de n *checkpoints* locais, um para cada processo, especificado por um conjunto de inteiros $\{c'_0, c'_1, \ldots, c'_{n-1}\}$, isto é $\hat{\Sigma} = \{\hat{\sigma}_0^{c'_0}, \hat{\sigma}_1^{c'_1}, \ldots, \hat{\sigma}_{n-1}^{c'_{n-1}}\}$. Analogamente à relação entre estados e *checkpoints* locais, um *checkpoint* global $\hat{\Sigma} = \{\hat{\sigma}_0^{c'_0}, \hat{\sigma}_1^{c'_1}, \ldots, \hat{\sigma}_{n-1}^{c'_{n-1}}\}$ corresponde a um estado global $\Sigma = \{\sigma_0^{c_0}, \sigma_1^{c_1}, \ldots, \sigma_{n-1}^{c_{n-1}}\}$ onde $\forall k : 0 \leq k \leq n-1 : c'_k \leq c_k$.

Como *checkpoints* não envolvem os eventos de troca de mensagens, temos uma condição de consistência para *checkpoints* globais ligeiramente diferente daquela associada a estados globais. Um *checkpoint* global é consistente se ele representa a fronteira de um corte consistente [22]. Mas, a condição de consistência é expressa em termos da relação de precedência causal da seguinte forma: um *checkpoint* global é consistente se, e somente se:

$$\forall i, j : 0 \leq i, j \leq n-1 : \hat{\sigma}_i^{c_i} \not\rightarrow \hat{\sigma}_j^{c_j}.$$

A consistência exige que os *checkpoints* sejam concorrentes entre si [23]. Por exemplo, na Figura 2.4 a existência de uma relação de precedência $\hat{\sigma}_1^1 \rightarrow \hat{\sigma}_2^1$ implica na existência de um evento e, posterior a $\hat{\sigma}_1^1$, que precede causalmente um evento e', anterior a $\hat{\sigma}_2^1$. Desta forma, o corte C que contém os eventos que geraram os estados $\hat{\sigma}_1^1$ e $\hat{\sigma}_2^1$ em sua fronteira não é consistente.

Apesar de muito parecida com a regra de consistência para todos os estados, a regra de consistência para *checkpoints* traz uma implicação séria para aplicações distribuídas

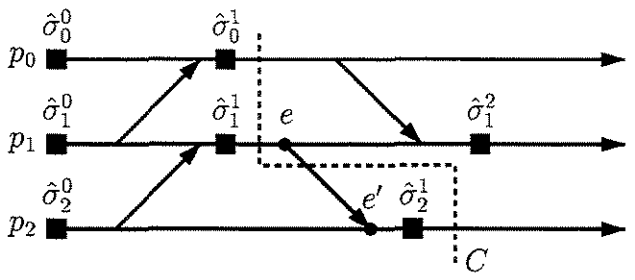


Figura 2.4: Consistência de *checkpoints* e efeito dominó

que desejem usar *checkpointing*. Uma escolha arbitrária de quais estados são usados como *checkpoints* pode levar, no pior caso, a que nenhum dos *checkpoints* armazenados possa ser aproveitado para compor um *checkpoint* global consistente. Este efeito foi observado pela primeira vez no contexto de recuperação de falhas por retrocesso de estado e é chamado de *efeito dominó* [24].

Na Figura 2.4 dois *checkpoints* concorrentes $\hat{\sigma}_0^1$ e $\hat{\sigma}_2^1$ não podem participar ao mesmo tempo de nenhum *checkpoint* global consistente, porque $\hat{\sigma}_0^1 \rightarrow \hat{\sigma}_1^2$ e $\hat{\sigma}_1^1 \rightarrow \hat{\sigma}_2^1$. Ou seja, estes dois *checkpoints* estão de tal forma relacionados que existe precedência causal entre o conjunto formado por estes dois *checkpoints* e todos os *checkpoints* do processo p_1. A Figura 2.4 mostra também um cenário onde, ao longo da execução da aplicação distribuída, o único *checkpoint* global consistente obtido por meio dos *checkpoints* locais é aquele formado pelos *checkpoints* iniciais dos processos.

O efeito dominó existe devido a uma outra relação de dependência entre *checkpoints* além da relação de precedência causal, que é determinante não só na consistência, mas na possibilidade de um *checkpoint* ou conjunto de *checkpoints* fazerem parte de algum *checkpoint* global consistente. Esta relação foi descoberta por Netzer e Xu [23] e é chamada de *caminho em ziguezague* ou *caminho-z*.

Existe um caminho-z entre dois *checkpoints* $\hat{\sigma}_i^k$ e $\hat{\sigma}_j^l$ se, e somente se:

- $i = j \wedge k < l$, ou
- existe uma seqüência de mensagens $m_1, m_2, \ldots, m_q$ $(q \geq 1)$ tal que:
  1. m_1 é enviada por p_i após $\hat{\sigma}_i^k$,
  2. se m_a $(1 \leq a < q)$ é recebida por um processo p_r, então m_{a+1} é enviada por p_r neste mesmo intervalo de *checkpoints* ou em algum intervalo posterior a este (embora m_{a+1} possa ter sido enviada antes ou depois de m_a ser recebida) e
  3. m_q é recebida por p_j antes de $\hat{\sigma}_j^l$.

Um *caminho-z não causal* ocorre quando existe um caminho-z entre dois *checkpoints* $\hat{\sigma}_i^k$ e $\hat{\sigma}_j^l$, tal que para algum par de mensagens (m_a, m_{a+1}) no caminho-z entre $\hat{\sigma}_i^k$ e $\hat{\sigma}_j^l$, m_{a+1} foi enviada antes de m_a ter sido recebida no mesmo intervalo de *checkpoints*.

Netzer e Xu estabeleceram a condição necessária e suficiente para que um conjunto de *checkpoints* possa ser estendido para um *checkpoint* global consistente em termos desta relação. Um conjunto S de *checkpoints* pode participar de algum *checkpoint* global consistente se, e somente se, não existir um caminho-z entre quaisquer dois *checkpoints* pertencentes ao conjunto S.

É possível observar, por inspeção, que todos os conjuntos de dois *checkpoints* da Figura 2.4, com no máximo um *checkpoint* inicial, possuem algum caminho-z entre seus elementos. Por isso, o único *checkpoint* global consistente é aquele formado pelos *checkpoints* iniciais.

A relação definida pelo caminho-z é reflexiva, podendo existir um caminho-z ligando um *checkpoint* $\hat{\sigma}$ a si mesmo. Se este for o caso e considerarmos o conjunto $S = \{\hat{\sigma}\}$, então $\hat{\sigma}$ não pode fazer parte de nenhum *checkpoint* global consistente. É dito que $\hat{\sigma}$ está envolvido em um *ciclo-z* e que este *checkpoint* é *inútil*. Na Figura 2.4 os *checkpoints* $\hat{\sigma}_0^1$ e $\hat{\sigma}_1^1$ são inúteis, pois ambos fazem parte de um ciclo-z.

Se o padrão de *checkpoints* de uma aplicação não possui *checkpoints* inúteis, então todos os *checkpoints* tirados pelos processos fazem parte de um *checkpoint* global consistente e esse padrão está livre de efeito dominó. Uma aplicação que deseje evitar que o efeito dominó se propague indefinidamente deve, no mínimo, garantir que uma parcela de seus *checkpoints* não sejam inúteis. Garantias mais fortes como por exemplo a ausência total de *checkpoints* inúteis ou até de caminhos-z não causais podem gerar padrões que possibilitem a obtenção mais fácil de *checkpoints* globais consistentes a partir dos *checkpoints* locais [11, 12, 30].

2.4 Algoritmos para *Checkpointing*

Chamamos de *checkpointing* o procedimento empregado pela aplicação para decidir quais estados dos processos individuais serão usados como *checkpoints*. Como mostramos na Seção anterior, esta escolha deve levar em conta os critérios de consistência entre *checkpoints* para garantir a ausência ou minimizar a ocorrência de *checkpoints* inúteis. Em geral considera-se três abordagens para *checkpointing*: síncrona, assíncrona e quase-síncrona [22].

A abordagem síncrona suspende a operação da aplicação enquanto os processos obtém um *checkpoint* global. Esta suspensão interrompe o fluxo de mensagens da aplicação entre os processos, garantindo que os *checkpoints* locais obtidos sejam concorrentes e o *checkpoint* global resultante seja consistente. Esta abordagem evita completamente

a existência de qualquer tipo de relação de dependência entre *checkpoints*, mas paga o preço da interrupção da execução da aplicação, interrupção essa que pode ser causada por apenas um processo que tenha um estado importante a ser preservado. Uma vantagem desta abordagem é que a determinação dos *checkpoints* globais consistentes da aplicação é imediata e está disponível sem custo adicional a todos os processos da aplicação [9].

A abordagem assíncrona, por outro lado, não impõe restrições sobre quais estados são usados como *checkpoints*, estando desta forma sujeita à ocorrência de *checkpoints* inúteis e do efeito dominó. Esta abordagem permite liberdade total aos processos individuais na escolha de seus *checkpoints*, mas expõe a aplicação ao risco de ter que descartar uma grande parte dos mesmos. Além disso, o processo de obtenção de um *checkpoint* global consistente, supondo que um exista, envolve uma computação centralizada ou custosa de ser executada de modo distribuído [30].

A abordagem quase-síncrona aparece como um compromisso entre as abordagens síncrona e assíncrona. Esta abordagem deixa os processos livres para escolherem os seus *checkpoints*, mas os força a tirarem *checkpoints* seguindo um algoritmo, de modo a evitar ou minimizar a ocorrência de relações de dependência entre *checkpoints*. Em geral, a decisão de quando forçar um *checkpoint* é tomada fazendo-se uso de informação de controle adicionada às mensagens normais da aplicação. Os *checkpoints* tirados livremente pelos processos são chamados *checkpoints básicos* e os *checkpoints* induzidos pelo algoritmo são chamados *checkpoints forçados*. Padrões de *checkpoints* gerados usando-se esta abordagem podem apresentar diferentes comportamentos [22] que afetam o processo de obtenção de *checkpoints* globais consistentes [11, 12]. Os algoritmos para *checkpointing* que utilizam a abordagem assíncrona são chamados *algoritmos quase-síncronos* e são o objeto de estudo desta dissertação.

Algoritmos Quase-síncronos

Os algoritmos quase-síncronos funcionam observado as mensagens e as relações de dependência inferidas das informações de controle. Estes algoritmos atuam processando as mensagens da aplicação, ou melhor, a informação de controle contida nestas mensagens, antes de que as mesmas sejam entregues ao processo. Caso o algoritmo detecte que o recebimento desta mensagem pode induzir relações de dependência indesejáveis, um *checkpoint* é forçado e só então a mensagem é entregue ao processo.

A Figura 2.5 mostra como o padrão de *checkpoints* da Figura 2.4 ficaria se um algoritmo quase-síncrono estivesse em execução. O *checkpoint* forçado $\hat{\sigma}_2^1$ (representado por um quadrado sem preenchimento) quebra os ciclos-z no quais os *checkpoints* $\hat{\sigma}_0^1$ e $\hat{\sigma}_1^1$ estão envolvidos, desta forma o conjunto $\{\hat{\sigma}_0^1, \hat{\sigma}_1^1\}$ passa a poder participar de um *checkpoint* global consistente $\hat{\Sigma} = \{\hat{\sigma}_0^1, \hat{\sigma}_1^1, \hat{\sigma}_2^1\}$.

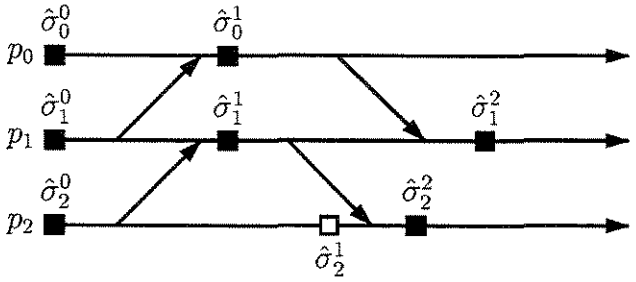


Figura 2.5: Protocolo quase-síncrono e *checkpoint* forçado.

Não é possível acompanhar a existência de caminhos-z ao longo da seqüência de mensagens e *checkpoints* de uma aplicação durante a execução da mesma [22]. Os algoritmos quase-síncronos forçam *checkpoints* para evitar a formação de estruturas associadas a possíveis relações de dependência entre *checkpoints*, como não há maneira de um processo saber seguramente se a estrutura fará ou não parte de um caminho-z real, estes algoritmos tendem a forçar mais *checkpoints* do que o mínimo necessário.

Os algoritmos quase-síncronos são então heurísticas que garantem certas propriedades do padrão de *checkpoints* gerado. Podemos identificar três grupos de padrões com propriedades bem definidas [22]:

ZPF (*z-path free*): Este padrão é livre de caminhos-z não causais que não sejam duplicados por um caminho-z causal. Este é o padrão mais forte de algoritmos quase-síncronos e possui a propriedade de que a única relação de dependência entre *checkpoints* é a precedência causal. Manivannan e Singhal [22] mostraram que algoritmos que respeitem este padrão também respeitam a propriedade de RDT (*rollback dependency trackability*), ou seja, exibem características que tornam fácil a obtenção de um *checkpoint* global consistente de forma distribuída [30]. Por outro lado, este padrão força um grande número de *checkpoints* e mostramos no Capítulo 5 que este número é pouco dependente do número de *checkpoints* básicos. Os algoritmos conhecidos que respeitam este padrão são chamados de *baseados em modelo* (*model-based*) [9, 30].

ZCF (*z-cicle free*): Este padrão é livre de ciclos-z, garantindo apenas a inexistência de *checkpoints* inúteis. Este é o mais fraco padrão que garante o aproveitamento de todos os *checkpoints* tirados pelos processos em pelo menos um *checkpoint* global consistente. Este padrão força menos *checkpoints* que o padrão ZPF, mas precisa de um mecanismo um pouco mais complicado para a obtenção de *checkpoints* globais consistentes [11, 12]. Os algoritmos conhecidos que respeitam este padrão são bem simples e eficientes em termos de informação de controle e são chamados de *baseados*

em índice (*index-based*) [9, 28].

PZCF (*partially z-cicle free*): Este padrão não garante a ausência de ciclos-z, mas se esforça para evitá-los. Este padrão se aproxima dos algoritmos assíncronos em relação a dificuldade de se obter um *checkpoint* global consistente, mas procura dar algumas garantias estatísticas sobre o impacto da ocorrência de *checkpoints* inúteis. Normalmente algoritmos que respeitam este padrão são simplificações dos algoritmos das classes ZPF e ZCF [22].

2.5 Comparação de Algoritmos Quase-Síncronos

Como observamos, os algoritmos quase-síncronos nem sempre forçam o número mínimo de *checkpoints* necessário para garantir as propriedades dos padrões de *checkpoints* que eles geram. Desta forma, é imperativo sermos capazes de avaliar a eficiência de cada um dos algoritmos para podermos decidir qual seria o algoritmo mais indicado para uma aplicação.

Não fomos capazes de encontrar entre os trabalhos existentes na área algum estudo abrangente sobre a comparação de algoritmos quase-síncronos para *checkpointing*. A maioria dos artigos discute pontualmente o desempenho de alguns poucos algoritmos, apenas no contexto da justificativa do autor de que a sua nova proposta de algoritmo é superior às anteriores.

Vejamos alguns exemplos de comparações de algoritmos quase-síncronos encontradas na literatura:

- Xu e Netzer [32] apresentam um estudo do desempenho do algoritmo que eles propuseram, fazendo comparação apenas com uma estratégia periódica (assíncrona) de *checkpointing*. O estudo foi feito medindo-se o número de *checkpoints* forçados na execução de 6 programas distribuídos, em função da freqüência dos *checkpoints* locais dos processos.

- Baldoni, Hélary, Mostefaoui e Raynal [3] apresentam um novo algoritmo ZPF e fazem estudos de simulação de seu desempenho. A comparação é feita apenas em relação a um outro algoritmo, medindo-se o número de *checkpoints* forçados em 3 cenários e em função função da freqüência dos *checkpoints* locais dos processos. Mesmo comparando apenas dois algoritmos, este é o único trabalho em nosso conhecimento que avalia o desempenho de algoritmos ZPF.

- Baldoni, Quaglia e Ciciani [4] também apresentam um algoritmo novo e o analisam com estudos de simulação. O estudo só considera dois algoritmos em dois cenários

distintos. Foram medidos o número de *checkpoints* forçados em função da freqüência dos *checkpoints* locais dos processos.

- Baldoni, Quaglia e Fornara justificam o algoritmo apresentado em [5] com um extenso estudo de simulação que considera vários cenários e configurações de simetria e de padrões de comunicação. No entanto, este estudo só compara dois algoritmos, medindo a redução no número de *checkpoints* forçados em relação a um terceiro.

- Alvisi et al. [1] comparam três algoritmos ZCF usando o registro de execução de 4 aplicações distribuídas. Eles usam também de simulação para estudar o efeito da escala sobre os algoritmos estudados. Este estudo somente se preocupa com os algoritmos da classe ZCF e não explora os efeitos da freqüência dos *checkpoints* locais dos processos, fator este muito importante no desempenho de algoritmos desta classe.

- Zambonelli [33] faz um estudo mais abrangente sobre os algoritmos da classe ZCF, estudando o comportamento de quatro algoritmos desta classe usando o registro de execução de quatro aplicações distribuídas. Porém, este trabalho deixa de considerar algoritmos importantes nesta classe e se limita aos cenários destas aplicações, não explorando outras situações.

Entre estes trabalhos percebemos um grande número de métricas diferentes, sendo que poucos algoritmos são avaliados sob uma mesma métrica. O objetivo principal desta pesquisa de mestrado é sanar esta deficiência, provendo uma comparação abrangente destes algoritmos realizada sob as mesmas condições de implementação e execução.

Capítulo 3

Metapromela

Para podermos ser capazes de realizar o estudo abrangente dos algoritmos quase-síncronos para *checkpointing* que nos propusemos, decidimos aliar simulação a um arcabouço simples para a implementação dos algoritmos, com o objetivo de maximizar o uso dos recursos disponíveis.

Infelizmente, não fomos capazes de encontrar um ambiente de simulação existente que satisfizesse nossas necessidades. Decidimos então estender a ferramenta Spin [18], que já atendia várias de nossas exigências e construirmos sobre ela nossa própria ferramenta, Metapromela.

Neste Capítulo apresentaremos as características desejáveis em uma ferramenta de simulação, as ferramentas Spin e Metapromela, como Metapromela foi construída a partir do Spin e uma descrição de como usá-la para construir modelos de algoritmos quase-síncronos.

3.1 Requisitos de Simulação

Nós identificamos algumas características desejáveis em uma ferramenta de simulação útil para o estudo de algoritmos para *checkpointing*:

Fácil implementação dos algoritmos: Deve ser fácil implementar ou descrever sem ambigüidade algoritmos quase-síncronos na ferramenta. Nós queremos ser capazes de implementar sem esforço um grande número de algoritmos e suas variações.

Ambiente de simulação controlado: Nós queremos ser capazes de ter um controle bem preciso dos parâmetros de simulação. Estes parâmetros estão normalmente relacionados a restrições de *software* e *hardware* do sistema distribuído. Nós queremos alterar estes parâmetros e observar o comportamento e desempenho dos algoritmos.

Reprodutibilidade de simulações: Todas as simulações devem ser reprodutíveis, permitindo que vários algoritmos sejam executados estritamente sob as mesmas condições. Adicionalmente, caso a ferramenta esteja disponível, esta reprodutibilidade permite que nossos dados sejam gerados e analisados independentemente.

Representação visual: Deve ser possível observar graficamente a simulação dos algoritmos, aumentando nossa habilidade de associar mentalmente as construções teóricas com as execuções reais dos algoritmos, estimulado nossa intuição.

Decidimos construir nossa ferramenta de simulação com base em uma outra ferramenta de simulação para sistemas distribuídos mais abrangente, uma ferramenta que já atendia grande parte dos requisitos almejados e que seria especializada conforme necessário. A ferramenta escolhida para servir como substrato foi o Spin [18] e nós construímos nossa ferramenta como um conjunto de *scripts* em Perl [29] que geram código para o Spin.

3.2 Spin

O Spin [18] é uma ferramenta para simular e executar análise de consistência em algoritmos e protocolos distribuídos. O sistema distribuído a ser estudado é modelado em uma linguagem chamada Promela, que possui como abstrações principais: processos, canais de comunicação e variáveis. O modelo em Promela é usado para a simulação de um sistema distribuído ou para a verificação formal exaustiva de propriedades do algoritmo, como ausência de *deadlocks* ou a existência de código não alcançável.

A característica mais importante do Spin para o nosso trabalho é a definição da linguagem Promela e o analisador desta linguagem fornecido com a ferramenta. Promela é deliberadamente simples e concisa para evitar a poluição do modelo com detalhes de implementação. Apesar de sua simplicidade, Promela fornece algumas primitivas poderosas para comunicação entre processos, sendo bastante adequada para a implementação de algoritmos distribuídos. É importante salientar que Promela é uma linguagem de modelagem, todos os aspectos que não estejam diretamente relacionados à interação entre os processos são eliminados do modelo. Detalhes sobre a linguagem Promela podem ser encontrados em [14].

Promela define um modelo de eventos não determinista uniforme. Cada expressão é um evento atômico e tem definido um critério de executabilidade. Se a expressão não é executável, o processamento pára até que a expressão se torne executável, provavelmente como conseqüência de uma ação em um outro processo. Na presença de uma seleção de expressões, todos os eventos executáveis têm a mesma probabilidade de serem executados. Infelizmente, o Spin não oferece controle sobre a prioridade de escolha dos eventos além deste modelo uniforme. Uma parcela considerável do trabalho no Metapromela foi na

direção de vencer estas limitações e fornecer algum mecanismo com o qual controlar as prioridades dos eventos.

A Figura 3.1 mostra algumas expressões em Promela. As expressões de atribuição `a = 0`, `b = 1`, `a = a + b + b` e `a = a - b` são sempre executáveis. Quando o operador de seleção `do` é encontrado, inicialmente somente as duas expressões de atribuição são executáveis, sendo que cada uma delas tem 50% de chance de ser escolhida para execução. A partir do momento que o valor acumulado na variável `a` for maior que 10, então a expressão `(a > 10)` passa a ser executável, podendo ser escolhida com a mesma probabilidade que as outras expressões do condicional `do`. Quando isto ocorrer, o laço será terminado pelo comando `break`.

```
a = 0;
b = 1;
do
:: a = a + b + b
:: a = a - b
:: (a > 10) -> break
od
```

Figura 3.1: Exemplo de expressões em Promela

Um exemplo que ilustra bem algumas das técnicas de modelagem de um algoritmo quase-síncrono na linguagem Promela é mostrado na Figura 3.2. Este algoritmo simples é conhecido como BCS e será descrito em mais detalhes no Capítulo 4 (Seção 4.2). O modelo em Promela deste algoritmo considera que um processo pode executar três operações básicas: mandar uma mensagem, receber uma mensagem ou tirar um *checkpoint* básico, sendo irrelevantes as demais atividades do processo. Durante uma simulação do algoritmo, estas três operações são executadas não-deterministamente, seqüencialmente para cada um dos processos da aplicação. A mensagem consiste apenas na informação de controle e é endereçada arbitrariamente a um dos processos. É interessante notar que a escolha do destinatário de cada mensagem só ocorre no momento do recebimento da mesma, pois todos os processos compartilham o mesmo canal e a ordem na qual os processos retiram mensagens deste canal é não-determinista.

Spin possui várias propriedades úteis, advindas de um ambiente de simulação controlado: é possível observar a execução como um observador onisciente, é possível escolher uma ordem exata de execução de eventos e gerar simulações pseudo-aleatórias reprodutíveis, entre outras funções. Também é possível observar visualmente a simulação através da

```
#define N 3        /* Número de processos na aplicação */
#define L 30       /* Tamanho do canal de comunicação */

chan outside = [L] of { byte } /* Declaração do canal de comunicação */

active [N] proctype node()  /* Declaração e instanciação dos processos */
{
   byte lc= 0;    /* Relógio lógico do processo */
   byte mlc;      /* Relógio lógico recebido em mensagens */

   printf("Processo %d, lc %d; take_basic_checkpoint\n", _pid, lc);

   do /* Seleção não-determinista de comandos com guarda */

   /* Evento interno (checkpoint básico) */
   :: skip ->  /* Guarda sempre habilitada */
        lc= lc+1;
        printf("Processo %d, lc %d; take_basic_checkpoint\n", _pid, lc)

   /* Envio de mensagem */
   :: outside!lc /* Guarda habilitada se o canal não estiver cheio */

   /* Recepção de mensagem */
   :: outside?mlc -> /* Guarda habilitada se o canal não estiver vazio */
        if
        :: (mlc > lc)  -> /* Checkpoint forçado */
              printf("Processo %d, lc %d; take_forced_checkpoint\n",
                     _pid, lc);
              lc= mlc
        :: (mlc <= lc) /* Prossegue sem tirar um checkpoint forçado */
        fi
   od
}
```

Figura 3.2: Algoritmo BCS em Promela

visualizador gráfico Xspin. A Figura 3.3 mostra a exibição de diagrama espaço tempo do Xspin, executando o algoritmo BCS.

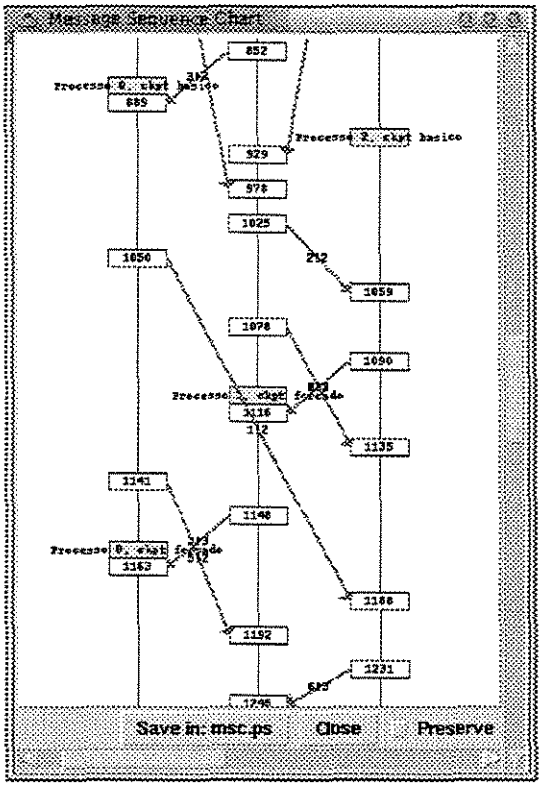


Figura 3.3: Xspin

O Spin é também uma ferramenta para a verificação formal exaustiva de propriedades de um modelo de sistema distribuído. A verificação exaustiva consiste na geração de todos os comportamentos possíveis de um algoritmo ou protocolo, buscando por estados que violem predicados que devem se manter verdadeiros durante a execução.

O suporte a verificação foi relevante para a escolha do Spin como ferramenta a ser usada no nosso trabalho, pois desejávamos estudar a possibilidade de usar a técnica de prova exaustiva para obter resultados referentes aos algoritmos para *checkpointing*. Infelizmente, o modelo computacional que adotamos não permite que resultados sejam obtidos por meio de provas exaustivas, pois não existe como impor uma quantidade finita de comportamentos aos algoritmos para *checkpointing*, uma vez que não existem limites para a diferença de velocidade entre os processos.

3.3 Metapromela

O exemplo da Figura 3.2 ilustra a simplicidade de um modelo em Promela, porém este modelo não permite modelar diferentes prioridades entre os eventos nem uma rede de comunicação mais complexa. Para poder usar o Spin para simular algoritmos quase-síncronos, nós devemos descrever um modelo completo em Promela de todo o sistema distribuído, composto pelos processos e pelos canais de comunicação, e devemos ser capazes de configurar neste modelo os parâmetros do sistema a ser modelado. Isto deve ser feito de forma prática para viabilizar um estudo abrangente dos algoritmos quase-síncronos.

Promela não possui mecanismos para facilitar a organização de programas grandes ou reuso de código. Para facilitar a implementação, nós separamos as definições da implementação do processo (que corresponde ao algoritmo), da rede de comunicação e da distribuição das prioridades associadas a cada evento. Somado a isso, nós extraímos o código comum a todos os algoritmos quase-síncronos, simplificando a implementação dos processos a simples especificação do comportamento de cada evento, de acordo com o algoritmo implementado.

O programa `metapromela` usa estas definições para gerar o modelo em Promela do sistema. Estes modelos podem ser usados diretamente no Spin ou Xspin para simular e visualizar a execução do algoritmo. Os passos envolvidos na construção e simulação de um modelo estão na Figura 3.4. Na próxima Seção descreveremos com mais detalhes como cada uma das definições são construídas.

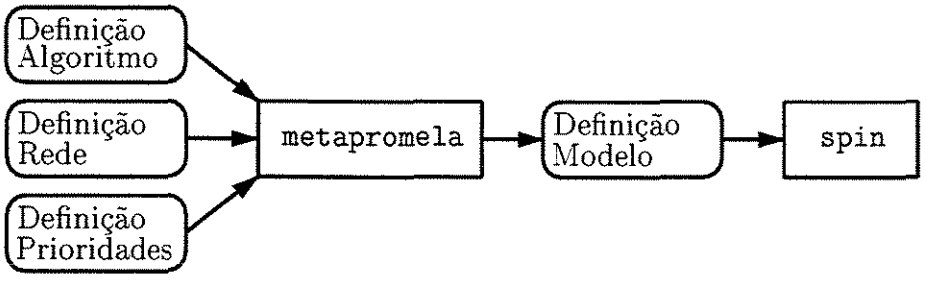


Figura 3.4: Simulação de um modelo.

A abordagem de definições separadas nos ofereceu algumas vantagens. Primeiro, explorando a estrutura similar dos algoritmos quase-síncronos, nós fomos capazes de simplificar a especificação dos algoritmos individuais definindo apenas o comportamento específico de cada algoritmo. Adicionalmente, ao colocar grande parte da implementação do modelo sob a responsabilidade da ferramenta, nós pudemos controlar mais uniformemente vários parâmetros da simulação do modelo. Esta separação também permite uma maior

modularidade e como conseqüência facilidade de alterar, corrigir ou adaptar a maneira como o sistema está sendo modelado, de forma independente aos algoritmos individuais. Os cenários de testes apresentados no Capítulo 5 só foram possíveis graças a esta flexibilidade.

A ferramenta também provê suporte para o teste automatizado de grupos de algoritmos, usando o conceito de cenário. Um cenário é a especificação de um conjunto de algoritmos e parâmetros usados para guiar a construção e simulação de modelos, permitindo a coleta de estatísticas de simulação. Parte da ferramenta Metapromela, o programa `run` lê um arquivo de configuração que descreve um cenário, executa uma série de simulações e fornece como saída tabelas comparativas de indicadores do desempenho dos algoritmos. A saída é facilmente usável pelo programa GnuPlot para desenhar gráficos comparativos. A estrutura de um cenário é mostrada na Figura 3.5. Uma descrição detalhada do arquivo de configuração de cenário está na Seção 3.5.

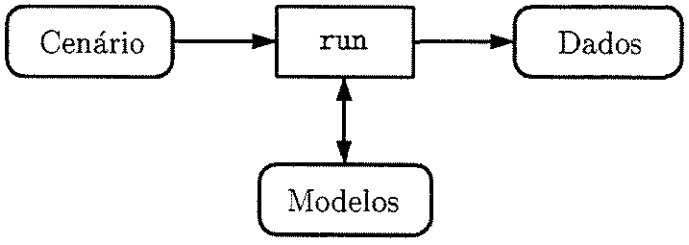


Figura 3.5: Execução de um cenário.

3.4 Construindo um Modelo com Metapromela

A especificação de um modelo usando Metapromela é feita definindo-se três arquivos de configuração, referentes ao algoritmo, à rede de comunicação e às prioridades relativas dos eventos. Nesta Seção descreveremos o formato de cada um destes arquivos e como usar o programa.

3.4.1 Definição do Algoritmo

Um algoritmo quase-síncrono age normalmente em três situações:

- Anexa informação de controle às mensagens da aplicação no momento de seu envio pelo processo;

- Processa a informação de controle anexada às mensagens recebidas antes que elas sejam entregues a aplicação e, caso necessário, força um *checkpoint*;
- Atualiza suas estruturas de dados na ocorrência de um *checkpoint* básico.

Desta forma, em cada processo, coerentemente com o modelo computacional adotado, nós consideramos a execução como uma sucessão de eventos atômicos de três tipos: evento interno, evento de envio de mensagem e evento de recepção de mensagem. Como resultado, alteramos a granularidade dos blocos atômicos em Promela, definido nossos próprios átomos que devem ser compostos por várias expressões deterministas em Promela.

A especificação do algoritmo se resume a implementar, em Promela, o comportamento do algoritmo em cada um dos eventos gerados pelo simulador. Esta implementação é feita em um arquivo, dividido em seções de acordo com o exemplo da Figura 3.6, que mostra a implementação do protocolo BCS (Seção 4.2). Para cada uma das seções que compõem a definição, o código em Promela deve estar entre os marcadores `MP<SEÇÃO>` e `MPEND`. Este código deve ser determinista, isto é, cada operador de seleção pode ter no máximo apenas uma expressão executável de cada vez. Não iremos entrar em detalhes sobre a sintaxe de Promela, maiores informações podem ser encontradas em [14] e [17].

O arquivo de configuração de algoritmo tem três partes principais: as duas primeiras seções são referentes a declarações globais do modelo, as duas seções seguintes configuram o início do algoritmo e as últimas três seções são efetivamente os eventos a serem implementados. O código a ser implementado pode, e em algumas situações deve, fazer uso de algumas constantes e funções pré-definidas no modelo. Em todas as seções estão sempre disponíveis as constantes `mp_n`, `mp_n2` e `mp_n3`, contendo respectivamente o número de processos na aplicação, este número ao quadrado e este número ao cubo, para facilitar a implementação de vetores, matrizes bidimensionais e tridimensionais.

A definição começa com a seção `MPTYPEDECL`. Esta seção é usada para a definição dos tipos de dados em Promela que serão usados na implementação do algoritmo nas outras seções. Em geral é aqui que é feita a definição do tipo da mensagem, que é na verdade o tipo da informação de controle. No exemplo da Figura 3.6 esta seção está vazia, pois a informação de controle usa o tipo elementar `int`. A Figura 3.7 mostra a definição desta seção para o algoritmo FDAS (Seção 4.1). A seção `MPTYPE` é usada para informar o tipo da mensagem e deve conter apenas uma linha, com apenas uma palavra, informando o tipo da informação a ser propagada entre os processos.

As duas próximas seções configuram aspectos relativos ao comportamento de cada um dos processos da aplicação individualmente. Tudo o que for declarado aqui, estará disponível de forma separada para cada um dos processos. Destas seções em diante algumas constantes e funções estão disponíveis. A constante `mp_pid` contém um identificador numérico do processo, seqüencial entre 0 e `mp_n` $-$ 1. As funções `mp_ckpt_basic(pid)` e

```
#
# Definição do algoritmo BCS
#

MPTYPEDECL
MPEND

MPTYPE
int
MPEND

MPDECL
  int lc;
MPEND

MPINIT
  lc= 0;  mp_ckpt_basic(mp_pid);  lc++;
MPEND

MPEVENT
  mp_ckpt_basic(mp_pid);  lc++;
MPEND

MPSEND
  mp_mess= lc;
MPEND

MPRECEIVE
  if
    :: (mp_mess <= lc)
    :: else ->
       mp_ckpt_forced(mp_pid);
       lc= mp_mess
  fi;
MPEND
```

Figura 3.6: Exemplo de definição de algoritmo.

```
MPTYPEDECL
typedef FDASdata {
   int dv[mp_n];
}
MPEND
```

Figura 3.7: Exemplo de tipo de mensagem.

`mp_ckpt_forced(pid)` devem ser chamadas sempre que um processo tirar um *checkpoint* básico ou forçado, respectivamente. Estas funções implementam várias atividades, entre elas a contagem de *checkpoints* básicos e forçados.

A seção `MPDECL` é usada para declarações de variáveis que serão usadas pelo algoritmo. Aqui podem ser usados os tipos definidos em `MPTYPEDECL` e as variáveis aqui declaradas são visíveis em todas as seções subseqüentes. A seção `MPINIT` é usada para o início do protocolo, devendo conter os comandos necessários para colocar as variáveis na condição inicial e normalmente o pedido para tirar o *checkpoint* básico inicial.

As três seções restantes são a implementação propriamente dita do algoritmo. A seção `MPEVENT` é usada para codificar a execução de um evento interno, ou seja, o evento da ocorrência de um *checkpoint* básico, o único evento desta classe que interessa ao modelo. O algoritmo quase-síncrono deve atualizar as suas estruturas de dados e efetuar o *checkpoint* chamando a função `mp_ckpt_basic(pid)`.

A seção `MPSEND` é usada para implementar o envio de uma mensagem. Estamos abstraindo o conteúdo da mensagem, só nos preocupando com a informação de controle. Nesta seção o algoritmo deve calcular a informação de controle a ser enviada e copiá-la para o local de envio. Para tanto esta seção fornece duas variáveis: `mp_mess` e `mp_dest`. A mensagem deve ser copiada para a variável `mp_mess`, que é do tipo definido na seção `MPTYPE` e a variável `mp_dest` contém o destinatário da mensagem, que pode ser usado no cálculo da informação de controle.

A seção `MPRECEIVE` é usada para implementar a recepção de uma mensagem. Novamente, a mensagem é apenas a informação de controle associada a ela. Nesta seção o algoritmo deve processar a informação de controle e decidir se o processo deve ou não tirar um *checkpoint* forçado usando a função `mp_ckpt_forced(pid)`, além de atualizar as suas estruturas de dados para refletir a informação de controle recebida. Esta seção fornece duas variáveis `mp_mess` e `mp_orig`, a mensagem recebida está na variável `mp_mess`, que é do tipo definido na seção `MPTYPE`, a variável `mp_orig` contém o identificador do processo que enviou a mensagem.

3.4.2 Definição da Rede de Comunicação

Nos modelos gerados por Metapromela a rede de comunicação é representada por um grafo orientado onde os processos são os vértices e as arestas orientadas representam os canais unidirecionais. A definição da rede nada mais é do que a especificação deste grafo. Um exemplo de definição de rede está mostrado na Figura 3.8. Este arquivo também está dividido em seções, mas o terminador de uma seção agora é uma linha em branco.

```
# Uma definição de rede de comunicação

MPNETTYPE
graph

MPSIZE
4 4

MPGRAPH
0 1
0 2
1 2
3 0
```

Figura 3.8: Exemplo de definição de rede de comunicação.

A seção `MPNETTYPE` especifica o tipo da rede de comunicação. Por brevidade de representação, a definição de rede de comunicação é feita de duas formas: `complete` define a rede como um grafo completo e `graph` define a rede como o grafo especificado nas outras seções. A Figura 3.9 mostra uma definição de rede de comunicação como um grafo completo.

A seção `MPSIZE` especifica o número de vértices e arestas do grafo, ou seja o número de processos e o número de canais de comunicação. Esta seção deve conter uma linha com um par de números <`número de processos`, `número de canais`>, caso a rede de comunicação seja do tipo `graph`. Caso a rede de comunicação seja `complete`, o número de canais é ignorado e pode estar ausente, devendo ser especificado apenas o número de processos.

Quando o tipo da rede é `graph`, o grafo da rede de comunicação é definido na seção `MPGRAPH`. Esta seção é organizada da seguinte forma: uma seqüência de pares, cada um por linha, representando os canais unidirecionais de comunicação, <`processo origem`, `processo destino`>. Esta seção deve aparecer após as seções `MPNETTYPE` e `MPSIZE` e

```
# Rede de comunicação completa com 6 processos

MPNETTYPE
complete

MPSIZE
6
```

Figura 3.9: Exemplo de definição de rede de comunicação como grafo completo.

somente caso a rede seja do tipo `graph`. Caso a rede seja do tipo `complete`, são criados canais bidirecionais entre todos os processos. O número de arestas deve ser igual ao número indicado em `MPSIZE` e os processos só podem assumir índices entre 0 e o número de processos indicado em `MPSIZE` menos um.

3.4.3 Definição das Prioridades

O simulador gera uma seqüência pseudo-aleatória de eventos dos processos do sistema, esta seqüência pode ser vista como uma sucessão de mensagens e *checkpoints* do sistema. A geração desta seqüência não é afetada pelo comportamento dos algoritmos, isto é, por suas computações ou pelos seus *checkpoints* forçados. Dada uma mesma semente do gerador de números pseudo-aleatórios, todas as simulações observarão exatamente a mesma seqüência de mensagens e *checkpoints* básicos.

A definição de prioridades controla como esta seqüência é gerada em termos das prioridades relativas entre os processos do sistema e entre os eventos de cada um dos processos. O arquivo de definição possui o formato exibido na Figura 3.10. Este arquivo possui apenas uma seção, `MPPRIORITY`, que é terminada por uma linha em branco ou pelo fim do arquivo.

A seção `MPPRIORITY` é onde são estabelecidas as prioridades relativas para cada processo e para os eventos de *checkpoint* básico, envio e recepção de mensagens. As prioridades são representadas por inteiros e são computadas de forma relativa à prioridade normal, que é 1. Um evento com prioridade 5 tem 5 vezes mais chances de ocorrer que um evento com prioridade 1. Para prioridades de eventos, é possível usar prioridades negativas, a prioridade -5 para um evento faz com que ele seja 5 vezes menos provável de acontecer que os eventos com prioridade 1.

Nesta seção o primeiro número, que deve estar sozinho em uma linha, representa o número de processos que terão suas prioridades atribuídas. Não é necessário atribuir prioridades a todos os processos do sistema, aqueles processos que forem omitidos terão

```
# Exemplo de definição de prioridades de eventos e processos

MPPRIORITY
5
0 3 -3 2 3
1 2 1 2 3
3 1 3 2 1
4 2 1 1 1
5 1 1 -2 -2
```

Figura 3.10: Exemplo de definição de prioridades.

prioridade 1 para si e para todos os seus eventos. Para cada um dos processos a terem prioridades atribuídas, deve existir uma linha contendo a lista de inteiros: <pid, prioridade, interno, envio, recepção>. Onde pid identifica o processo e prioridade dá a prioridade deste processo em relação aos outros processos do sistema. Os números interno, envio e recepção dão as prioridades dos eventos internos, de envio de mensagens e de recepção de mensagens, respectivamente, para este processo. Estes três últimos valores podem ser negativos. Caso seja atribuída prioridade a um processo cujo identificador não está presente no sistema, a definição de prioridade para este processo é ignorada.

3.4.4 Uso da Ferramenta

Uma vez criados os arquivos de definição do algoritmo, da rede de comunicação e das prioridades, um modelo é gerado usando-se o programa metapromela, da seguinte forma:

```
metapromela [-q] <algoritmo> <rede> <prioridades> <condição>
```

A opção -q gera um modelo que executa no Spin sem produzir saída alguma no terminal até o fim da execução. Caso esta opção não esteja presente, são exibidas mensagens a cada *checkpoint* tirado pelos processos. Os parâmetros algoritmo, rede e prioridades são os nomes de arquivos que contém as definições das partes do modelo.

O último parâmetro, condição, é uma expressão booleana, escrita em Promela, que quando falsa encerra a execução do processo. Esta condição é usada para determinar até quando executar o modelo e pode ser escrita em termos de qualquer variável definida pelo algoritmo ou de quatro variáveis globais definidas e atualizadas automaticamente pelo modelo: mp_basic, mp_forced, mp_sentmess e mp_receivedmess. Estas variáveis contam o número total de *checkpoints* básicos, de *checkpoints* forçados, de mensagens enviadas e de mensagens recebidas, respectivamente.

Um exemplo de chamada do programa poderia ser:

```
metapromela bcs.mprl completa6 none-faster "mp_basic < 1000"
```

O resultado desta execução é um modelo em Promela, que é escrito na saída padrão do programa. Esta saída pode ser redirecionada para um arquivo ou canalizada diretamente para o executável `spin`, passando antes pelo pré-processador C `cpp`.

```
metapromela ... | cpp | spin
```

O modelo gerado também pode ser usado no Xspin, para visualizar graficamente a simulação. Ao encerrar a simulação o modelo escreve na saída padrão a contagem das variáveis `mp_basic`, `mp_forced`, `mp_sentmess` e `mp_receivedmess`, que podem ser usadas para avaliar o comportamento do algoritmo.

3.5 Executando um Cenário

Usando o programa `metapromela` podemos gerar código em Promela para um modelo. Porém, a simulação de apenas um modelo não é suficiente para tirarmos conclusões sobre os algoritmos. Para fazermos comparações de vários algoritmos, em função de algum dos parâmetros do sistema, usamos um cenário. Nesta Seção descreveremos o formato do arquivo de definição de cenário e como usar o programa `run`.

3.5.1 Definição de um Cenário

Um cenário representa uma série de modelos, executados em seqüência para alguns algoritmos determinados, onde dados são obtidos em função da definição de rede de comunicação ou em função das prioridades. Isto é, podemos selecionar um grupo de algoritmos e, para uma definição de prioridades fixa, variar a rede de comunicação ou, para uma definição de rede de comunicação fixa, variar as prioridades. Os cenários, como são executados atualmente, somente coletam dados quantitativos referentes ao número de *checkpoints* forçados.

A definição de cenário é feita por arquivo como o exibido na Figura 3.11. O arquivo está dividido em seções marcadas por um identificador entre colchetes e que são terminadas por uma linha em branco. Nas seções onde são feitas referências a nomes de arquivos é possível usar caracteres coringa, que serão expandidos na ordem e pelas regras da *shell* sendo usada.

A definição começa na seção `[ALGORITHMS]`, onde são definidos os algoritmos que participarão dos testes. Devem estar listados os nomes de arquivos de definição separados por espaços ou por quebras de linha, sendo possível usar caracteres coringa. Os testes do

```
# Exemplo de definição de cenário (Rede de comunicação)

[ALGORITHMS]
algoritmos/bcs* algoritmos/fdi.mprml
algoritmos/fdas*

[TYPE]
net

[RANGE]
2 10 3

#[NETWORKS] Se esta seção não existe, supõe-se rede completa

[PRIORITIES]
prioridades/normal

[WHILE]
mp_basic < (100*mp_n)

[DIVIDE]
yes

[ITERATIONS]
3

[SEED]
23 42
```

Figura 3.11: Definição de cenário em função da rede de comunicação.

cenário serão executados para cada um destes algoritmos, de tal forma que para cada uma das iterações do cenário todos os algoritmos são executados como exatamente a mesma seqüência de mensagens e *checkpoints* básicos.

As quatro seções seguintes, `[TYPE]`, `[RANGE]`, `[NETWORKS]` e `[PRIORITIES]` definem os parâmetros do sistema a ser modelado. A seção `[TYPE]` informa se o teste será em função da rede de comunicação (`net`) ou em função das prioridades (`prio`) e deve conter apenas uma palavra com o rótulo apropriado. A Figura 3.12 mostra uma definição em função das prioridades.

A próxima seção, `[RANGE]`, define quantos modelos serão testados e quais valores devem ser associados a cada um dos modelos. Esta seção deve conter uma linha com três números <`início`, `fim`, `passo`>, os modelos são numerados a partir de `início` até `fim`, em incrementos de `passo`. Os testes seguirão os números desta seqüência, variando a rede de comunicação ou as prioridades, que serão definidas nas seções `[NETWORKS]` e `[PRIORITIES]`, e executando para cada uma das definições um número de iterações a ser definido na seção `[ITERATIONS]`.

A seção `[NETWORKS]` especifica as redes de comunicação a partir das quais os modelos serão gerados, devendo conter nomes de arquivos de definição de rede, separados por espaços ou quebras de linha. Caso o tipo de teste seja `net`, a seção `[NETWORKS]` é opcional. Caso exista, esta seção deve conter um arquivo para cada um dos números da seqüência definida na seção `[RANGE]`. A ordem de aparição destes arquivos na definição e a ordem de expansão de caracteres coringa pela *shell* dará a ordem de associação entre os arquivos e os números da seqüência. Caso esta seção não esteja presente (Figura 3.11) são gerados para cada um dos números da seqüência uma rede de comunicação equivalente a um grafo completo, com o número de processos igual ao número da seqüência. Se o tipo de teste for `prio`, é obrigatória a presença da seção `[NETWORKS]` com exatamente um nome de arquivo de definição de rede de comunicação (Figura 3.12).

Analogamente à seção `[NETWORKS]`, a seção `[PRIORITIES]` especifica as prioridades a partir das quais os modelos serão gerados, devendo conter nomes de arquivos de prioridades, separados por espaços ou quebras de linha. Esta seção deve estar sempre presente e também tem o seu conteúdo em função do tipo do teste. Se o teste for `net`, esta seção deve conter exatamente um nome de arquivo de definição de prioridades (Figura 3.11). Caso o teste seja do tipo `prio` (Figura 3.12) então a seção deve conter uma lista de arquivos de definição de prioridades, que serão expandidos e associados aos números da seqüência definida na seção `[RANGE]` como descrito no parágrafo anterior.

As últimas quatro seções configuram características do cenário que não estão diretamente relacionadas ao modelo. A seção `[WHILE]` define uma condição, em Promela, que quando falsa encerra a execução dos processos individuais. Esta condição é a mesma usada pelo programa `metapromela`, podendo fazer uso das variáveis de modelo `mp_basic`,

```
# Exemplo de definição de cenário (Prioridades)

[ALGORITHMS]
algoritmos/bcs.mprml algoritmos/bcs-aftersend.mprml

[TYPE]
prio

[RANGE]
1 20 1

[NETWORKS]
redes/complete6

[PRIORITIES]
prioridades/acelerando?
prioridades/acelerando1?
prioridades/acelerando20

[WHILE]
mp_basic < (300*mp_n)

[DIVIDE]
no

[ITERATIONS]
10

[SEED]
69 5
```

Figura 3.12: Definição de cenário em função das prioridades.

`mp_forced`, `mp_sentmess` e `mp_receivedmess`.

A seção `[DIVIDE]` indica se o resultado, medido em número de *checkpoints* forçados, deve ser dividido pelo número de processos da aplicação. Esta divisão só faz sentido quando trabalhamos com testes do tipo `net` e estamos variando o número de processos da aplicação. Se o tipo do teste for `prio` esta seção é ignorada.

A seção `[ITERATIONS]` define o número de iterações para cada um dos modelos gerados. Para cada uma das iterações é gerado uma seqüência diferente de mensagens e *checkpoints* básicos e esta seqüência é aplicada sobre cada um dos algoritmos.

Por último, a seção `[SEED]` define a semente a ser usada pelo gerador de números pseudo-aleatórios. A definição deve conter uma linha com dois números: <`semente`, `incremento`>. Onde `semente` é o número usado como semente na primeira iteração, ao qual é adicionado `incremento` para as iterações subseqüentes.

3.5.2 Uso da Ferramenta

Para fazer as simulações referentes a um cenário, usamos o programa `run` da seguinte forma:

```
run <cenário>
```

Onde `cenário` é o nome de arquivo da definição de cenário. Durante a execução das simulações o programa exibe na tela mensagens indicando o progresso do cenário.

Terminada a simulação, são gerados três arquivos de saída no diretório corrente: <`cenário`>`.rawdata`, <`cenário`>`.data` e <`cenário`>`.plot`. O primeiro arquivo contém todos os dados coletados em cada uma das iterações, para todos os algoritmos. O segundo arquivo contém apenas as médias de todas as iterações para todos os algoritmos e o arquivo <`cenário`>`.plot` contém um esboço de configuração do GnuPlot que usa o arquivo <`cenário`>`.data`.

O arquivo <`cenário`>`.plot` pode ser usado para uma visualização do estado do teste ainda durante a sua execução. Os arquivos são atualizados com os dados mais recentes após o término de todas as iterações para um dado modelo, sendo possível acompanhar os dados à medida que os mesmos são coletados ou interromper os testes quando necessário sem perder os dados já obtidos.

Capítulo 4

Algoritmos Estudados

Neste Capítulo apresentamos os algoritmos quase-síncronos para *checkpointing* utilizados em nosso estudo comparativo. A apresentação de cada algoritmo contém a sua especificação em pseudo-código e uma breve discussão sobre o seu comportamento. A descrição é, na nossa opinião, suficiente para que o leitor analise o comportamento integral do algoritmo de forma simples e intuitiva. Evitamos um tratamento baseado no formalismo da apresentação original do algoritmo em nome da uniformidade e simplicidade da apresentação.

O pseudo-código usado para especificar os algoritmos é dividido em função dos eventos principais dos processos da aplicação: *checkpoint* básico, envio e recepção de mensagens. Existem também seções referentes a declaração de variáveis e ao início do algoritmo. É definida também uma constante "pid" para representar o identificador do processo que executa o algoritmo.

Os algoritmos estão organizados de acordo com o padrão de *checkpoints* gerado. Começamos na Seção 4.1 descrevendo os algoritmos que geram padrão ZPF (*z-path free*) e passamos em seguida na Seção 4.2 para os algoritmos que geram padrão ZCF (*z-cicle free*).

A seleção dos algoritmos foi feita de forma abrangente, procurando englobar o maior número possível de algoritmos reconhecidos e citados em trabalhos na área. No entanto, deixamos de fora alguns algoritmos que, apesar de reconhecidos, não se enquadravam nos parâmetros do nosso estudo, notavelmente os algoritmos que geram padrão PZCF. Uma breve descrição destes algoritmos e do porquê eles não foram considerados no estudo é feita na Seção 4.3.

4.1 Algoritmos ZPF

Algoritmos que geram padrões de *checkpoints* ZPF foram umas das primeiras formas encontradas de se evitar o efeito dominó no contexto de tolerância a falhas [26, 27]. Este padrão apresenta propriedades interessantes que o torna útil não só para recuperação por retrocesso mas também para outras aplicações, como por exemplo *breakpoints* distribuídos causais [10, 30].

Os algoritmos desta classe são chamados de baseados em modelo e em suas formas mais simples, tomam a decisão de forçar um *checkpoint* baseando-se apenas no regime de troca de mensagens e não nas relações de dependência entre os *checkpoints*.

Wang [30] identificou quatro algoritmos propostos anteriormente que se preocupam apenas com o regime de troca de mensagens: CASBR, CAS, CBR e NRAS. Estes algoritmos tem o seu nome tirado do padrão de comportamento a que eles restringem a aplicação e, de forma geral, são extremamente custosos em termos do número de *checkpoints* forçados.

O algoritmo CASBR (*Checkpoint-After-Send-Before-Receive*) induz o processo a tirar um *checkpoint* forçado sempre após o envio e antes da recepção de uma mensagem. O algoritmo CAS (*Checkpoint-After-Send*) induz o processo a tirar um *checkpoint* forçado sempre após o envio de uma mensagem. O algoritmo CBR (*Checkpoint-Before-Receive*) induz o processo a tirar um *checkpoint* forçado sempre antes da recepção de uma mensagem. Estes três algoritmos são extremamente simples de serem implementados, geram um padrão ZPF [22] e, apesar de seu custo, têm propriedades que podem ser úteis em algumas aplicações [30]. Exemplos de padrões de *checkpoints* gerados por estes algoritmos estão nas Figuras 4.1, 4.2 e 4.3. As descrições destes algoritmos estão nos Algoritmos 4.1, 4.2 e 4.3.

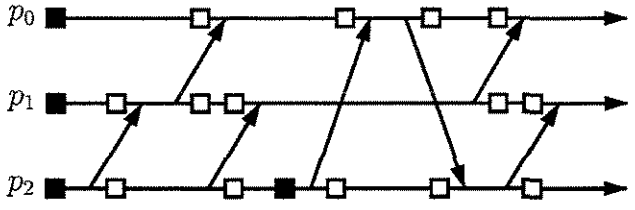


Figura 4.1: Exemplo do padrão gerado pelo algoritmo CASBR.

O algoritmo NRAS (*No-Receive-After-Send*) foi um dos primeiros algoritmos propostos para evitar o efeito dominó em um sistema de recuperação de falhas por retrocesso [26]. Este algoritmo força um *checkpoint* sempre antes de receber uma mensagem após já ter enviado uma outra mensagem no mesmo intervalo. Um exemplo do padrão de *check-*

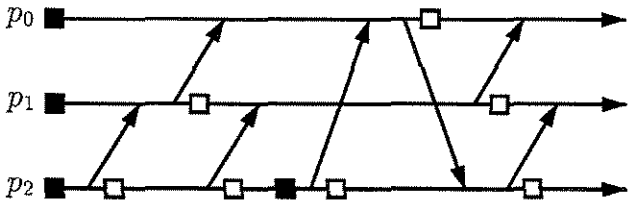


Figura 4.2: Exemplo do padrão gerado pelo algoritmo CAS.

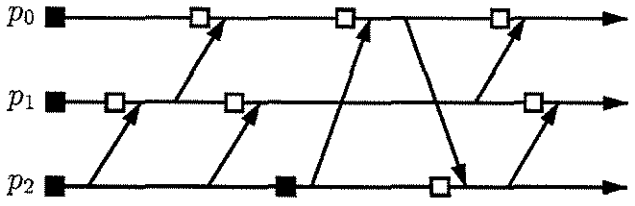


Figura 4.3: Exemplo do padrão gerado pelo algoritmo CBR.

Algoritmo 4.1 Algoritmo CASBR

Início:
Armazena o *checkpoint* inicial

***Checkpoint* básico:**
Armazena o *checkpoint* básico

Envio da mensagem (m) para p_k:
Envia a mensagem (m)
Armazena o *checkpoint* forçado

Recebimento da mensagem (m) de p_k:
Armazena o *checkpoint* forçado
Recebe a mensagem (m)

Algoritmo 4.2 Algoritmo CAS

Início:
Armazena o *checkpoint* inicial

***Checkpoint* básico:**
Armazena o *checkpoint* básico

Envio da mensagem (m) para p_k:
Envia a mensagem (m)
Armazena o *checkpoint* forçado

Recebimento da mensagem (m) de p_k:
Recebe a mensagem (m)

Algoritmo 4.3 Algoritmo CBR

Início:
Armazena o *checkpoint* inicial

***Checkpoint* básico:**
Armazena o *checkpoint* básico

Envio da mensagem (m) para p_k:
Envia a mensagem (m)

Recebimento da mensagem (m) de p_k:
Armazena o *checkpoint* forçado
Recebe a mensagem (m)

points gerado por este algoritmo está na Figura 4.4 e a descrição deste algoritmo está no Algoritmo 4.4.

Intuitivamente é possível observar que estes algoritmos geram um padrão ZPF porque eles evitam que exista um envio e uma recepção de mensagem em seqüência dentro de qualquer intervalo de *checkpoints* dos processos da aplicação. Desta forma, não é possível que ocorra a seqüência de mensagens necessária para um caminho-z não causal, independente destes caminhos estarem duplicados causalmente ou não. O algoritmo NRAS é o que executa esta quebra de forma mais objetiva, forçando um *checkpoint* somente entre o envio e a recepção de mensagens.

Claramente estes algoritmos forçam um número maior de *checkpoints* do que o necessário para tornar o padrão ZPF. O que falta para melhorar a eficiência destes algoritmos é uma maneira de detectar a presença de caminhos-z não causais duplicados causalmente.

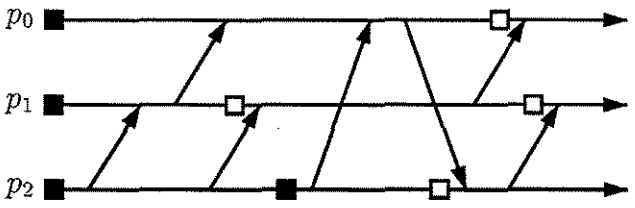


Figura 4.4: Exemplo do padrão gerado pelo algoritmo NRAS.

Para tanto é necessário acompanhar a relação de precedência causal.

O algoritmo FDI (*Fixed-Dependency-Interval*) [27] usa um vetor de relógios [25] para acompanhar a precedência causal entre os *checkpoints* e induz um *checkpoint* forçado sempre que uma mensagem traz informações sobre novos *checkpoints*. Ocorre então que o vetor de relógios (ou de dependência entre *checkpoints*) permanece o mesmo dentro de cada um dos intervalos de *checkpoints* e este comportamento gera um padrão ZPF [30]. Na prática este algoritmo exibe uma melhora no padrão de *checkpoints* quando não força um *checkpoint* ao receber uma mensagem que traz informação apenas sobre dependências conhecidas, o que implica que qualquer caminho-z que poderia se formar com o recebimento desta mensagem é duplicado causalmente.

Um exemplo do padrão de *checkpoints* gerado pelo algoritmo FDI está na Figura 4.5 e a descrição deste algoritmo está no Algoritmo 4.5.

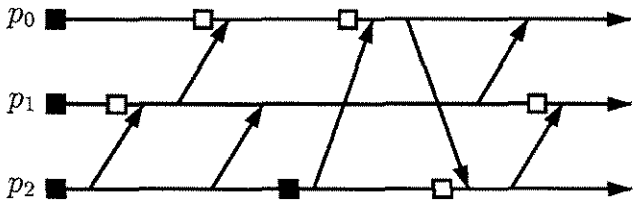


Figura 4.5: Exemplo do padrão gerado pelo algoritmo FDI.

No entanto, o algoritmo FDI leva apenas em conta a dependência causal, ignorando as relações entre as mensagens do sistema. O algoritmo FDAS (*Fixed-Dependency-After-Send*) [30] combina o relógio do algoritmo FDI com a observação do algoritmo NRAS de que um caminho-z não causal só se forma após algum envio de mensagem. Desta forma, este algoritmo quebra quaisquer caminhos-z, exceto aqueles causalmente duplicados no intervalo corrente, gerando um padrão ZPF [30]. Um exemplo do padrão de *checkpoints* gerado por este algoritmo está na Figura 4.6 e a descrição deste algoritmo está no Algoritmo 4.6.

Algoritmo 4.4 Algoritmo NRAS

Variáveis:
sent ≡ booleano

Início:
sent := **false**
Armazena o *checkpoint* inicial

***Checkpoint* básico:**
sent := **false**
Armazena o *checkpoint* básico

Envio da mensagem (m) para p_k:
sent := **true**
Envia a mensagem (m)

Recebimento da mensagem (m) de p_k:
se sent :
    sent := **false**
    Armazena o *checkpoint* forçado
Recebe a mensagem (m)

Algoritmo 4.5 Algoritmo FDI

Variáveis:
dv ≡ **vetor**$[0 \ldots n-1]$ **de** inteiros

Início:
∀ i : dv[i] := 0
dv[pid] := dv[pid] + 1
Armazena o *checkpoint* inicial

***Checkpoint* básico:**
dv[pid] := dv[pid] + 1
Armazena o *checkpoint* básico

Envio da mensagem (m) para p_k:
Envia a mensagem (m, dv)

Recebimento da mensagem (m, m.dv) de p_k:
se m.dv[k] > dv[k] :
    dv[pid] := dv[pid] + 1
    Armazena o *checkpoint* forçado
∀ i : dv[i] := max(dv[i], m.dv[i])
Recebe a mensagem (m)

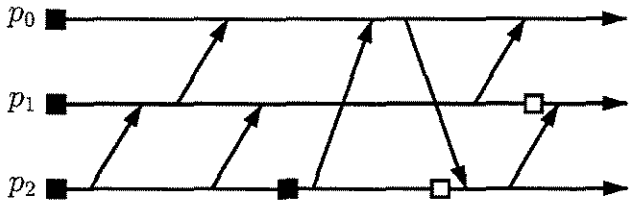


Figura 4.6: Exemplo do padrão gerado pelo algoritmo FDAS.

Algoritmo 4.6 Algoritmo FDAS

Variáveis:
dv ≡ **vetor** [0 . . . $n-1$] **de** inteiros
sent ≡ **booleano**

Início:
∀ i : dv[i] := 0
sent := **false**
dv[pid] := dv[pid] + 1
Armazena o *checkpoint* inicial

***Checkpoint* básico:**
sent := **false**
dv[pid] := dv[pid] + 1
Armazena o *checkpoint* básico

Envio da mensagem (m) para p_k:
sent := **true**
Envia a mensagem (m, dv)

Recebimento da mensagem (m, m.dv) de p_k:
se sent ∧ m.dv[k] > dv[k] :
    sent := **false**
    dv[pid] := dv[pid] + 1
    Armazena o *checkpoint* forçado
∀ i : dv[i] := max(dv[i], m.dv[i])
Recebe a mensagem (m)

O algoritmo RDT-Partner [13] faz uso da observação de que quando um processo p_i recebe uma mensagem de um processo p_j após ter enviado apenas mensagens m' para p_j, ele só precisa se preocupar em quebrar o caminho-z formado por m e m' caso este caminho-z implique na formação de um ciclo-z. Esta pequena otimização, quando aliada aos mecanismos usados pelo algoritmo FDAS, permite evitar alguns *checkpoints* forçados, principalmente se na aplicação ocorrem padrões de troca de mensagem na forma de pedido e resposta. Um exemplo do padrão de *checkpoints* gerado pelo algoritmo RDT-Partner está na Figura 4.7 e a descrição deste algoritmo está no Algoritmo 4.7.

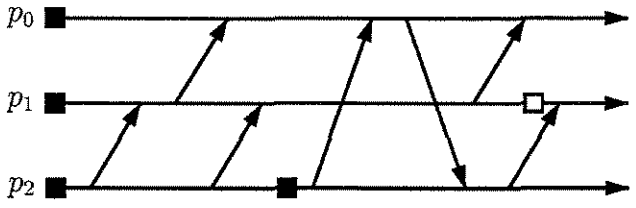


Figura 4.7: Exemplo do padrão gerado pelo algoritmo RDT-Partner.

Na tentativa de evitar a quebra desnecessária de caminhos-z causalmente duplicados, o algoritmo BHMR [3] procura levar em conta toda a informação de causalidade disponível no momento de decidir se força ou não um *checkpoint*. Para tanto, este protocolo propaga uma matriz de booleanos que carrega toda a história causal da mensagem sendo recebida, semelhante a uma matriz de relógios [25]. De posse desta informação o algoritmo procura determinar se, no contexto da informação disponível, o caminho-z não causal sendo criado está causalmente duplicado. Um exemplo do padrão de *checkpoints* gerado por este algoritmo está na Figura 4.8 e a descrição deste algoritmo está no Algoritmo 4.8.

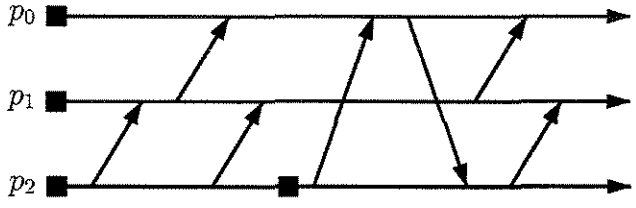


Figura 4.8: Exemplo do padrão gerado pelo algoritmo BHMR.

Algoritmo 4.7 Algoritmo RDT-Partner

Variáveis:
dv ≡ **vetor**[0...$n-1$] **de** inteiros
simple ≡ **vetor**[0...$n-1$] **de** booleanos
partner ≡ inteiro

Início:
∀ i : dv[i] := 0
∀ i : simple[i] := **false**
dv[pid] := dv[pid] + 1
simple[pid] := **true**
partner := NO_PARTNER
Armazena o *checkpoint* inicial

***Checkpoint* básico:**
∀ i : simple[i] := **false**
dv[pid] := dv[pid] + 1
simple[pid] := **true**
partner := NO_PARTNER
Armazena o *checkpoint* básico

Envio da mensagem (m) para p_k:
se partner = NO_PARTNER :
    partner := k
senão se partner ≠ k :
    partner := MORE_THAN_ONE_PARTNER
Envia a mensagem (m, dv, simple[k])

Recebimento da mensagem (m, m.dv, m.simple) de p_k:
se (m.dv[k] > dv[k]) :
    se (partner ≠ NO_PARTNER) ∧
      (partner ≠ k ∨ (partner = k ∧ m.dv[pid] = dv[pid] ∧ ¬m.simple)) :
        ∀ i : simple[i] := **false**
        dv[pid] := dv[pid] + 1
        simple[pid] := **true**
        partner := NO_PARTNER
        Armazena o *checkpoint* forçado
    simple[k] := **true**
∀ i : dv[i] := max(dv[i], m.dv[i])
Recebe a mensagem (m)

Algoritmo 4.8 Algoritmo BHMR

Variáveis:
dv ≡ **vetor**[0...$n-1$] **de** inteiros
simple ≡ **vetor**[0...$n-1$] **de** booleanos
sent ≡ **vetor**[0...$n-1$] **de** booleanos
causal ≡ **vetor**[0...$n-1$, 0...$n-1$] **de** booleanos

Início:
∀ i : dv[i] := 0
∀ i : sent[i] := **false**
∀ i ≠ pid : simple[i] := **false**
∀ i ≠ j : causal[i, j] := **false**
dv[pid] := dv[pid] + 1
simple[pid] := **true**
∀ i : causal[i, i] := **true**
Armazena o *checkpoint* inicial

***Checkpoint* básico:**
dv[pid] := dv[pid] + 1
∀ i : sent[i] := **false**
∀ i ≠ pid : simple[i] := **false**, causal[pid, i] := **false**
Armazena o *checkpoint* básico

Envio da mensagem (m) para p_k:
sent[k] := **true**
Envia a mensagem (m, dv, simple, causal)

Recebimento da mensagem (m, m.dv, m.simple, m.causal) de p_k:
c1 := (dv[pid] = m.dv[pid]) ∧ ¬m.simple[pid]
c2 := ∃ i : (sent[i] ∧ ∃ j : ((m.dv[j] > dv[j]) ∧ ¬ m.causal[j, i]))
se c1 ∨ c2 :
    dv[pid] := dv[pid] + 1
    ∀ i : sent[i] := **false**
    ∀ i ≠ pid : simple[i] := **false**, causal[pid, i] := **false**
    Armazena o *checkpoint* forçado
∀ i : **se** m.dv[i] > dv[i] :
        dv[i] := m.dv[i]
        simple[i] := m.simple[i]
        ∀ j : causal[i, j] := m.causal[i, j]
    senão se m.dv[i] = dv[i] :
        simple[i] := simple[i] ∧ m.simple[i]
        ∀ j : causal[i, j] := causal[i, j] ∨ m.causal[i, j]
causal[k, pid] := true
∀ i : causal[i, pid] := causal[i, pid] ∨ causal[i, k]
Recebe a mensagem (m)

4.2 Algoritmos ZCF

Algoritmos que geram padrões de *checkpoints* ZCF impõem sobre a aplicação regras menos rígidas do que os algoritmos que geram padrões ZPF. Como reflexo desta menor interferência os algoritmos tendem a ser mais simples e propagar menos informação de controle. No entanto, dependendo do padrão de comunicação da aplicação, nem sempre é possível obter os mais recentes *checkpoints* globais consistentes determinados pelos *checkpoints* locais dos processos.

A grande maioria dos algoritmos ZCF usam apenas um inteiro como identificador de *checkpoints* e como informação de controle, por isso são conhecidos como baseados em índice. O primeiro algoritmo proposto nesta classe foi o BCS [6], que apesar de sua simplicidade possui as características principais de todos os outros algoritmos desta classe.

O algoritmo BCS usa a relação de precedência causal para determinar quando deve induzir um *checkpoint* forçado. Para tanto, o algoritmo propaga um relógio lógico escalar muito semelhante ao relógio proposto por Lamport [20, 25], com a diferença de que o mesmo é incrementado apenas na ocorrência de um *checkpoint* básico. Ao receber uma mensagem, caso esta traga como informação de controle um relógio maior do que o atual, ou seja, informação causal sobre um novo *checkpoint* em algum dos outros processos, é forçado um *checkpoint*. Este comportamento gera um padrão ZCF [22] e intuitivamente podemos ver que o padrão é garantido devido ao fato de que uma mensagem m com informação causal de algum *checkpoint* $\hat{\sigma}$ nunca é recebida no mesmo intervalo de *checkpoints* onde uma mensagem m' tenha sido enviada, tal que m' possa preceder causalmente $\hat{\sigma}$. Desta forma, quebram-se todos os ciclos-z.

Um exemplo do padrão de *checkpoints* gerado pelo algoritmo BCS está na Figura 4.9. Nas figuras que representam algoritmos baseados em índice, estarão marcados sobre os *checkpoints* os índices a eles associados e entre parênteses os índices propagados pelas mensagens. A descrição do algoritmo BCS está no Algoritmo 4.9.

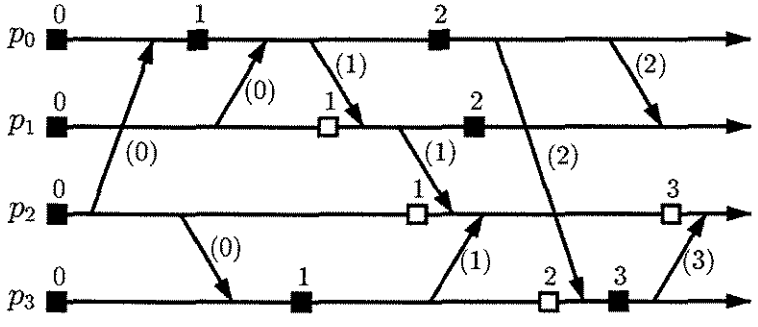


Figura 4.9: Exemplo do padrão gerado pelo algoritmo BCS.

Algoritmo 4.9 Algoritmo BCS

Variáveis:
lc ≡ inteiro

Início:
lc := 0
Armazena o *checkpoint* inicial

***Checkpoint* básico:**
lc := lc + 1
Armazena o *checkpoint* básico

Envio da mensagem (m) para p_k:
Envia a mensagem (m, lc)

Recebimento da mensagem (m, m.lc) de p_k:
se m.lc > lc :
    Armazena o *checkpoint* forçado
    lc := m.lc
Recebe a mensagem (m)

O algoritmo BCS é muito simples e eficiente, mas pode ainda ser melhorado com algumas otimizações. De forma similar ao algoritmo FDI, o algoritmo BCS não leva em conta a estrutura da comunicação, forçando um *checkpoint* mesmo antes que o processo tenha enviado alguma mensagem no mesmo intervalo. Nesta situação não pode existir um caminho-z, implicando na não existência de um ciclo-z. Podemos então construir um algoritmo baseado em índice semelhante ao FDAS [30], sendo este algoritmo já proposto como parte de outro algoritmo mais complexo (BQF [5], a ser descrito ainda nesta Seção). Chamamos este algoritmo de BCS-Aftersend, um exemplo do padrão de *checkpoints* gerado por este algoritmo está na Figura 4.10 e a descrição deste algoritmo está no Algoritmo 4.10.

Podemos também detectar a mesma estrutura de comunicação explorada pelo algoritmo RDT-Partner para evitar de forçar *checkpoints* em situações de troca de mensagens na forma de pedido e resposta. Chamamos este algoritmo de BCS-Partner e ele evita de forçar *checkpoints* nas situações onde a mensagem recebida por um processo p_i, com valor de relógio maior que o atual, se origina do único processo para o qual p_i enviou uma mensagem e não existe a formação de um ciclo-z. Este ciclo-z específico pode ser detectado usando-se apenas informações propagadas causalmente [13]. Um exemplo do padrão de *checkpoints* gerado por este algoritmo está na Figura 4.11 e a descrição deste algoritmo está no Algoritmo 4.11.

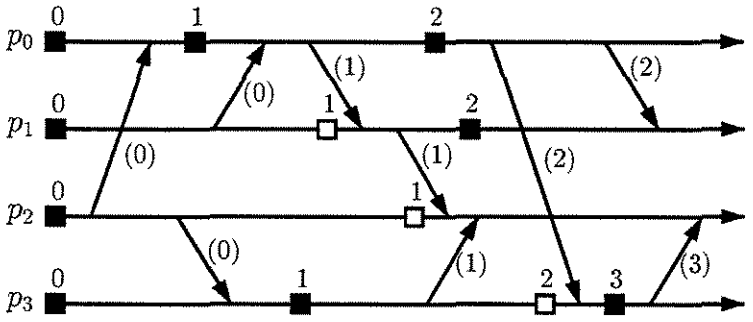


Figura 4.10: Exemplo do padrão gerado pelo algoritmo BCS-Aftersend.

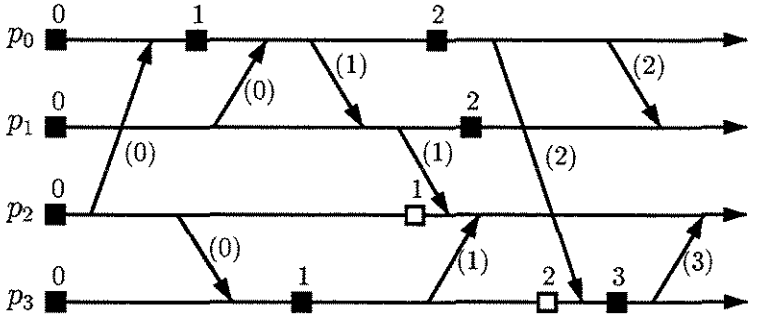


Figura 4.11: Exemplo do padrão gerado pelo algoritmo BCS-Partner.

Algoritmo 4.10 Algoritmo BCS-Aftersend

Variáveis:
lc ≡ inteiro
sent ≡ booleano

Início:
lc := 0
sent := **false**
Armazena o *checkpoint* inicial

***Checkpoint* básico:**
lc := lc + 1
Armazena o *checkpoint* básico
sent := **false**

Envio da mensagem (m) para p_k:
sent := **true**
Envia a mensagem (m, lc)

Recebimento da mensagem (m, m.lc) de p_k:
se m.lc > lc :
    se sent :
        Armazena o *checkpoint* forçado
        sent := **false**
    lc := m.lc
Recebe a mensagem (m)

Algoritmo 4.11 Algoritmo BCS-Partner

Variáveis:
lc ≡ inteiro
partner ≡ inteiro
dv ≡ **vetor**[0 . . . $n-1$] **de** inteiros
simple ≡ **vetor**[0 . . . $n-1$] **de** booleanos

Início:
lc := 0
∀ i : dv[i] := 0
∀ i : simple[i] := **false**
dv[pid] := dv[pid] + 1
simple[pid] := **true**
partner := NO_PARTNER
Armazena o *checkpoint* inicial

***Checkpoint* básico:**
lc := lc + 1
dv[pid] := dv[pid] + 1
∀ i : simple[i] := **false**
simple[pid] := **true**
partner := NO_PARTNER
Armazena o *checkpoint* básico

Envio da mensagem (m) para p_k:
se partner = NO_PARTNER :
    partner := k
senão se partner ≠ k :
    partner := MORE_THAN_ONE_PARTNER
Envia a mensagem (m, lc, simple[k], dv[k], dv[pid])

Recebimento da mensagem (m, m.lc, m.simple, m.dv_receiver, m.dv_sender) de p_k:
se (m.lc > lc) ∧ partner ≠ NO_PARTNER :
    se partner ≠ k ∨ (partner = k ∧ m.dv_receiver = dv[pid] ∧ ¬m.simple) :
        dv[pid] := dv[pid] + 1
        ∀ i : simple[i] := **false**
        simple[pid] := **true**
        partner := NO_PARTNER
        Armazena o *checkpoint* forçado
lc := max(lc, m.lc)
se m.dv_sender > dv[k] :
    dv[k] := m.dv_sender
    simple[k] := **true**
Recebe a mensagem (m)

O algoritmo HMNR [15, 16] estende a noção de parceiro a um grupo de processos. O algoritmo permite que um processo p_i, mesmo após ter enviado mensagens para vários processos, possa receber uma mensagem m de p_j, com relógio maior do que o atual, sem ter que forçar um *checkpoint* se a mensagem m trouxer informações de que as mensagens já enviadas por p_i não podem fechar um ciclo-z passando por p_j. Para tanto, este algoritmo propaga três vetores além do índice usado pelos outros algoritmos. Um exemplo do padrão de *checkpoints* gerado por este algoritmo está na Figura 4.12 e a descrição deste algoritmo está no Algoritmo 4.12.

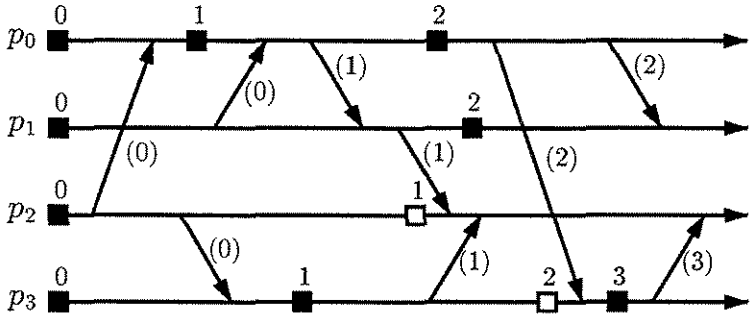


Figura 4.12: Exemplo do padrão gerado pelo algoritmo HMNR.

Uma outra forma de explorarmos o padrão de mensagens para evitar a ocorrência de *checkpoints* forçados em algoritmos que geram padrões ZCF usando um índice vem da observação de que é o incremento deste índice na ocorrência de um *checkpoint* básico que pode induzir *checkpoints* forçados em outros processos. Naturalmente, o incremento dos índices é o que garante a correção do algoritmo, mas existem situações onde um *checkpoint* básico pode ser agendado sem que seja necessário incrementar o índice associado ao mesmo. O algoritmo Lazy-BCS [28] faz uso da condição mais simples na qual o índice não precisa ser incrementado: ao tirar um *checkpoint* básico em um processo que somente recebeu mensagens de outros processos com índices estritamente menores que seu índice atual [5].

Intuitivamente este comportamento preguiçoso dos índices continua garantido que não existem ciclos-z, pois se um processo só recebeu mensagens com índices estritamente menores então o último *checkpoint* não foi tirado obrigatoriamente para manter a consistência, isto é, quebrar um ciclo-z, e o próximo *checkpoint* pode usar o mesmo índice. Um exemplo do padrão de *checkpoints* gerado pelo algoritmo Lazy-BCS está na Figura 4.13 e a descrição deste algoritmo está no Algoritmo 4.13.

As otimizações usadas pelos algoritmos BCS-Aftersend e BCS-Partner usam estruturas diferentes daquelas exploradas pelo Lazy-BCS e podem ser combinadas. Temos então

Algoritmo 4.12 Algoritmo HMNR

Variáveis:
lc ≡ inteiro
dv ≡ **vetor**[0...$n-1$] **de** inteiros
simple ≡ **vetor**[0...$n-1$] **de** booleanos
synch ≡ **vetor**[0...$n-1$] **de** booleanos
sent_to ≡ **vetor**[0...$n-1$] **de** booleanos

Início:
lc := 0
∀ i : dv[i] := 0
∀ i : simple[i] := **false**, synch[i] := **false**, sent_to[i] := **false**
dv[pid] := dv[pid] + 1
simple[pid] := **true**
synch[pid] := **true**
Armazena o *checkpoint* inicial

***Checkpoint* básico:**
lc := lc + 1
dv[pid] := dv[pid] + 1
∀ i ≠ pid : simple[i] := **false**, synch[i] := **false**
∀ i : sent_to[i] := **false**
Armazena o *checkpoint* básico

Envio da mensagem (m) para p_k:
sent_to[k] := **true**
Envia a mensagem (m, lc, dv, synch, simple)

Recebimento da mensagem (m, m.lc, m.dv, m.synch, m.simple) de p_k:
se m.lc > lc :
    se (∃ i : sent_to[i] ∧ ¬m.synch[i]) ∨ (m.dv[pid] = dv[pid] ∧ ¬m.simple[pid]) :
        dv[pid] := dv[pid] + 1
        ∀ i ≠ pid : simple[i] := **false**, synch[i] := **false**
        ∀ i : sent_to[i] := **false**
        Armazena o *checkpoint* forçado
    lc := m.lc
    ∀ i ≠ pid : synch[i] := m.synch[i]
    synch[pid] := **true**
senão se m.lc = lc :
    ∀ i : synch[i] := synch[i] ∨ m.synch[i]
∀ i ≠ pid :
    se m.dv[i] > dv[i] :
        dv[i] := m.dv[i]
        simple[i] := m.simple[i]
    senão se m.dv[i] = dv[i] :
        simple[i] := simple[i] ∧ m.simple[i]
Recebe a mensagem (m)

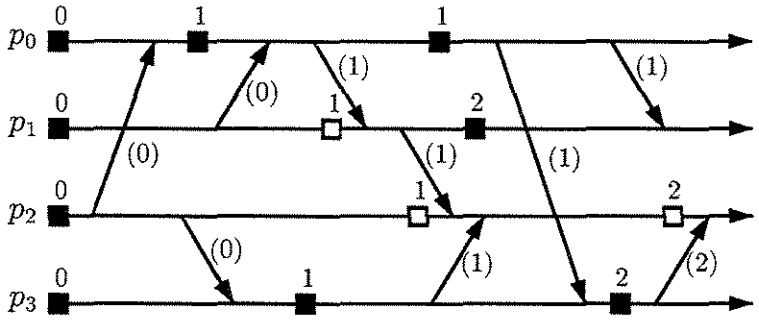


Figura 4.13: Exemplo do padrão gerado pelo algoritmo Lazy-BCS.

Algoritmo 4.13 Algoritmo Lazy-BCS

Variáveis:
lc ≡ inteiro
equiv ≡ booleano

Início:
lc := 0
equiv := **true**
Armazena o *checkpoint* inicial

***Checkpoint* básico:**
se ¬equiv :
lc := lc + 1
equiv := **true**
Armazena o *checkpoint* básico

Envio da mensagem (m) para p_k:
Envia a mensagem (m, lc)

Recebimento da mensagem (m, m.lc) de p_k:
se m.lc > lc :
Armazena o *checkpoint* forçado
lc := m.lc
equiv := **false**
senão se m.lc = lc :
equiv := **false**
Recebe a mensagem (m)

os algoritmos Lazy-BCS-Aftersend [28] e Lazy-BCS-Partner, que são apenas a concatenação de dois algoritmos procurando ao mesmo tempo evitar de forçar *checkpoints* e de incrementar os índices. Estas simples combinações resultam em algoritmos extremamente eficientes.

Exemplos dos padrões de *checkpoints* gerado pelos algoritmos Lazy-BCS-Aftersend e Lazy-BCS-Partner estão nas Figuras 4.14 e 4.15, respectivamente. A descrição do algoritmo Lazy-BCS-Aftersend está no Algoritmo 4.14 e a descrição do algoritmo Lazy-BCS-Partner está no Algoritmo 4.15.

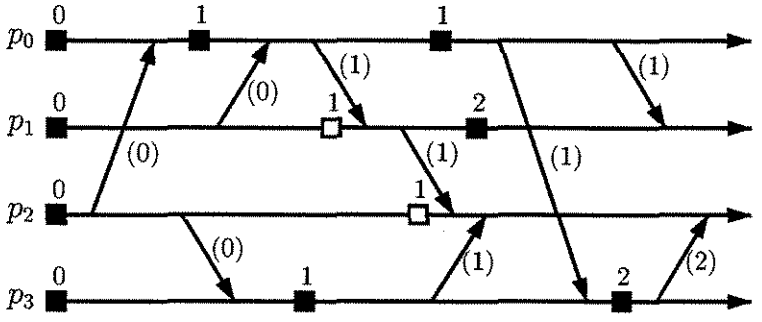


Figura 4.14: Exemplo do padrão gerado pelo algoritmo Lazy-BCS-Aftersend.

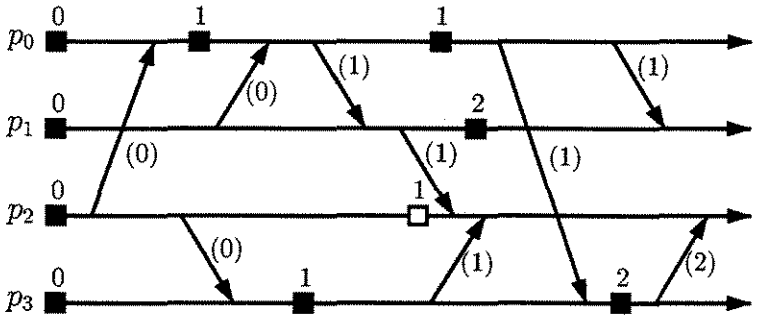


Figura 4.15: Exemplo do padrão gerado pelo algoritmo Lazy-BCS-Partner.

O algoritmo BQF [5] leva ainda mais adiante a tentativa de se tirar *checkpoints* básicos sem incrementar os índices. Ao contrário do algoritmo Lazy-BCS que decide se o índice pode ser aumentado ao armazenar o *checkpoint* básico, o algoritmo BQF procura retardar ao máximo esta decisão. O processo deve decidir que índice dar a um *checkpoint* básico já armazenado ao tirar o próximo *checkpoint* básico ou ao enviar uma mensagem, que deve carregar o índice definitivo. Ocorre um ganho sobre o algoritmo Lazy-BCS quando,

Algoritmo 4.14 Algoritmo Lazy-BCS-Aftersend

Variáveis:
lc ≡ inteiro
sent ≡ booleano
equiv ≡ booleano

Início:
lc := 0
sent := **false**
equiv := **true**
Armazena o *checkpoint* inicial

***Checkpoint* básico:**
se ¬equiv :
lc := lc + 1
equiv := **true**
Armazena o *checkpoint* básico
sent := **false**

Envio da mensagem (m) para p_k:
sent := **true**
Envia a mensagem (m, lc)

Recebimento da mensagem (m, m.lc) de p_k:
se m.lc > lc :
se sent :
Armazena o *checkpoint* forçado
sent := **false**
lc := m.lc
equiv := **false**
senão se m.lc = lc :
equiv := **false**
Recebe a mensagem (m)

Algoritmo 4.15 Algoritmo Lazy-BCS-Partner

Variáveis:
lc ≡ inteiro
partner ≡ inteiro
dv ≡ **vetor**[0 . . . $n-1$] **de** inteiros
simple ≡ **vetor**[0 . . . $n-1$] **de** booleanos
equiv ≡ booleano

Início:
lc := 0
∀ i : dv[i] := 0
∀ i : simple[i] := **false**
dv[pid] := dv[pid] + 1
simple[pid] := **true**
partner := NO_PARTNER
equiv := **true**
Armazena o *checkpoint* inicial

***Checkpoint* básico:**
se ¬equiv :
    lc := lc + 1
    equiv := **true**
dv[pid] := dv[pid] + 1
∀ i : simple[i] := **false**
simple[pid] := **true**
partner := NO_PARTNER
Armazena o *checkpoint* básico

Envio da mensagem (m) para p_k:
se partner = NO_PARTNER :
    partner := k
senão se partner ≠ k :
    partner := MORE_THAN_ONE_PARTNER
Envia a mensagem (m, lc, simple[k], dv[k], dv[pid])

Recebimento da mensagem (m, m.lc, m.simple, m.dv_receiver, m.dv_sender) de p_k:
se (m.lc > lc) ∧ partner ≠ NO_PARTNER :
    se partner ≠ k ∨ (partner = k ∧ m.dv_receiver = dv[pid] ∧ ¬m.simple) :
        dv[pid] := dv[pid] + 1
        ∀ i : simple[i] := **false**
        simple[pid] := **true**
        partner := NO_PARTNER
        Armazena o *checkpoint* forçado
se m.lc ≥ lc :
    equiv := **false**
lc := max(lc, m.lc)
se m.dv_sender > dv[k] :
    dv[k] := m.dv_sender
    simple[k] := **true**
Recebe a mensagem (m)

mesmo tendo recebido mensagens com o mesmo índice, o processo descobre que não precisa incrementar seu índice pois os processos dos quais dependia também tiraram *checkpoints* básicos e que todos não incrementaram seus índices. Para detectar esta situação, o algoritmo BQF propaga além do índice do *checkpoint* atual um vetor de inteiros.

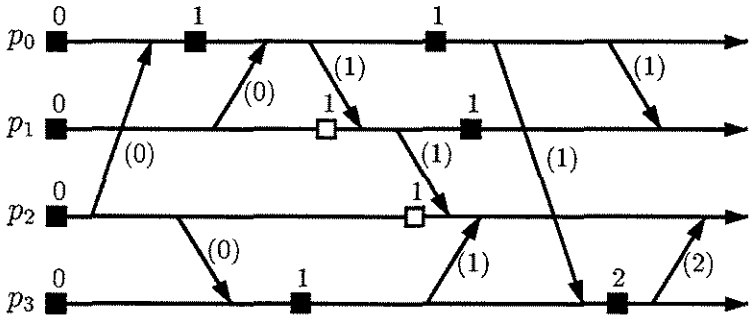


Figura 4.16: Exemplo do padrão gerado pelo algoritmo BQF.

O padrão de *checkpoints* gerado pelo algoritmo BQF é ZCF [5] e temos um exemplo deste padrão na Figura 4.16. O algoritmo BQF como descrito em [5] inclui a mesma otimização do BCS-Aftersend e ainda uma outra otimização proposta por Manivannan e Singhal [21]. Na implementação que usamos nas nossas comparações, esta última otimização foi retirada, pela mesma razão que o algoritmo proposto em [21] não foi considerado para o estudo (ver Seção 4.3). A descrição do algoritmo BQF está no Algoritmo 4.16.

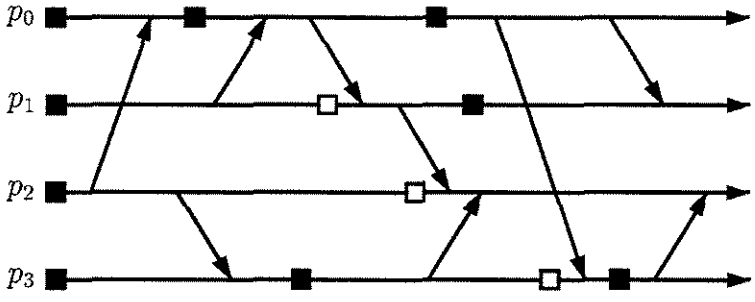


Figura 4.17: Exemplo do padrão gerado pelo algoritmo BQC.

O algoritmo BQC [4] usa uma abordagem diferente de todos os outros algoritmos desta Seção para garantir um padrão ZCF. Ao invés de fazer uso de um índice, ele usa um vetor de dependências e uma matriz para detectar a formação de uma estrutura chamada

Algoritmo 4.16 Algoritmo BQF

Variáveis:
eq ≡ **vetor**[0...$n-1$] **de** inteiros
past ≡ **vetor**[0...$n-1$] **de** inteiros
present ≡ **vetor**[0...$n-1$] **de** inteiros
lc ≡ inteiro
prov ≡ booleano
sent ≡ booleano

Início:
∀ i : eq[i] := 0, past[i] := -1, present[i] := -1
lc := 0, prov := **false**, sent := **false**
Armazena o *checkpoint* inicial com índice lc

***Checkpoint* básico:**
se prov ∧ (∃ i : past[i] > -1) :
    lc := lc + 1
    ∀ i : eq[i] := 0, past[i] := -1
    Atribui índice lc para o *checkpoint* anterior
senão
    ∀ i : past[i] := present[i]
    Deixa o índice provisório do *checkpoint* anterior como atribuído
eq[pid] := eq[pid] + 1
Armazena o *checkpoint* básico com índice provisório lc,eq[pid]
prov := **true**, sent := **false**
∀ i : present[i] := -1

Envio da mensagem (m) para p_k:
se prov ∧ (∃ i : past[i] > -1) :
    lc := lc + 1
    ∀ i : eq[i] := 0, past[i] := -1, present[i] := -1
    Atribui índice lc para o *checkpoint* anterior
prov := **false**, sent := **true**
Envia a mensagem (m, lc, eq)

Recebimento da mensagem (m, m.lc, m.eq) de p_k:
se m.lc > lc :
    se sent :
        Armazena o *checkpoint* forçado
        sent := **false**
    lc := m.lc
    ∀ i : eq[i] := m.eq[i], past[i] := -1, present[i] := -1
    Atribui índice lc para o *checkpoint* anterior
prov := **false**
present[k] := m.eq[k]
senão se m.lc = lc :
    se present[k] < m.eq[k] :
        present[k] := m.eq[k]
    ∀ i : eq[i] := max(eq[i], m.eq[i])
    ∀ i : **se** past[i] < m.eq[i] :
            past[i] := -1
Recebe a mensagem (m)

pelos autores do algoritmo de *ciclo-z suspeito*. Esta estrutura pode não fazer parte de um ciclo-z, mas todo ciclo-z deve obrigatoriamente contê-la [4]. O algoritmo então quebra qualquer ocorrência desta estrutura garantindo um padrão ZCF. Este algoritmo tem o comportamento muito parecido com o algoritmo FDAS, acrescentando a matriz para poder se limitar a quebrar os ciclos-z suspeitos. Um exemplo do padrão de *checkpoints* gerado por este algoritmo está na Figura 4.17, onde foram omitidos os índices que não são usados por esse algoritmo. A descrição deste algoritmo está no Algoritmo 4.17.

4.3 Algoritmos Não Comparados

Apesar da nossa intenção de tornar este estudo o mais abrangente possível, nós não avaliamos os algoritmos que geram padrões de *checkpoints* PZCF.

Em geral, os algoritmos PZCF são construídos, ou podem ser vistos, como simplificações de algoritmos que garantam um padrão ZPF ou ZCF. Por exemplo, o algoritmo proposto por Xu e Netzer [32] procura quebrar apenas os ciclos-z formados entre um par de processos como o algoritmo RDT-Partner, mas não garante a quebra de ciclos-z que envolvam mais de dois processos. O algoritmo proposto por Wang e Fuchs [31] é uma variação do algoritmo BCS que induz *checkpoints* apenas quando os índices são múltiplos de um determinado valor, garantindo apenas uma porcentagem de *checkpoints* úteis.

Para compararmos estes algoritmos é necessário, além de se contar o número de *checkpoints* forçados, contar o número de *checkpoints* inúteis. Avaliando desta forma o quanto estes algoritmos economizam em termos de *checkpoints* forçados e também o impacto que esta economia terá sobre o processo de obtenção de *checkpoints* globais consistentes. Para realizar a contagem de *checkpoints* inúteis é necessário efetivamente construir os *checkpoints* globais consistentes, construção essa que não tem uma implementação trivial no ambiente de simulação usado. Além deste fator, os vários tipos de algoritmos PZCF deriváveis dos algoritmos existentes, além dos já publicados, garantem por si só um estudo separado.

Além dos algoritmos que geram padrão PZCF, alguns outros algoritmos não foram considerados porque faziam suposições muito específicas sobre comportamentos aceitáveis pela aplicação. Nós estamos estudando apenas algoritmos quase-síncronos gerais, que não impõem restrição alguma sobre o padrão de *checkpoints* básicos dos processos individuais. Desta forma, deixamos fora do estudo as variantes do algoritmo proposto por Manivannan e Singhal [21].

Este algoritmo supõe especificamente uma situação de tolerância a falhas por recuperação de estado por retrocesso, usando um esquema de *checkpoints* periódicos. Neste contexto, os autores do algoritmo propõem a substituição de um *checkpoint* básico pelo último *checkpoint* forçado do processo, caso ele exista. Ocorre então uma diminuição do

Algoritmo 4.17 Algoritmo BQC

Variáveis:
dv ≡ **vetor**[0...$n-1$] **de** inteiros
ipred ≡ **vetor**[0...$n-1$] **de** inteiros
pred ≡ **vetor**[0...$n-1$, 0...$n-1$] **de** inteiros
sent ≡ booleano

Início:
∀ i : dv[i] := 0
dv[pid] := dv[pid] + 1
∀ i : ipred[i] := -1
∀ i,j : pred[i, j] := -1
sent := **false**
Armazena o *checkpoint* inicial

***Checkpoint* básico:**
∀ i : pred[pid, i] := max(pred[pid, i], ipred[i])
∀ i : ipred[i] := -1
dv[pid] := dv[pid] + 1
sent := **false**
Armazena o *checkpoint* básico

Envio da mensagem (m) para p_k:
sent := **true**
Envia a mensagem (m, dv, pred)

Recebimento da mensagem (m, m.dv, m.pred) de p_k:
se sent ∧ (∃ i : (m.dv[i] > dv[i]) ∧ (∃ j : m.pred[i, j]+1 > max(m.dv[j], dv[j]))) :
    ∀ i : pred[pid, i] := max(pred[pid, i], ipred[i])
    ∀ i : ipred[i] := -1
    dv[pid] := dv[pid] + 1
    sent := **false**
    Armazena o *checkpoint* forçado
∀ i : dv[i] := max(dv[i], m.dv[i])
∀ i,j : pred[i, j] := max(pred[i, j], m.pred[i, j])
ipred[k] := max(ipred[k], m.dv[k])
Recebe a mensagem (m)

número de *checkpoints* total da aplicação. No entanto, esta otimização não é geral o suficiente e não foi considerada em nosso estudo pois a seleção exata pelos processos de quais estados locais são *checkpoints* é essencial para problemas de depuração distribuída [10] e de detecção de predicados não-estáveis [19].

Capítulo 5

Análise e Conclusão

Neste Capítulo apresentaremos os resultados obtidos no nosso estudo comparativo dos algoritmos quase-síncronos para *checkpointing*. Iniciamos o Capítulo descrevendo as métricas que adotamos para a comparação do desempenho dos algoritmos, descrevendo também os modelos de simulação empregados para obter os dados. Em seguida, organizamos os algoritmos de acordo com suas propriedades de desempenho e analisamos, para cada classe de padrão gerado, os pontos positivos e negativos de cada algoritmo.

5.1 Métricas para Comparação

Em nosso estudo usamos como métrica principal para a comparação dos algoritmos o número de *checkpoints* forçados induzidos na aplicação. Esta medida permite avaliar o impacto, em termos do custo adicional associado às tarefas de construir e armazenar os *checkpoints* locais, do uso dos vários algoritmos quase-síncronos para *checkpointing*.

A outra métrica adotada foi o tamanho da informação de controle empregada pelos algoritmos. Por sua vez, esta medida permite estimar comparativamente o impacto, agora em termos da carga sobre a rede de comunicação, do uso dos vários algoritmos de *checkpointing*.

Estas duas métricas permitem ter uma idéia razoável do desempenho dos algoritmos sendo estudados. Adicionalmente, devemos escolher em que cenários os algoritmos devem ser exercitados para que os valores obtidos para cada uma destas métricas possam ser traduzidos em observações coerentes sobre o comportamento destes algoritmos.

Em relação ao tamanho da informação de controle a escolha de um cenário é desnecessária, pois todos os algoritmos estudados mantêm o tamanho da estrutura de dados que é propagada junto às mensagens da aplicação constante durante toda a execução. A variação ocorre apenas em função do número de processos no sistema e, supondo que para um dado algoritmo A o tamanho da informação de controle para uma execução de um

sistema com n processos é dado por $f_A(n)$, então temos que o custo total deste algoritmo em carga da rede é:

$$m \cdot f_A(n),$$

onde m é o número de mensagens trocadas durante toda a execução. Logo, no contexto desta métrica o estudo se resume a comparar as funções f dos algoritmos. Usamos então a ferramenta mais usual para este tipo de comparação, a análise assintótica [8], empregando a notação O sem nos preocupar em especificar exatamente a função f.

Em relação ao número de *checkpoints* forçados observamos que esta medida varia bastante de acordo com a seqüência de mensagens e *checkpoints* básicos, logo a escolha dos cenários deve ser feita de maneira bem cuidadosa. A seqüência de mensagens e *checkpoints* básicos pode variar em função de vários aspectos, como a proporção entre mensagens e *checkpoints* básicos, o padrão de troca de mensagens entre os processos e a topologia da rede. Neste estudo nós concentramos os esforços em três destes critérios: a escala do sistema, o tamanho dos intervalos de *checkpoints* e a diferença na freqüência dos *checkpoints* básicos entre os processos da aplicação.

Cenários onde varia o tamanho em processos do sistema são úteis para avaliar a escalabilidade dos algoritmos estudados. Estamos interessados no impacto do número de processos sobre a ocorrência de *checkpoints* forçados, principalmente para aqueles algoritmos que fazem uso da informação de causalidade na forma de vetores e matrizes de relógios.

O tamanho dos intervalos de *checkpoints* é medido em termos do número de eventos de comunicação que existem entre dois *checkpoints*. Este tamanho afeta o comportamento dos algoritmos quase-síncronos de duas formas. Primeiro, quanto menor o tamanho destes intervalos, maior será a freqüência com a qual um processo tira *checkpoints* básicos, logo tende a ser maior o número de *checkpoints* forçados de forma a garantir a utilidade destes *checkpoints*. Por outro lado, quanto maiores estes intervalos, também maior fica a probabilidade de que a informação causal de um *checkpoint* básico alcance um número maior de processos antes do próximo *checkpoint* e aumenta também a possibilidade de formação de caminhos-z e ciclos-z. Estudamos então cenários onde o tamanho destes intervalos varia de forma a avaliar seu impacto nos algoritmos estudados.

Um outro fator envolvendo o tamanho dos intervalos é a diferença da freqüência de *checkpoints* básicos entre os processos da aplicação. Um ou mais processos mais ativos que os demais, mas que se comuniquem continuamente com os processos menos ativos da aplicação, geram um número maior de *checkpoints* básicos e propagam informação sobre os mesmos podendo afetar o comportamento dos algoritmos quase-síncronos. Para avaliar este impacto, comparamos os algoritmos em cenários onde a diferença ou assimetria entre os processos varia.

Elaboramos situações de teste combinando os critérios de escala e tamanho de inter-

valos de *checkpoints* com o critério de assimetria e adotamos cinco cenários de teste que permitem uma avaliação abrangente dos algoritmos, no contexto dos critérios descritos acima e usando como métrica o número médio de *checkpoints* forçados por processo da aplicação. Estes cinco cenários são:

SP: Processos simétricos com dados colhidos em função do número de processos no sistema. Todos os processos possuem intervalos com 40 eventos de comunicação em média e as medidas foram feitas para aplicações com 2 a 16 processos.

SI: Processos simétricos com dados colhidos em função do tamanho médio dos intervalos de *checkpoints*. O sistema neste cenário é composto por 6 processos e as medidas foram feitas para intervalos com uma média de 4 a 118 eventos de comunicação.

AV: Processos assimétricos com dados colhidos em função da diferença de simetria. O sistema é composto por 6 processos e as medidas foram feitas variando-se o tamanho dos intervalos de *checkpoints* de um dos processos da aplicação. Os dados estão em função da diferença do tamanho médio dos intervalos deste único processo em relação aos intervalos com na média 44 eventos de comunicação dos outros processos, diferença esta que varia de 2 a 40 eventos de comunicação.

AP: Processos assimétricos com dados colhidos em função do número de processos no sistema. Os processos do sistema possuem na média intervalos com 44 eventos de comunicação exceto um processo com apenas 14 eventos de comunicação na média em cada intervalo de *checkpoints*. As medidas foram feitas para aplicações com 2 a 16 processos.

AI: Processos assimétricos com dados colhidos em função do tamanho médio dos intervalos de *checkpoints*. O sistema é composto por 6 processos, dos quais apenas um possui intervalos na média com 30 eventos de comunicação a menos que os demais. As medida foram feitas para intervalos deste processo mais rápido com uma média de 4 a 118 eventos de comunicação.

Todos os cenário descritos acima possuem alguns parâmetros comuns de configuração do sistema:

- A rede de comunicação em todos os cenários é completa, isto é, os processos podem mandar mensagens diretamente para qualquer um dos outros processos;
- Os canais de comunicação não perdem, corrompem ou alteram a ordem das mensagens enviadas;

- O evento de recepção possui prioridade ligeiramente maior que o evento de envio. Desta forma modelamos um sistema onde a latência das mensagens é muito pequena em relação ao tempo de duração dos intervalos.

Com estes cenários realizamos os testes coletando pontos experimentais ao longo do intervalo descrito para cada cenário. No caso dos cenários SP e AP usamos 14 pontos e nos cenários SI, AV e AI usamos 20 pontos experimentais. Cada um destes pontos experimentais foi obtido pela média da execução de todos os algoritmos para 10 padrões de mensagens e *checkpoints* básicos gerados aleatoriamente com no total uma média de 12000 eventos de comunicação por processo ao longo da execução. Para todos os pontos experimentais o desvio padrão das 10 execuções foi sempre menor que 4% da média, oscilando em torno dos 1% da média.

Os dados obtidos nesses experimentos estão listados no Apêndice A. As tabelas deste Apêndice mostram para cada um dos pontos experimentais a média de *checkpoints* forçados por processo ao longo das execuções e o desvio padrão como um percentual desta média. Ao longo deste Capítulo apresentaremos estes dados usando gráficos, onde os pontos experimentais estão ligados por retas, sem qualquer tipo de interpolação.

5.2 Análise Comparativa

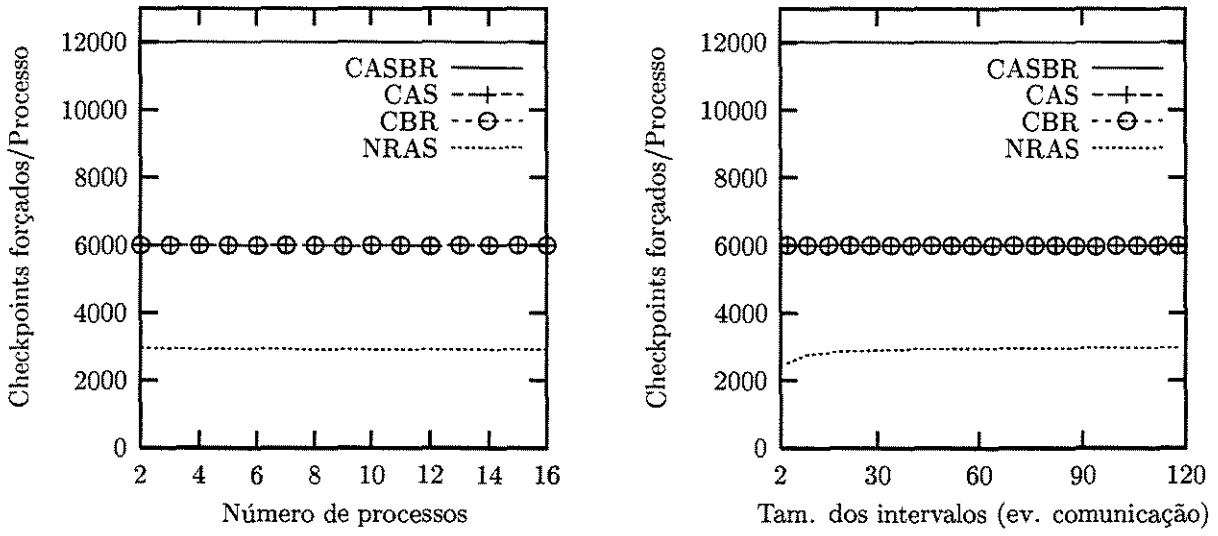
Faremos agora uma análise do desempenho dos vários algoritmos estudados de acordo com as métricas propostas na Seção anterior. A análise está dividida em duas partes principais, os algoritmos que garantem padrão ZPF e os que garantem padrão ZCF. Dentro destas classes agrupamos os algoritmos de acordo com as suas características de desempenho, apontando o pontos principais do comportamento esperado de cada grupo.

5.2.1 Algoritmos ZPF

Começamos o estudo dos algoritmos ZPF pelo conjunto de algoritmos mais elementares desta classe, aqueles que não fazem uso de informação de controle: CASBR, CAS, CBR e NRAS. As Figuras 5.1 e 5.2 mostram os gráficos obtidos para estes quatro algoritmos.

Estes algoritmos não usam informação de controle, logo eles não geram carga extra sobre a rede de comunicação. Por outro lado, estes algoritmos induzem um número muito alto de *checkpoints* forçados, número este função direta da quantidade de mensagens trocadas pelo sistema.

Observando as Figuras 5.1 e 5.2 é possível notar que, apesar de alto, o número de *checkpoints* forçados por estes algoritmos é bem previsível e independe das várias situações representadas pelos gráficos. No caso do algoritmo CASBR o número de *checkpoints* forçados será igual ao número de eventos de comunicação e no caso dos algoritmos CAS



(a) Simétrico, var. n° de processos (SP)

(b) Simétrico, var. tam. dos intervalos (SI)

Figura 5.1: Algoritmos ZPF somente baseados em modelo - 1

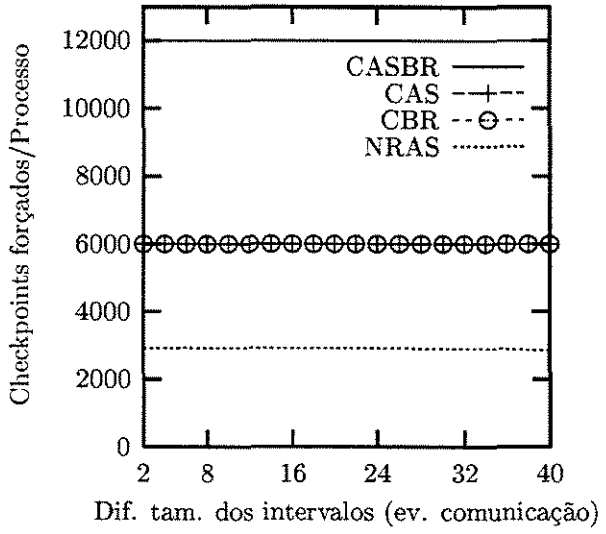
e CBR este número será igual a metade do número de eventos de comunicação. Nos cenários que estudamos, os eventos de recepção de mensagens e os eventos de envio de mensagens possuem a mesma probabilidade de ocorrência, logo o algoritmo NRAS deve induzir um número de *checkpoints* forçados igual a um quarto do número de eventos de comunicação, supondo que os eventos de *checkpoints* básicos são bem menos freqüentes.

Mesmo com essa previsibilidade, estes algoritmos só aparentam ser indicados para aquelas aplicações que exigem as garantias de padrão de *checkpoints* específicas fornecidas pelos mesmos [10].

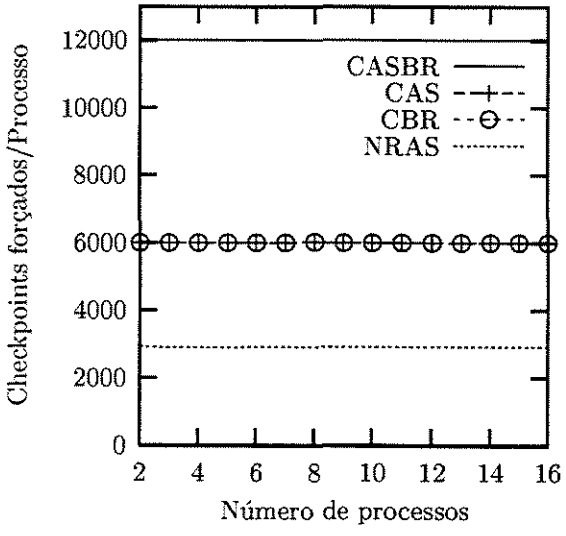
Algoritmos mais sofisticados que garantem o padrão ZPF empregam algum tipo de relógio lógico para auxiliar na decisão de quando forçar um *checkpoint*. Dois dos algoritmos mais simples que usam relógios são o FDI e o FDAS. Os gráficos do desempenho destes algoritmos estão nas Figuras 5.3 e 5.4.

Tanto o algoritmo FDI quanto o FDAS propagam um vetor de relógios de tamanho $O(n)$ como informação de controle, impondo uma carga consideravelmente maior sobre a rede de comunicação em relação aos algoritmos que não fazem uso de relógio.

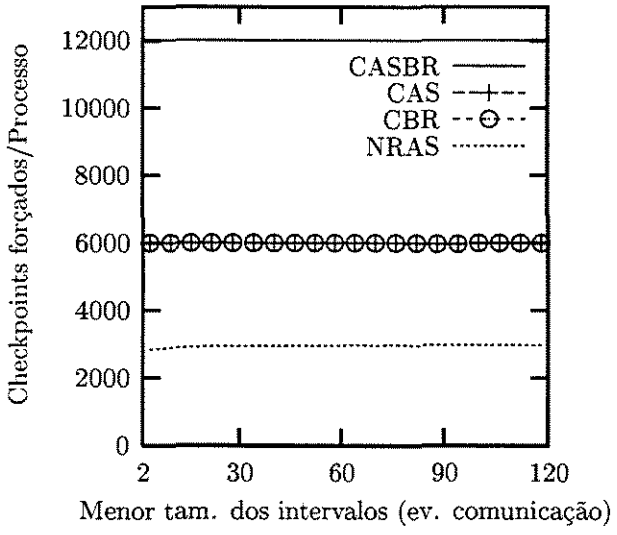
Observando os dados referentes aos cenário SP e AP (Figuras 5.3(a) e 5.4(b)) podemos perceber que ambos os algoritmos possibilitam alguma economia de *checkpoints* forçados em relação aos algoritmos sem relógio, porém eles sofrem muito o impacto do aumento da escala do sistema. Mesmo assim, a escala faz no pior caso com que os algoritmos se



(a) Assimétrico, var. simetria (AV)

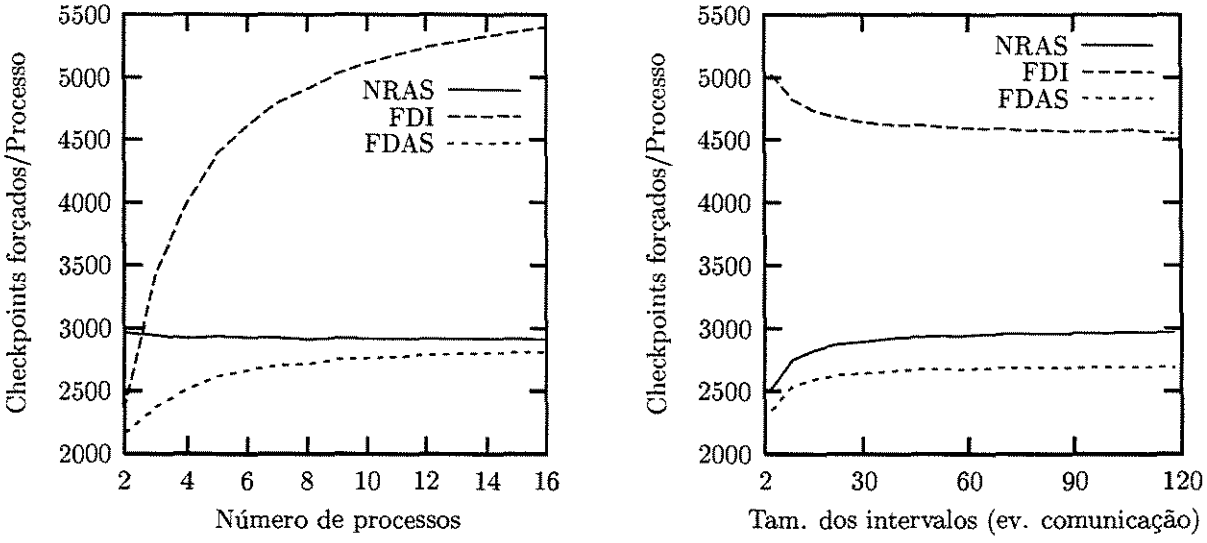


(b) Assimétrico, var. nº de processos (AP)



(c) Assimétrico, var. tam. dos intervalos (AI)

Figura 5.2: Algoritmos ZPF somente baseados em modelo - 2



(a) Simétrico, var. n° de processos (SP)

(b) Simétrico, var. tam. dos intervalos (SI)

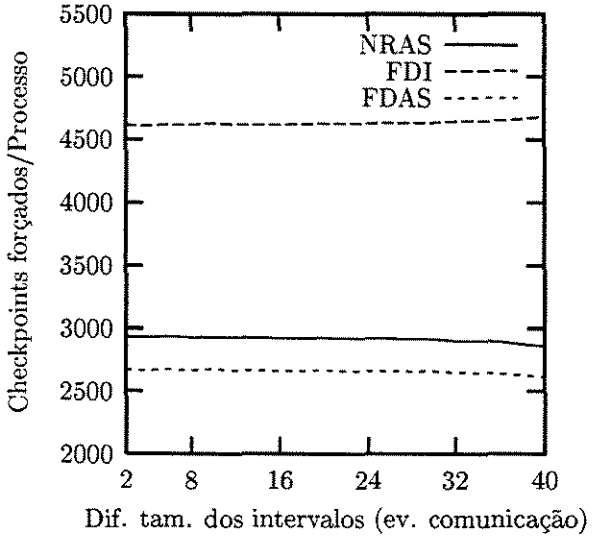
Figura 5.3: Algoritmos ZPF que usam relógios simples - 1

aproximem, mas continuem melhores, que os algoritmos sem relógio. O FDI se aproxima do CBR e o FDAS se aproxima do NRAS.

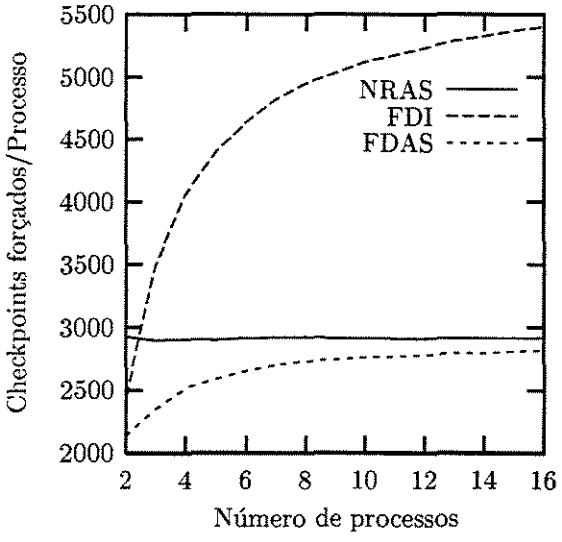
Provavelmente a escala afeta de forma tão negativa estes algoritmos em nossos experimentos porque eles fazem uso de vetores de relógios e nossos cenários permitem troca de mensagens entre quaisquer dois processos do sistema. Desta forma, na medida que a escala do sistema aumenta, também aumenta a probabilidade de uma mensagem trazer novas informações causais sobre algum dos outros processos do sistema. O algoritmo FDI vai tender a forçar um *checkpoint* a cada recebimento de mensagem e o FDAS vai tender a forçar um *checkpoint* a cada recebimento de mensagem precedido por envio de mensagem.

Pelos dados dos cenários SI, AI e AV (Figuras 5.3(b), 5.4(c) e 5.4(a)) percebemos que estes algoritmos não sofrem influência significativa dos tamanhos dos intervalos ou da simetria. Algum efeito é observado no cenário SI para intervalos curtos. Este efeito é favorável ao algoritmo FDAS devido a maior ocorrência de *checkpoints* básicos, a semelhança do NRAS, e desfavorável ao FDI pois a ocorrência de *checkpoints* básicos incrementa os vetores bem rapidamente.

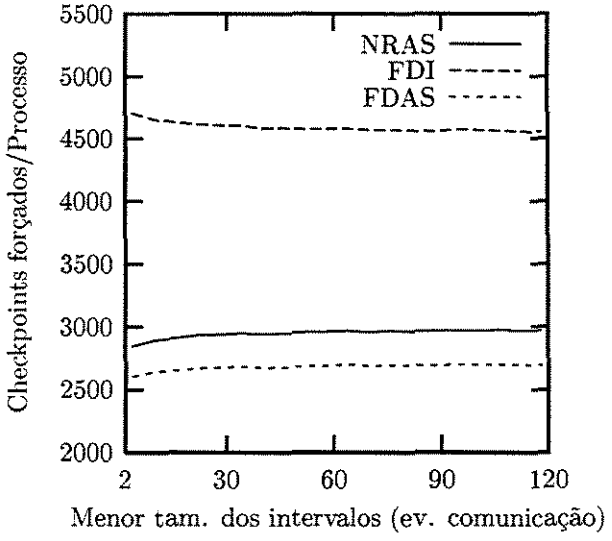
Claramente o algoritmo FDAS apresenta o uso mais efetivo do vetor de relógio, apresentando um ganho em termos do número de *checkpoints* forçados considerável para sistemas de 2 a 6 processos em relação ao NRAS. Ambos os algoritmos FDI e FDAS geram um padrão ZPF com as mesmas propriedades e propagam informação de controle $O(n)$,



(a) Assimétrico, var. simetria (AV)



(b) Assimétrico, var. n° de processos (AP)

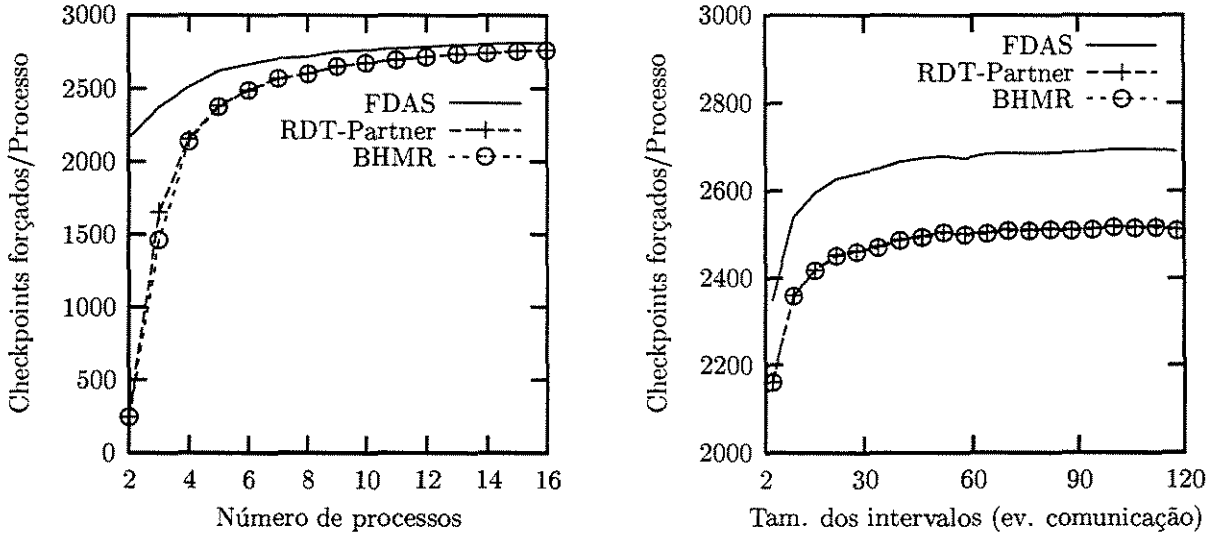


(c) Assimétrico, var. tam. dos intervalos (AI)

Figura 5.4: Algoritmos ZPF que usam relógios simples - 2

logo o uso do algoritmo FDI não parece ser razoável devido a velocidade com a qual seu desempenho degrada em função da escala do sistema. Mesmo o algoritmo FDAS, para sistemas com mais que 8 processos, já não apresenta ganhos que justifiquem o preço pago em termos de carga na rede, exceto nos casos onde o custo de um *checkpoint* local é muito maior que o custo da banda passante na rede.

Os algoritmos RDT-Partner e BHMR apresentam soluções mais sofisticadas para o problema da geração de um padrão ZPF minimizando o número de *checkpoints* forçados. Os dados colhidos sobre estes algoritmos estão nas Figuras 5.5 e 5.6.



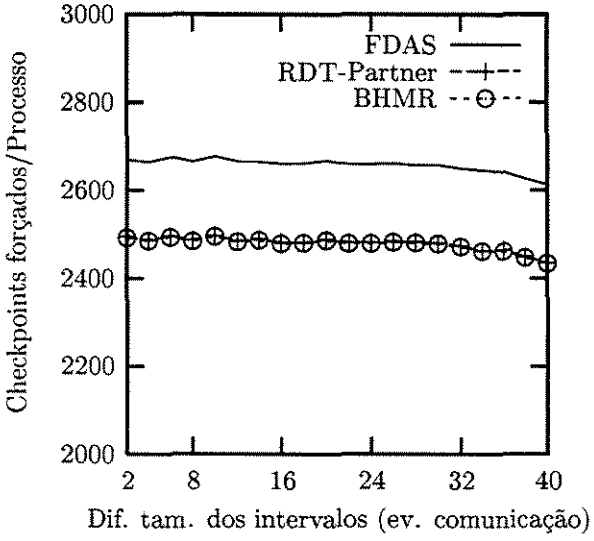
(a) Simétrico, var. n° de processos (SP)

(b) Simétrico, var. tam. dos intervalos (SI)

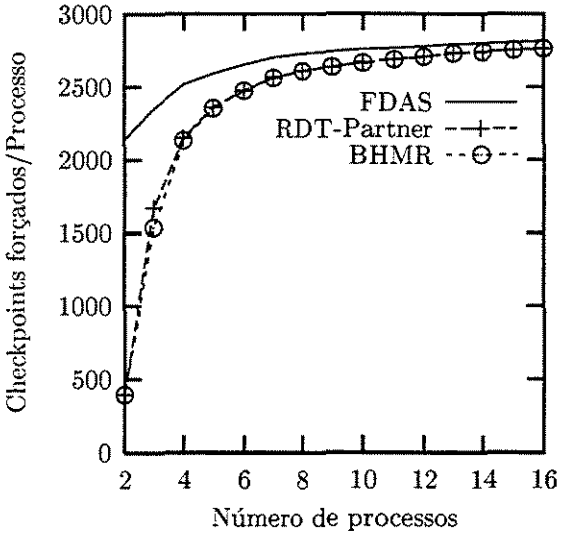
Figura 5.5: Algoritmos ZPF que usam relógios elaborados - 1

O algoritmo RDT-Partner propaga como informação de controle um booleano e um vetor ($O(n)$) enquanto o algoritmo BHMR propaga dois vetores e uma matriz ($O(n^2)$). O custo em termos de carga da rede do algoritmo RDT-Partner se equipara então ao custo dos algoritmos mais simples FDI e FDAS.

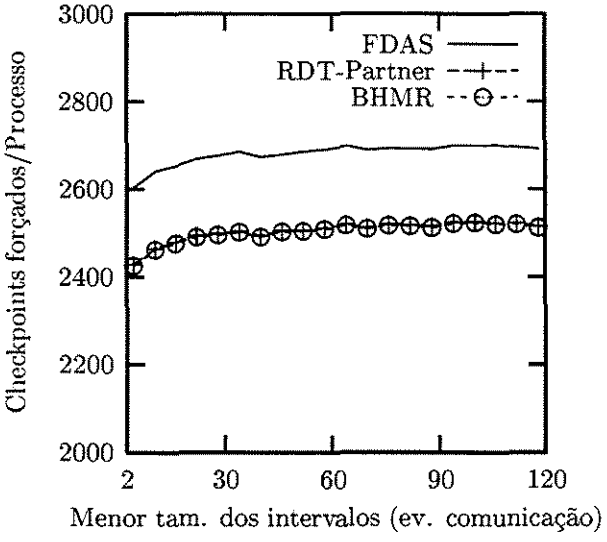
Os dados referentes aos cenários SP e AP (Figuras 5.5(a) e 5.6(b)) mostram que estes algoritmos conseguem ganhos consideráveis em relação ao FDAS para sistemas com poucos processos mas sofrem sensivelmente com o aumento da escala do sistema. Novamente este efeito ocorre devido ao relógio empregado por estes algoritmos e ao fato dos nossos experimentos considerarem um modelo de comunicação uniforme, enquanto estes algoritmos procuram explorar configurações bem particulares do padrão de comunicação do sistema.



(a) Assimétrico, var. simetria (AV)



(b) Assimétrico, var. n° de processos (AP)



(c) Assimétrico, var. tam. dos intervalos (AI)

Figura 5.6: Algoritmos ZPF que usam relógios elaborados - 2

De forma semelhante aos outros algoritmos ZPF estudados, o RDT-Partner e o BHMR não sofrem influência do tamanho dos intervalos e da assimetria do sistema, como pode ser observado pelos gráficos dos cenários SI, AI e AV (Figuras 5.5(b), 5.6(c) e 5.6(a)). Observamos apenas um pequeno ganho para intervalos muito curtos, semelhantes aqueles observados para os algoritmo NRAS e FDAS.

Ambos os algoritmos RDT-Partner e BHMR apresentam ganhos no número de *checkpoints* forçados, mas sofrem penalidades a medida que a escala do sistema cresce. Observamos também que em nosso ambiente experimental o algoritmo BHMR não apresentou ganhos consideráveis neste número em relação ao RDT-Partner apesar de usar uma matriz de relógios, elevando a informação de controle de $O(n)$ a $O(n^2)$. Este dado indica que provavelmente a ocorrência das estruturas exploradas apenas pelo BHMR é bem menos freqüente do que a ocorrência daquelas exploradas pelo RDT-Partner. Com base nesta observação, nos parece mais apropriado o uso do algoritmo RDT-Partner para aplicações com poucos processos ou onde ocorre comunicação mais localizada e do algoritmo NRAS para aplicações de maior tamanho com comunicação mais distribuída entre todos os processos ou onde o custo dos *checkpoints* locais é muito menor que o custo da comunicação. O algoritmo RDT-Partner supera o FDAS sem aumentar o tamanho da informação de controle e sem impor uma implementação mais complicada, enquanto o NRAS apresenta custo zero sobre a rede e implementação trivial.

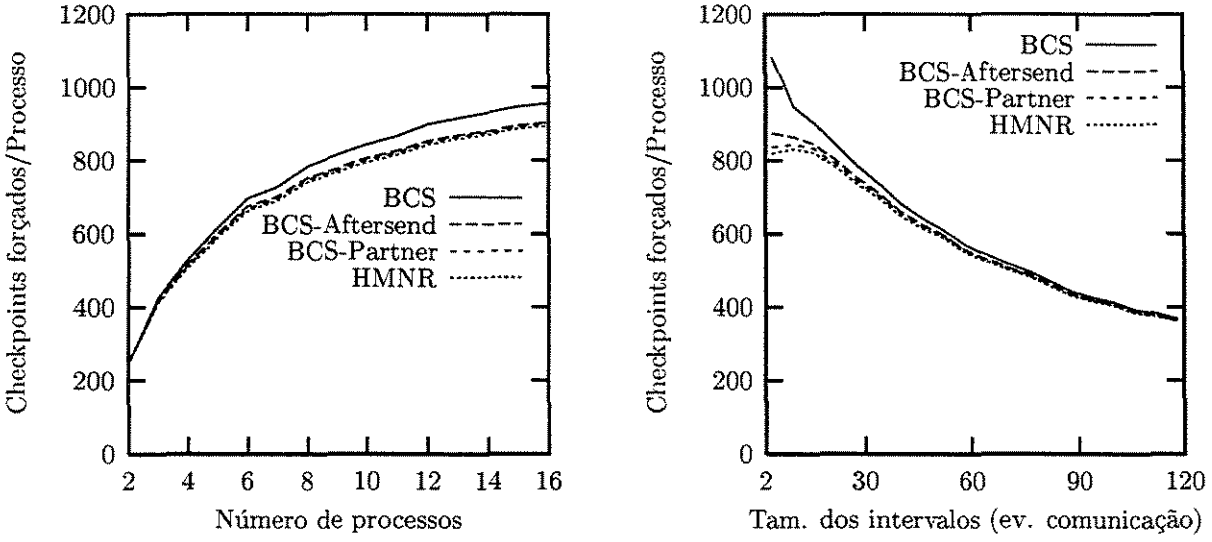
5.2.2 Algoritmos ZCF

O algoritmo BCS é o principal representante da classe de algoritmos que geram padrões ZCF e estudamos os outros algoritmos desta classe de acordo com a otimização em relação ao BCS que eles representam. O primeiro grupo corresponde aos algoritmos que procuram incrementar o relógio sem ter que tirar um *checkpoint* forçado. Neste grupo colocamos, além do BCS, os algoritmos BCS-Aftersend, BCS-Partner e HMNR. Os dados obtidos para este grupo de algoritmos estão nas Figuras 5.7 e 5.8.

Em relação a informação de controle propagada junto às mensagens da aplicação os algoritmos BCS, BCS-Aftersend e BCS-Partner propagam apenas um número constante de inteiros, com complexidade de tamanho $O(1)$. O algoritmo HMNR por sua vez propaga três vetores além do índice, tendo complexidade $O(n)$.

De forma geral, observamos em todos os cenários que o desempenho dos algoritmos reflete o fato deles poderem ser vistos como otimizações sucessivas do BCS. Em todos os gráficos os algoritmos estão, em ordem decrescente de número de *checkpoints* forçados, nesta ordem: BCS, BCS-Aftersend, BCS-Partner e HMNR.

Observando os cenários SP e AP (Figuras 5.7(a) e 5.8(b)), percebemos que estes algoritmos, mesmo não fazendo uso de vetores de relógios são também sensíveis à escala



(a) Simétrico, var. n° de processos (SP)

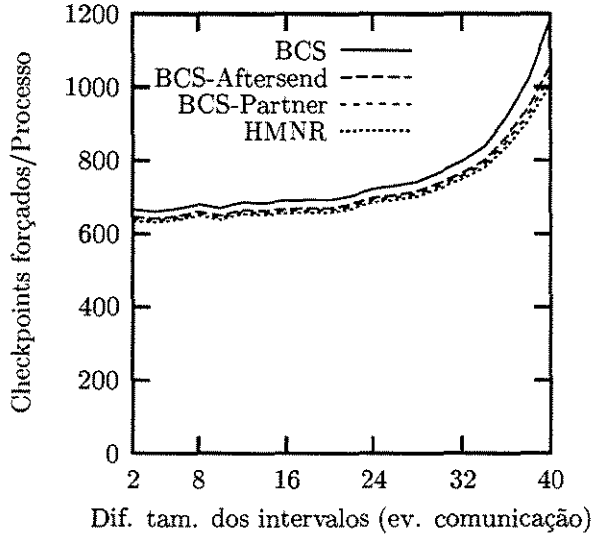
(b) Simétrico, var. tam. dos intervalos (SI)

Figura 5.7: Algoritmos ZCF que evitam *checkpoints* desnecessários - 1

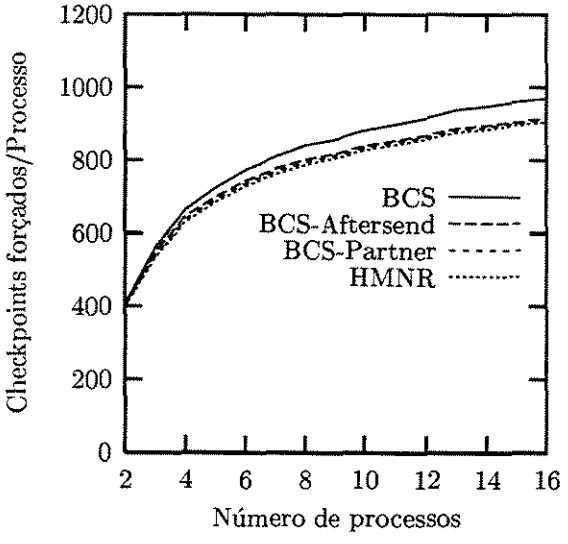
do sistema, não de forma tão intensa quanto os algoritmos ZPF mais ainda de forma considerável. Novamente temos que este efeito é causado pelo maior número de processos com informações causais potencialmente novas, levando os algoritmos a induzir *checkpoints* forçados. É possível também perceber logo no cenário AP que a assimetria afeta o comportamento destes algoritmos, pois o desempenho deles não é tão bom para poucos processos como no cenário SP.

Nos cenários SI e AI (Figuras 5.7(b) e 5.8(c)) observamos um dado importantíssimo sobre o algoritmo BCS e seus derivados, o desempenho do algoritmos é afetado intensamente pelo tamanho dos intervalos. Podemos perceber pelo gráfico que quanto maiores os intervalos, menor é o número de *checkpoints* forçados pelos algoritmos. Esse comportamento é observado em todos os algoritmos e todos são afetados positivamente da mesma maneira. Este efeito é conseqüência direta do padrão de *checkpoints* que estes algoritmos produzem, para cada *checkpoint* básico só são induzidos *checkpoints* forçados para garantir a inexistência de ciclos-z, quando este objetivo é alcançado só serão forçados novos *checkpoints* após um novo *checkpoint* básico. Intervalos maiores diminuem a ocorrência destes *checkpoints* ao longo da execução da aplicação, reduzindo o número de *checkpoints* forçados.

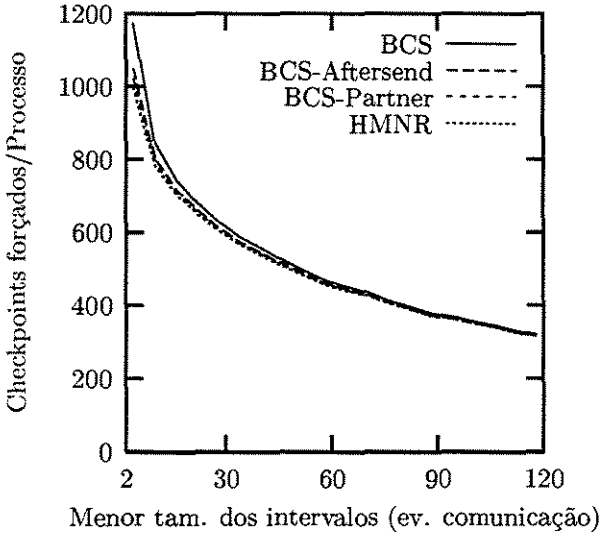
O cenário AV (Figura 5.8(a)) mostra outra propriedade destes algoritmos, eles também possuem o desempenho sensível a diferença de simetria entre os processos. Essa é uma



(a) Assimétrico, var. simetria (AV)



(b) Assimétrico, var. nº de processos (AP)



(c) Assimétrico, var. tam. dos intervalos (AI)

Figura 5.8: Algoritmos ZCF que evitam *checkpoints* desnecessários - 2

outra característica do BCS que é observada com a mesma intensidade nos outros algoritmos. A diferença de assimetria se reflete em um processo tirando *checkpoints* básicos em um ritmo maior que o do resto do sistema, induzindo os outros processos a tirarem *checkpoints* forçados para garantirem a utilidade de seus *checkpoints*. No cenário AV temos a combinação de dois fatores: na medida que um dos processos acelera, seus intervalos tendem a ficar menores. Este fato aumenta consideravelmente o número de *checkpoints* forçados no final do intervalo analisado, gerando a elevação final abrupta da curva.

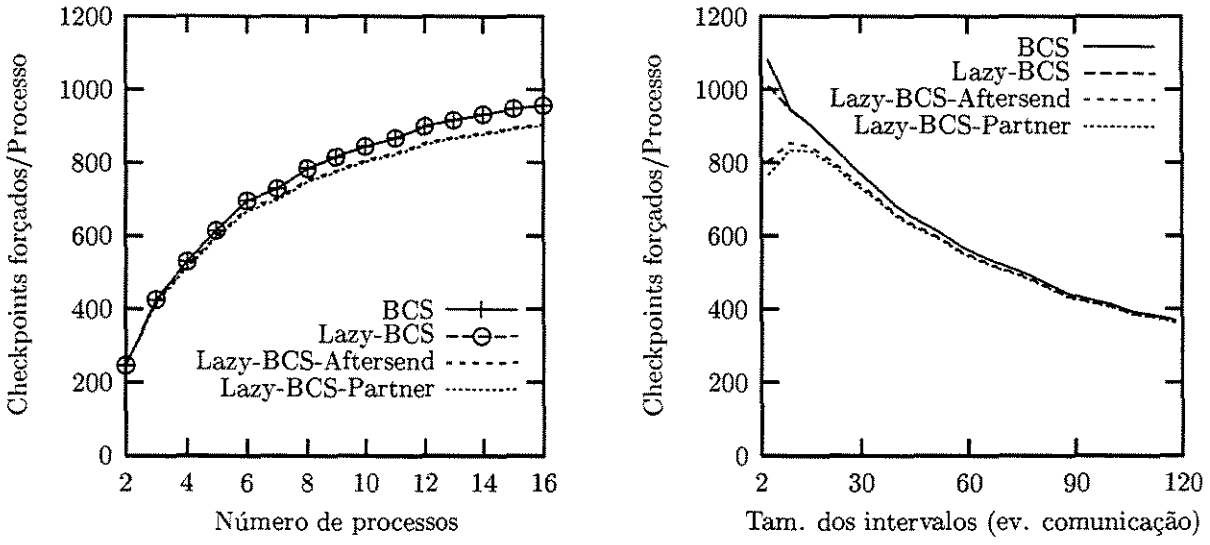
Em relação ao desempenho destes algoritmos, observamos que todos forçam um número de *checkpoints* muito semelhante. O algoritmos BCS-Aftersend, BCS-Partner e HMNR só apresentam ganhos significativos frente ao BCS em sistemas com muitos processos ou com intervalos de *checkpoints* bastante curtos. Estes três algoritmos apresentam também desempenho muito semelhante entre si, o que torna o HMNR pouco atraente, devido a sua informação de controle com tamanho $O(n)$. Mesmo o BCS-Partner só se justifica para aplicações onde é muito presente a estrutura de comunicação formada por pedido/resposta.

Os algoritmos do próximo grupo tentam uma outra abordagem para melhorar o BCS. Os algoritmos Lazy-BCS, as combinações Lazy-BCS-Aftersend e Lazy-BCS-Partner e o algoritmo BQF procuram tirar *checkpoints* básicos sem incrementar os índices. Os gráficos destes algoritmos menos o BQF estão nas Figuras 5.9 e 5.10, o algoritmo BQF tem seus gráficos nas Figuras 5.11 e 5.12.

A informação de controle propagada pelos algoritmos Lazy-BCS, Lazy-BCS-Aftersend e Lazy-BCS-Partner tem tamanho $O(1)$ enquanto o algoritmo BQF usa informação de controle de tamanho $O(n)$. Novamente temos que estes algoritmos se comportam como otimizações incrementais, mantendo nos gráficos sempre a ordem: Lazy-BCS, Lazy-BCS-Aftersend e Lazy-BCS-Partner, sendo que o BQF tem o desempenho muito parecido com o algoritmo Lazy-BCS-Partner.

No cenário SP e AP (Figuras 5.9(a) e 5.10(b)) observamos que estes algoritmos têm o seu desempenho afetado pela escala da mesma forma que os algoritmos ZCF do grupo anterior. É possível perceber também que o algoritmo Lazy-BCS apresenta um ganho muito pequeno em relação ao BCS, ganho esse apenas perceptível no cenário AP. Desta forma, o desempenho destes algoritmos nestes cenários não se altera significativamente em função da otimização do algoritmo Lazy-BCS empregada em todos eles.

Observamos nos cenários SI e AI (Figuras 5.9(b) e 5.10(c)) que como esperado estes algoritmos, por serem derivados do BCS, também melhoram o seu desempenho na medida que o tamanho dos intervalos aumenta. Este ganho no entanto faz com que as diferenças entre os algoritmos individuais quase que desapareça e notamos a presença da otimização do Lazy-BCS apenas quando os intervalos estão bem curtos. Intervalos curtos aumentam a probabilidade de que um processo possa tirar um *checkpoint* sem incrementar o seu índice,



(a) Simétrico, var. nº de processos (SP)

(b) Simétrico, var. tam. dos intervalos (SI)

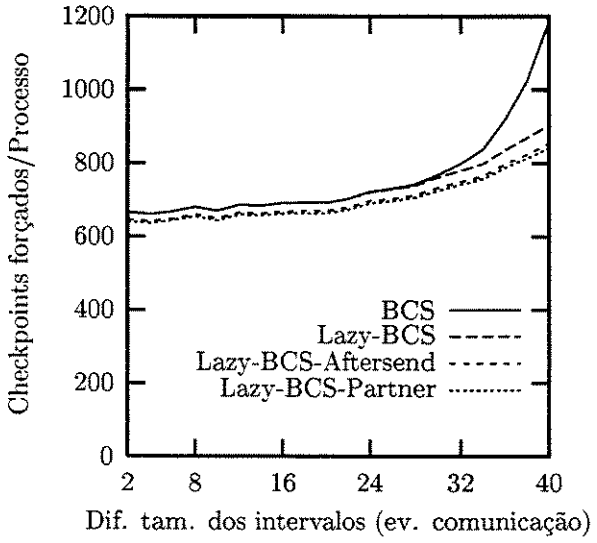
Figura 5.9: Algoritmos ZCF que evitam incrementar os índices - 1

sendo uma das situações onde as otimizações derivadas de Lazy-BCS exibem maiores ganhos.

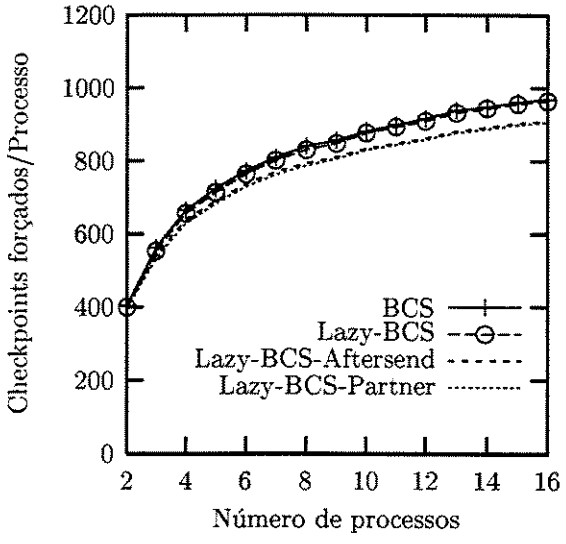
É no cenário AV (Figura 5.10(a)) que a otimização usada no algoritmo Lazy-BCS tem o melhor desempenho relativo. Neste cenário ocorre a combinação de um processo tirando *checkpoints* básicos em uma freqüência maior que a do resto do sistema, em intervalos cada vez menores. Temos então uma situação onde o sistema como um todo vai se beneficiar caso este processo não incremente seus índices, aliado aos intervalos curtos que aumentam a probabilidade de que isto ocorra.

Temos então que o uso da otimização Lazy-BCS ajuda a evitar situações onde o algoritmo BCS apresenta desempenho prejudicado por assimetria dos processos do sistema. Porém este algoritmo isolado perde a oportunidade de explorar outras situações que poderiam evitar *checkpoints* forçados. Podemos fugir desta limitação empregado a combinação deste algoritmo com os algoritmos BCS-Aftersend e BCS-Partner, que apresentam um desempenho que alia os pontos fortes das duas abordagens, com uma implementação simples e informação de controle de tamanho $O(1)$.

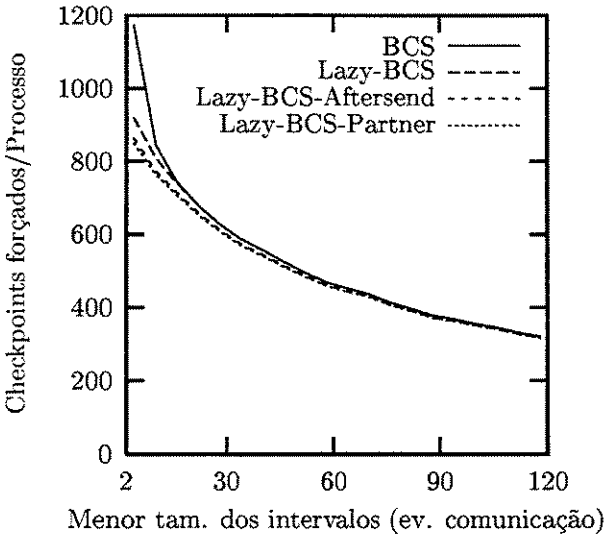
A estratégia de adiamento do incremento dos índices empregada pelo algoritmo BQF tem um desempenho muito semelhante aquele obtido pelo algoritmo Lazy-BCS-Partner em nossos cenários de testes (Figuras 5.11 e 5.12). Este fato indica que não vale a pena pagar o preço da informação de controle $O(n)$ deste algoritmo.



(a) Assimétrico, var. simetria (AV)



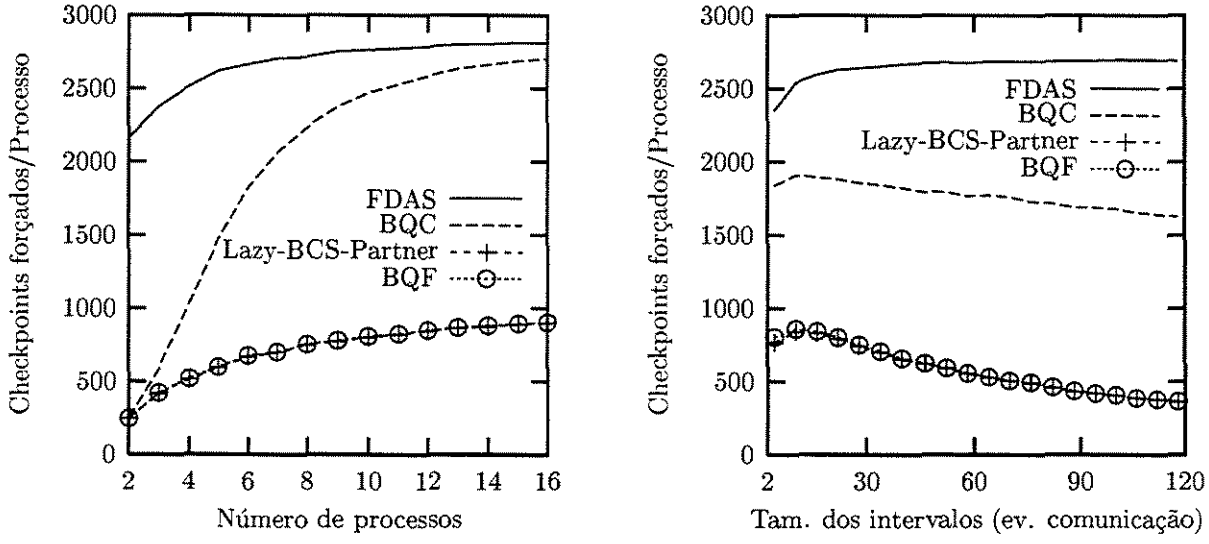
(b) Assimétrico, var. n° de processos (AP)



(c) Assimétrico, var. tam. dos intervalos (AI)

Figura 5.10: Algoritmos ZCF que evitam incrementar os índices - 2

Uma abordagem bem diferente daquela empregada pelas variações do BCS é usada no algoritmo BQC. Os dados obtidos para este algoritmo estão nas Figuras 5.11 e 5.12.



(a) Simétrico, var. n° de processos (SP)

(b) Simétrico, var. tam. dos intervalos (SI)

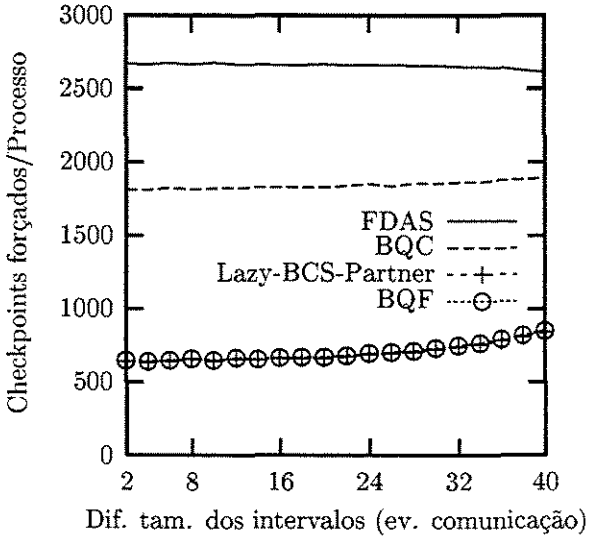
Figura 5.11: O algoritmo BQC - 1

O algoritmo BQC usa informações de controle de tamanho $O(n^2)$ e é muito sensível a variação de escala do sistema. Observamos isto nos cenários SP e AP (Figuras 5.11(a) e 5.12(b)), onde também é possível perceber que a medida que o tamanho do sistema cresce o desempenho deste algoritmo se aproxima do desempenho dos algoritmos FDAS e NRAS.

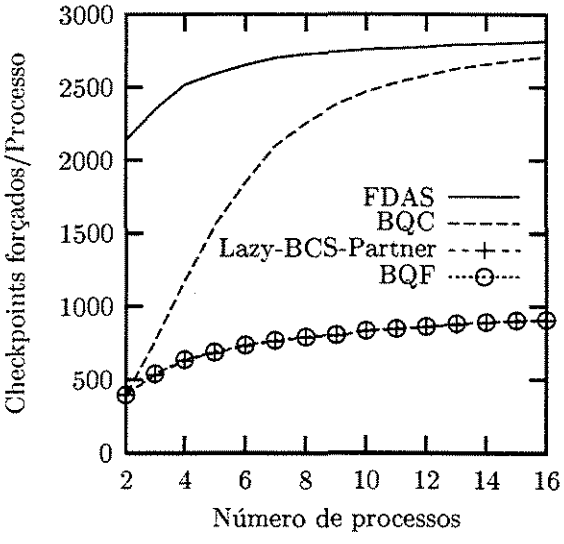
Por outro lado, nos cenários SI, AI e AV (Figuras 5.11(b), 5.12(c) e 5.12(a)) observamos que este algoritmo é menos sensível ao tamanho dos intervalos de *checkpoints* e a assimetria do sistema. Ao considerarmos que o número de *checkpoints* forçados é bastante alto para o padrão gerado e que a informação de controle tem tamanho $O(n^2)$, este algoritmo não possui nenhum atrativo para seu uso na prática.

5.2.3 O Custo de um Padrão ZPF

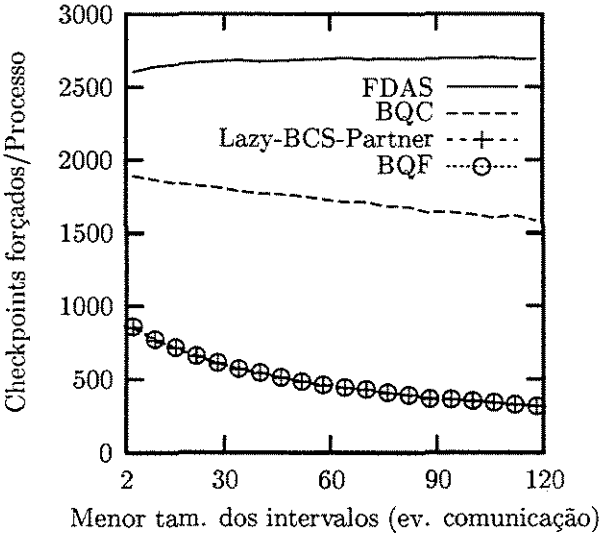
Na análise dos grupos de algoritmos das Seções anteriores percebemos uma grande diferença entre o número de *checkpoints* forçados induzidos pelos algoritmos ZPF e pelos algoritmos ZCF. As Figuras 5.11 e 5.12 ilustram bem esta diferença, comparando alguns algoritmos de cada classe.



(a) Assimétrico, var. simetria (AV)



(b) Assimétrico, var. n° de processos (AP)



(c) Assimétrico, var. tam. dos intervalos (AI)

Figura 5.12: O algoritmo BQC - 2

Os algoritmos ZPF conhecidos, ou não exigem a propagação de um relógio ou exigem a propagação de um relógio com tamanho $O(n)$, enquanto os algoritmos ZCF propagam em sua maioria informação de controle de tamanho constante. Mesmo usando um relógio mais complexo, os algoritmos ZPF tiram um número consideravelmente maior de *checkpoints* para garantir este padrão.

O número de *checkpoints* forçados pelos algoritmos ZPF não é influenciado significativamente pelo tamanho dos intervalos de *checkpoints*. Por um lado, este é uma fato positivo, pois estes algoritmos apresentam um comportamento mais previsível em situações onde o tamanho dos intervalos pode variar de forma inesperada. Porém, esta característica destes algoritmos implica que eles, para uma dada configuração do sistema, tiram *checkpoints* forçados em uma taxa constante independente da taxa de *checkpoints* básicos, isto é, independente do julgamento dos processos sobre quais estados são relevantes. Esta taxa mantém-se constante mesmo no caso extremo onde apenas são tirados os *checkpoints* básicos iniciais.

Os algoritmos ZCF por sua vez, conseguem forçar *checkpoints* apenas na medida em que os processos da aplicação tiram os seus *checkpoints* básicos, fazendo uso de informação de controle de tamanho constante. Por outro lado, a obtenção de um *checkpoint* global consistente neste padrão é um pouco mais complicada e existem aplicações que exigem as propriedades do padrão ZPF, limitando o uso exclusivo destes algoritmos em aplicações de *checkpointing*.

5.3 Conclusão

Neste trabalho apresentamos um estudo abrangente do desempenho dos algoritmos quase-síncronos para *checkpointing*. Obtivemos pela primeira vez dados comparativos em um ambiente uniforme sobre estes algoritmos. Fomos capazes de avaliar o impacto sobre o desempenho dos algoritmos de fatores como a escala do sistema, a freqüência de *checkpoints* básicos e a diferença na velocidade dos processos da aplicação. Obtivemos um grande conhecimento sobre as características destes algoritmos, sendo este estudo um valioso referencial para projetistas de sistemas em busca de algoritmos para *checkpointing* para as suas aplicações distribuídas.

De acordo com os dados obtidos em nossos experimentos, concluímos que:

- Em relação aos algoritmos que geram um padrão de *checkpoints* ZPF, é mais apropriado o uso do algoritmo NRAS para aplicações com um grande número de processos e com comunicação distribuída entre todos os processos ou onde o custo dos *checkpoints* locais é muito menor que o custo da comunicação, e do algoritmo RDT-Partner para aplicações de menor tamanho ou onde a comunicação é mais localizada.

Estes dois algoritmos aparentam fornecer um equilíbrio interessante entre o número de *checkpoints* forçados, o custo de comunicação envolvido e a complexidade de sua implementação.

- Em relação aos algoritmos que geram um padrão de *checkpoints* ZCF, temos que o algoritmo BCS já oferece um excelente balanço entre o número de *checkpoints* forçados e os custos de seu emprego. Porém este algoritmo apresenta deficiências em algumas situações, como por exemplo quando os intervalos de *checkpoints* possuem poucas mensagens ou quando os processos tiram *checkpoints* em ritmos distintos. Podemos fugir destas limitações empregado a combinação deste algoritmo com algumas otimizações. Notamos que as combinações mais simples, como o algoritmo Lazy-BCS-Aftersend, apresentam um desempenho que alia os pontos fortes das duas abordagens de otimização com uma implementação simples e informação de controle de tamanho constante.

É importante ressaltar que, na mesma medida que a simulação nos dá liberdade para estudar o funcionamento dos algoritmos em cenários criados sob o nosso controle, nossas análises refletem o comportamento dos algoritmos apenas sob o ambiente de simulação. Mesmo acreditando que os cenários por nós escolhidos representem fatores relevantes ao funcionamento destes algoritmos, eles deixam de lado outros fatores que também possuem a sua importância.

Em especial, não exploramos neste trabalho o impacto de diferentes topologias de rede e padrões de comunicação entre os processos, nos atendo a configurações onde os processos trocam mensagens de forma uniforme. Por exemplo, seria interessante observar o comportamento dos algoritmos quando os processos se comunicam em uma arquitetura cliente/servidor ou quando existem grupos sobrepostos de processos. A ferramenta de simulação fornece o suporte para construção deste tipo de configuração, mas decidimos concentrar nossos esforços no estudo de fatores relacionados com a proporção entre *checkpoints* básicos e mensagens.

Adicionalmente, identificamos alguns pontos onde a ferramenta de simulação poderia ser melhorada. Percebemos que o uso do Spin tornou o processo de simulação muito custoso em termos computacionais, pois esta ferramenta simula o sistema na granularidade dos comandos individuais e não dos eventos de nosso modelo de sistema distribuído. Possíveis soluções a este problema envolveriam a construção de modelos em Promela mais eficientes ou a eliminação do Spin da ferramenta de simulação.

Poderíamos também acrescentar à ferramenta de simulação a capacidade de exercitar os algoritmos usando o registro da execução de uma aplicação distribuída real, ou extrapolando esta idéia, um "gerador de carga" que poderia representar classes de aplicações distribuídas de forma mais precisa. Poderíamos então validar os resultados aqui obtidos

com aplicações reais, em primeiro lugar para avaliar se os cenários estudados são relevantes e em seguida para confirmar se os algoritmos se comportam de forma geral como observado em nosso cenários.

Apêndice A

Médias e Desvios Padrão

Neste Apêndice estão listados os dados obtidos em nossos experimentos. As tabelas mostram para cada um dos pontos experimentais a média de *checkpoints* forçados por processo ao longo das execuções e o desvio padrão como um percentual desta média. Estas tabelas estão organizadas por cenário e, dentro de cada tabela, as colunas representam os algoritmos e as linhas o parâmetro de prioridade ou escala usado no teste.

Para facilitar a leitura, repetimos neste Apêndice a descrição dos experimentos realizados.

Elaboramos situações de teste combinando os critérios de escala e tamanho de intervalos de *checkpoints* com o critério de assimetria e adotamos cinco cenários de teste que permitem uma avaliação abrangente dos algoritmos, no contexto dos critérios descritos acima e usando como métrica o número médio de *checkpoints* forçados por processo da aplicação. Estes cinco cenários são:

SP: Processos simétricos com dados colhidos em função do número de processos no sistema. Todos os processos possuem intervalos com 40 eventos de comunicação em média e as medidas foram feitas para aplicações com 2 a 16 processos.

SI: Processos simétricos com dados colhidos em função do tamanho médio dos intervalos de *checkpoints*. O sistema neste cenário é composto por 6 processos e as medidas foram feitas para intervalos com uma média de 4 a 118 eventos de comunicação.

AV: Processos assimétricos com dados colhidos em função da diferença de simetria. O sistema é composto por 6 processos e as medidas foram feitas variando-se o tamanho dos intervalos de *checkpoints* de um dos processos da aplicação. Os dados estão em função da diferença do tamanho médio dos intervalos deste único processo em relação aos intervalos com na média 44 eventos de comunicação dos outros processos, diferença esta que varia de 2 a 40 eventos de comunicação.

AP: Processos assimétricos com dados colhidos em função do número de processos no sistema. Os processos do sistema possuem na média intervalos com 44 eventos de comunicação exceto um processo com apenas 14 eventos de comunicação na média em cada intervalo de *checkpoints*. As medidas foram feitas para aplicações com 2 a 16 processos.

AI: Processos assimétricos com dados colhidos em função do tamanho médio dos intervalos de *checkpoints*. O sistema é composto por 6 processos, dos quais apenas um possui intervalos na média com 30 eventos de comunicação a menos que os demais. As medida foram feitas para intervalos deste processo mais rápido com uma média de 4 a 118 eventos de comunicação.

Todos os cenário descritos acima possuem alguns parâmetros comuns de configuração do sistema:

- A rede de comunicação em todos os cenários é completa, isto é, os processos podem mandar mensagens diretamente para qualquer um dos outros processos;
- Os canais de comunicação não perdem, corrompem ou alteram a ordem das mensagens enviadas;
- O evento de recepção possui prioridade ligeiramente maior que o evento de envio. Desta forma modelamos um sistema onde a latência das mensagens é muito pequena em relação ao tempo de duração dos intervalos.

Com estes cenários realizamos os testes coletando pontos experimentais ao longo do intervalo descrito para cada cenário. No caso dos cenários SP e AP usamos 14 pontos e nos cenários SI, AV e AI usamos 20 pontos experimentais. Cada um destes pontos experimentais foi obtido pela média da execução de todos os algoritmos para 10 padrões de mensagens e *checkpoints* básicos gerados aleatoriamente com no total uma média de 12000 eventos de comunicação por processo ao longo da execução. Para todos os pontos experimentais o desvio padrão das 10 execuções foi sempre menor que 4% da média, oscilando em torno dos 1% da média.

	CASBR		CAS		CBR		NRAS		FDI		FDAS	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
2	11999.4	0.005	6000.1	0.004	5999.3	0.010	2965.8	0.625	2343.8	0.576	2160.8	0.844
3	11999.9	0.002	6000.2	0.005	5999.7	0.004	2943.1	0.822	3437.1	0.618	2373.6	0.923
4	11999.6	0.003	6000.1	0.002	5999.6	0.008	2919.3	0.456	4006.3	0.447	2512.8	0.756
5	11999.5	0.003	6000.1	0.002	5999.4	0.004	2937.5	0.510	4391.0	0.410	2618.5	0.645
6	11999.8	0.002	6000.2	0.002	5999.6	0.003	2927.8	0.384	4614.0	0.309	2664.7	0.543
7	11999.7	0.003	6000.1	0.003	5999.6	0.004	2927.1	0.446	4792.8	0.521	2703.3	0.610
8	11999.7	0.001	6000.1	0.002	5999.6	0.003	2912.5	0.301	4908.6	0.140	2715.7	0.278
9	11999.7	0.003	6000.1	0.002	5999.6	0.004	2923.5	0.350	5028.0	0.153	2752.4	0.356
10	11999.5	0.004	6000.1	0.002	5999.4	0.006	2917.9	0.206	5111.5	0.170	2763.2	0.358
11	12000.0	0.001	6000.2	0.002	5999.8	0.001	2916.8	0.312	5177.6	0.252	2774.2	0.498
12	11999.6	0.002	6000.1	0.001	5999.6	0.004	2917.3	0.304	5238.4	0.104	2785.1	0.252
13	11999.6	0.002	6000.1	0.001	5999.6	0.004	2919.0	0.165	5283.0	0.248	2798.6	0.135
14	11999.5	0.002	6000.1	0.001	5999.5	0.004	2915.1	0.171	5329.1	0.111	2802.6	0.122
15	11999.9	0.001	6000.2	0.001	5999.7	0.003	2916.8	0.303	5363.6	0.104	2812.0	0.269
16	11999.8	0.002	6000.1	0.001	5999.7	0.004	2914.3	0.326	5401.4	0.125	2812.7	0.304

Tabela A.1: Médias e desvios padrão, cenário SP - 1

	FDAS-Partner		BHMR		BCS		BCS-Aftersend		BCS-Partner		HMNR	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
2	246.6	2.481	246.6	2.481	246.6	2.481	246.6	2.481	246.6	2.481	246.6	2.481
3	1656.1	0.910	1463.0	1.609	425.8	1.755	420.9	1.691	415.7	1.820	413.1	1.700
4	2154.5	1.176	2135.5	1.319	529.6	1.954	518.4	2.024	512.5	1.952	507.6	1.971
5	2381.3	0.288	2375.3	0.304	614.5	1.201	599.1	1.243	594.3	1.174	588.2	1.219
6	2488.7	0.326	2486.3	0.339	696.3	0.990	674.7	1.033	667.6	1.054	661.7	1.118
7	2566.3	0.607	2564.8	0.617	727.9	1.949	701.7	1.899	696.7	1.785	690.7	1.730
8	2599.3	0.326	2598.4	0.335	783.8	1.082	752.1	1.001	747.3	1.044	741.6	1.028
9	2651.5	0.316	2650.8	0.318	815.0	1.430	779.2	1.306	774.1	1.197	767.6	1.122
10	2675.5	0.335	2674.9	0.332	843.7	1.373	806.6	1.215	801.8	1.220	795.5	1.212
11	2695.7	0.446	2695.2	0.441	866.9	1.398	826.5	1.394	821.7	1.481	816.2	1.464
12	2715.2	0.315	2714.8	0.310	898.9	0.923	853.1	0.850	848.8	0.869	843.2	0.868
13	2735.9	0.096	2735.6	0.105	915.7	0.514	869.3	0.605	864.7	0.608	859.3	0.672
14	2741.2	0.152	2740.9	0.152	931.3	1.540	881.1	1.223	876.8	1.178	871.3	1.184
15	2754.7	0.197	2754.5	0.198	947.3	0.822	895.5	0.894	891.3	0.872	886.5	0.864
16	2761.8	0.359	2761.7	0.360	956.3	0.719	903.8	0.646	900.2	0.591	894.8	0.583

Tabela A.2: Médias e desvios padrão, cenário SP - 2

	Lazy-BCS		Lazy-BCS-Aftersend		Lazy-BCS-Partner		BQF		BQC	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
2	246.6	2.481	246.6	2.481	246.6	2.481	246.6	2.481	246.6	2.481
3	425.8	1.755	420.9	1.691	415.7	1.820	420.9	1.691	590.3	0.581
4	529.6	1.954	518.4	2.024	512.5	1.952	518.4	2.024	1031.8	1.753
5	614.5	1.201	599.1	1.243	594.3	1.174	599.1	1.243	1477.1	1.549
6	696.3	0.990	674.7	1.033	667.6	1.054	674.7	1.033	1826.3	1.043
7	727.9	1.949	701.7	1.899	696.7	1.785	701.7	1.899	2064.7	0.630
8	783.8	1.082	752.1	1.001	747.3	1.044	752.1	1.001	2236.5	0.447
9	815.0	1.430	779.2	1.306	774.1	1.197	779.2	1.306	2374.1	0.435
10	843.7	1.373	806.6	1.215	801.8	1.220	806.6	1.215	2471.1	0.234
11	866.9	1.398	826.5	1.394	821.7	1.481	826.5	1.394	2529.7	0.406
12	898.9	0.923	853.1	0.850	848.8	0.869	853.1	0.850	2584.0	0.364
13	915.7	0.516	869.4	0.608	864.7	0.610	869.4	0.608	2633.4	0.213
14	931.3	1.540	881.1	1.223	876.8	1.178	881.1	1.223	2660.4	0.131
15	947.3	0.822	895.5	0.894	891.3	0.872	895.5	0.894	2687.7	0.344
16	956.3	0.719	903.8	0.646	900.2	0.591	903.8	0.646	2702.1	0.302

Tabela A.3: Médias e desvios padrão, cenário SP - 3

	CASBR		CAS		CBR		NRAS		FDI		FDAS	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
4	71998.0	0.003	36000.9	0.002	35997.1	0.004	15088.5	0.444	30136.9	0.277	14084.7	0.551
10	71996.9	0.003	36000.7	0.002	35996.2	0.006	16497.5	0.499	28946.9	0.309	15236.2	0.455
16	71998.3	0.001	36000.5	0.002	35997.8	0.003	16941.3	0.556	28408.8	0.368	15568.2	0.639
22	71998.1	0.001	36000.4	0.001	35997.7	0.002	17227.5	0.459	28126.0	0.307	15761.6	0.496
28	71998.6	0.001	36000.7	0.001	35997.9	0.002	17336.0	0.584	27894.4	0.382	15834.8	0.616
34	71997.6	0.003	36000.6	0.003	35997.0	0.006	17449.6	0.326	27790.1	0.347	15902.9	0.453
40	71998.4	0.002	36000.5	0.001	35997.9	0.003	17577.2	0.501	27708.5	0.563	15997.9	0.728
46	71998.5	0.002	36000.4	0.001	35998.1	0.004	17595.6	0.397	27720.0	0.456	16046.8	0.637
52	71998.5	0.003	36001.1	0.002	35997.4	0.006	17646.1	0.211	27627.5	0.278	16074.3	0.313
58	71996.9	0.004	36000.8	0.003	35996.1	0.007	17639.9	0.512	27578.0	0.269	16041.4	0.614
64	71997.0	0.004	36000.6	0.002	35996.4	0.007	17713.1	0.449	27525.8	0.215	16103.0	0.436
70	71997.4	0.003	36001.1	0.002	35996.3	0.005	17737.0	0.456	27544.2	0.482	16118.7	0.664
76	71997.4	0.003	36000.6	0.002	35996.8	0.005	17743.5	0.486	27455.7	0.208	16115.2	0.410
82	71997.9	0.002	36000.5	0.002	35997.4	0.005	17748.6	0.516	27470.8	0.540	16117.6	0.688
88	71997.2	0.004	36000.4	0.002	35996.8	0.008	17766.1	0.423	27426.9	0.408	16128.0	0.514
94	71997.1	0.003	36000.6	0.002	35996.5	0.005	17786.0	0.384	27416.0	0.544	16136.6	0.641
100	71998.7	0.002	36000.7	0.003	35998.0	0.003	17785.4	0.394	27417.2	0.400	16171.1	0.378
106	71998.5	0.003	36000.6	0.003	35997.9	0.004	17823.2	0.266	27485.0	0.382	16171.5	0.486
112	71998.1	0.002	36000.2	0.001	35997.9	0.004	17821.9	0.335	27394.1	0.569	16171.9	0.446
118	71997.7	0.002	36000.2	0.001	35997.5	0.004	17806.0	0.342	27329.0	0.390	16142.2	0.711

Tabela A.4: Médias e desvios padrão, cenário SI - 1

	FDAS-Partner		BHMR		BCS		BCS-Aftersend		BCS-Partner		HMNR	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
4	12970.7	0.680	12964.7	0.673	6488.0	1.159	5246.2	0.942	5017.3	1.127	4915.5	1.132
10	14163.1	0.469	14152.8	0.460	5677.3	1.314	5179.7	1.190	5059.9	1.103	4986.8	1.074
16	14514.0	0.619	14501.6	0.629	5410.7	1.260	5069.6	1.306	4995.0	1.248	4929.3	1.264
22	14716.1	0.594	14700.9	0.566	5060.7	1.411	4808.1	1.476	4747.7	1.452	4691.6	1.438
28	14774.3	0.605	14757.2	0.610	4705.2	1.683	4508.5	1.835	4457.4	1.838	4409.9	1.853
34	14835.2	0.446	14819.7	0.441	4410.7	2.124	4255.2	1.955	4216.0	1.953	4174.2	1.943
40	14933.9	0.741	14916.7	0.755	4088.8	1.983	3958.8	1.906	3925.5	1.865	3888.3	1.972
46	14975.2	0.719	14959.4	0.731	3853.7	1.490	3745.8	1.330	3718.0	1.300	3689.1	1.302
52	15038.6	0.435	15025.1	0.435	3662.7	1.732	3571.3	1.829	3549.3	1.901	3524.6	1.939
58	15007.8	0.512	14992.2	0.513	3422.4	2.343	3342.5	2.218	3323.3	2.315	3301.4	2.264
64	15025.7	0.423	15009.1	0.439	3259.5	2.885	3194.0	2.956	3174.7	3.043	3153.2	3.065
70	15066.8	0.539	15049.5	0.523	3110.0	2.528	3049.8	2.482	3034.7	2.461	3015.7	2.386
76	15060.4	0.538	15046.2	0.554	2985.8	2.370	2930.4	2.406	2915.9	2.407	2899.5	2.542
82	15080.1	0.658	15063.9	0.661	2818.6	1.986	2768.0	1.841	2755.2	1.838	2738.6	1.720
88	15064.4	0.534	15048.8	0.533	2652.6	2.805	2609.4	2.869	2597.9	2.922	2583.9	2.928
94	15080.3	0.711	15064.9	0.721	2558.4	2.295	2518.7	2.375	2507.9	2.405	2494.7	2.438
100	15115.9	0.516	15101.7	0.501	2480.6	1.415	2444.5	1.463	2435.7	1.463	2424.0	1.509
106	15104.8	0.534	15087.6	0.556	2359.7	2.000	2326.7	1.963	2318.8	1.961	2308.4	1.992
112	15112.9	0.542	15094.2	0.544	2304.5	2.209	2273.8	2.301	2265.9	2.281	2254.3	2.253
118	15069.4	0.798	15051.0	0.808	2211.8	1.877	2186.0	1.935	2179.2	1.983	2171.4	1.961

Tabela A.5: Médias e desvios padrão, cenário SI - 2

	Lazy-BCS		Lazy-BCS-Aftersend		Lazy-BCS-Partner		BQF		BQC	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
4	6060.5	1.201	4819.5	1.264	4602.9	1.491	4815.3	1.238	11030.2	0.555
10	5665.5	1.443	5121.8	1.349	5003.0	1.269	5122.8	1.358	11455.8	0.631
16	5413.9	1.366	5063.2	1.450	4986.3	1.358	5062.9	1.438	11363.8	1.055
22	5064.7	1.401	4809.8	1.466	4749.3	1.439	4809.9	1.464	11281.8	0.748
28	4706.1	1.701	4509.2	1.854	4457.8	1.850	4508.8	1.846	11127.7	0.924
34	4410.7	2.124	4255.2	1.955	4216.0	1.953	4255.2	1.955	11057.3	1.106
40	4088.9	1.983	3958.8	1.906	3925.5	1.865	3958.8	1.906	10933.9	1.193
46	3853.7	1.490	3745.8	1.330	3718.0	1.300	3745.8	1.330	10787.2	0.765
52	3662.7	1.732	3571.3	1.829	3549.3	1.901	3571.3	1.829	10764.7	1.073
58	3422.4	2.343	3342.5	2.218	3323.3	2.315	3342.5	2.218	10595.4	1.730
64	3259.5	2.885	3194.0	2.956	3174.7	3.043	3194.0	2.956	10623.9	1.157
70	3110.0	2.528	3049.8	2.482	3034.7	2.461	3049.8	2.482	10567.4	1.243
76	2985.8	2.370	2930.4	2.406	2915.9	2.407	2930.4	2.406	10372.1	1.569
82	2818.6	1.986	2768.0	1.841	2755.2	1.838	2768.0	1.841	10295.8	1.220
88	2652.6	2.805	2609.4	2.869	2597.9	2.922	2609.4	2.869	10143.1	1.879
94	2558.4	2.295	2518.7	2.375	2507.9	2.405	2518.7	2.375	10122.3	1.663
100	2480.6	1.415	2444.5	1.463	2435.7	1.463	2444.5	1.463	10076.7	1.185
106	2359.7	2.000	2326.7	1.963	2318.8	1.961	2326.7	1.963	9928.7	1.880
112	2304.5	2.209	2273.8	2.301	2265.9	2.281	2273.8	2.301	9865.3	2.644
118	2211.8	1.877	2186.0	1.935	2179.2	1.983	2186.0	1.935	9760.2	2.058

Tabela A.6: Médias e desvios padrão, cenário SI - 3

	CASBR		CAS		CBR		NRAS		FDI		FDAS	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
2	71997.0	0.004	36000.5	0.002	35996.5	0.007	17571.6	0.356	27701.9	0.300	16013.4	0.404
4	71997.3	0.003	36000.4	0.001	35996.9	0.007	17574.0	0.423	27651.6	0.442	15982.5	0.499
6	71997.9	0.002	36000.6	0.003	35997.3	0.004	17621.7	0.326	27719.4	0.469	16048.4	0.454
8	71998.3	0.002	36001.0	0.002	35997.3	0.005	17562.4	0.475	27692.8	0.336	15986.3	0.430
10	71997.1	0.003	36000.5	0.002	35996.6	0.006	17588.7	0.450	27786.0	0.224	16061.2	0.460
12	71998.2	0.003	36001.0	0.003	35997.2	0.005	17573.5	0.698	27702.3	0.463	15990.6	0.772
14	71999.1	0.002	36001.1	0.003	35998.0	0.003	17559.7	0.207	27702.4	0.264	15984.8	0.369
16	71998.3	0.003	36000.6	0.002	35997.7	0.005	17515.6	0.494	27713.4	0.313	15961.9	0.633
18	71997.8	0.002	36000.6	0.002	35997.2	0.004	17525.5	0.417	27736.4	0.300	15965.8	0.476
20	71998.0	0.003	36000.8	0.002	35997.2	0.006	17545.4	0.324	27765.4	0.340	16003.5	0.499
22	71997.8	0.003	36000.7	0.001	35997.1	0.006	17508.9	0.513	27754.7	0.316	15955.4	0.606
24	71997.7	0.003	36000.8	0.002	35996.9	0.005	17514.5	0.339	27766.6	0.283	15955.7	0.398
26	71997.2	0.003	36000.5	0.002	35996.7	0.006	17512.6	0.466	27803.5	0.386	15967.4	0.655
28	71998.3	0.002	36000.4	0.002	35997.9	0.004	17490.7	0.397	27785.0	0.371	15938.8	0.424
30	71998.0	0.002	36000.4	0.001	35997.6	0.004	17491.2	0.359	27808.4	0.444	15944.5	0.488
32	71998.0	0.002	36000.4	0.001	35997.6	0.004	17402.7	0.473	27875.2	0.503	15896.1	0.720
34	71997.5	0.003	36000.5	0.002	35997.0	0.005	17406.5	0.456	27886.1	0.324	15869.4	0.454
36	71996.5	0.002	36000.1	0.001	35996.4	0.004	17365.1	0.446	27912.9	0.153	15848.0	0.450
38	71997.0	0.004	36001.0	0.003	35996.0	0.007	17236.0	0.289	27989.5	0.432	15758.8	0.495
40	71998.5	0.003	36001.0	0.003	35997.5	0.003	17145.6	0.328	28088.6	0.350	15678.0	0.474

Tabela A.7: Médias e desvios padrão, cenário AV - 1

	FDAS-Partner		BHMR		BCS		BCS-Aftersend		BCS-Partner		HMNR	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
2	14968.6	0.426	14952.8	0.429	3996.3	1.744	3875.9	1.707	3845.3	1.707	3814.1	1.786
4	14921.9	0.485	14905.7	0.480	3963.7	1.612	3846.6	1.367	3816.6	1.294	3781.4	1.334
6	14980.5	0.575	14963.7	0.582	4005.8	1.995	3888.2	2.031	3860.8	2.146	3826.4	2.163
8	14937.3	0.349	14920.5	0.325	4081.5	1.550	3955.7	1.464	3927.6	1.560	3896.4	1.588
10	14983.3	0.398	14967.4	0.394	4016.9	2.367	3897.6	2.466	3861.8	2.453	3828.4	2.420
12	14910.3	0.742	14894.4	0.733	4116.6	1.929	3985.8	1.865	3953.0	1.903	3922.3	1.904
14	14932.1	0.374	14917.0	0.377	4099.5	1.314	3973.4	1.269	3940.2	1.220	3907.4	1.229
16	14894.6	0.537	14877.2	0.514	4144.0	1.412	4001.7	1.412	3970.3	1.417	3935.9	1.445
18	14893.9	0.504	14879.6	0.512	4150.3	1.219	4016.6	1.369	3980.4	1.497	3944.9	1.517
20	14933.1	0.500	14917.4	0.505	4149.2	1.360	4018.1	1.398	3981.6	1.330	3942.7	1.286
22	14891.0	0.531	14873.7	0.547	4217.2	1.920	4075.8	1.878	4040.2	1.791	4001.9	1.796
24	14887.6	0.397	14872.6	0.395	4329.4	1.763	4186.8	1.673	4150.6	1.644	4112.6	1.606
26	14907.3	0.551	14892.4	0.550	4384.3	1.719	4229.5	1.767	4190.9	1.686	4148.4	1.762
28	14897.8	0.490	14882.6	0.498	4449.3	1.123	4293.4	1.119	4253.1	1.093	4208.5	1.147
30	14884.6	0.390	14871.6	0.410	4606.2	1.656	4432.9	1.665	4385.3	1.614	4342.4	1.473
32	14841.5	0.630	14829.8	0.623	4788.3	1.008	4589.6	1.092	4543.1	0.971	4492.9	0.946
34	14780.3	0.459	14768.6	0.459	5028.4	1.165	4798.6	1.142	4740.4	1.079	4681.6	1.076
36	14785.5	0.457	14770.8	0.445	5518.1	1.593	5196.5	1.394	5115.7	1.317	5036.2	1.357
38	14695.7	0.507	14681.3	0.494	6133.1	1.131	5659.3	1.108	5551.2	1.083	5447.2	1.152
40	14612.9	0.500	14598.9	0.527	7103.6	1.283	6356.3	1.188	6194.4	1.148	6043.1	1.192

Tabela A.8: Médias e desvios padrão, cenário AV - 2

	Lazy-BCS		Lazy-BCS-Aftersend		Lazy-BCS-Partner		BQF		BQC	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
2	3996.3	1.744	3875.9	1.707	3845.3	1.707	3875.9	1.707	10851.9	1.051
4	3963.7	1.612	3846.6	1.367	3816.6	1.294	3846.6	1.367	10853.5	1.073
6	4005.8	1.995	3888.2	2.031	3860.8	2.146	3888.2	2.031	10931.4	0.954
8	4081.5	1.550	3955.7	1.464	3927.6	1.560	3955.7	1.464	10874.9	0.886
10	4016.9	2.367	3897.6	2.466	3861.8	2.453	3897.6	2.466	10924.2	1.314
12	4116.7	1.924	3985.9	1.860	3953.1	1.898	3985.9	1.860	10937.0	1.112
14	4099.7	1.320	3973.5	1.274	3940.3	1.224	3973.5	1.274	10969.8	0.713
16	4143.7	1.410	4001.4	1.411	3970.0	1.418	4001.4	1.411	10964.4	0.728
18	4149.7	1.216	4015.7	1.356	3979.5	1.485	4015.7	1.356	10957.9	1.036
20	4148.5	1.367	4016.7	1.408	3980.2	1.338	4016.7	1.408	10959.4	1.325
22	4214.6	1.985	4073.6	1.931	4037.9	1.842	4073.1	1.940	11017.6	0.941
24	4320.8	1.768	4178.6	1.668	4142.5	1.627	4178.6	1.668	11101.5	1.129
26	4369.0	1.679	4214.5	1.719	4176.6	1.635	4214.6	1.718	10994.4	1.273
28	4427.8	1.023	4272.8	1.002	4232.9	0.967	4271.4	1.022	11118.6	0.456
30	4564.2	1.671	4393.5	1.676	4346.6	1.619	4392.9	1.676	11109.9	0.773
32	4676.2	1.231	4490.9	1.340	4447.8	1.231	4488.9	1.312	11161.0	1.170
34	4783.7	1.319	4588.9	1.228	4541.7	1.249	4585.9	1.162	11171.6	0.778
36	5011.5	1.781	4782.3	1.651	4723.7	1.600	4772.4	1.702	11253.8	0.660
38	5205.1	1.149	4933.6	1.088	4871.0	1.073	4924.0	1.048	11295.7	0.866
40	5417.4	1.672	5114.5	1.377	5040.6	1.300	5101.5	1.359	11340.7	0.747

Tabela A.9: Médias e desvios padrão, cenário AV - 3

	CASBR		CAS		CBR		NRAS		FDI		FDAS	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
2	11999.5	0.003	6000.1	0.004	5999.4	0.009	2924.9	0.923	2442.1	1.600	2136.6	1.401
3	11999.7	0.002	6000.1	0.002	5999.7	0.004	2897.3	0.378	3484.7	0.962	2348.3	0.768
4	11999.6	0.002	6000.1	0.002	5999.6	0.004	2904.9	0.458	4059.9	0.412	2515.6	0.545
5	11999.5	0.003	6000.1	0.003	5999.4	0.005	2898.5	0.341	4407.5	0.453	2590.3	0.384
6	11999.7	0.001	6000.1	0.002	5999.6	0.002	2913.8	0.637	4636.2	0.393	2653.6	0.656
7	11999.6	0.002	6000.1	0.001	5999.5	0.005	2915.4	0.430	4820.3	0.382	2701.9	0.494
8	11999.6	0.003	6000.1	0.003	5999.5	0.004	2919.6	0.285	4947.1	0.438	2727.9	0.373
9	11999.6	0.004	6000.1	0.001	5999.4	0.007	2911.8	0.284	5028.8	0.214	2743.1	0.302
10	11999.6	0.002	6000.1	0.001	5999.6	0.005	2914.2	0.145	5118.3	0.092	2759.5	0.150
11	11999.5	0.002	6000.0	0.001	5999.5	0.004	2908.8	0.286	5174.3	0.218	2766.9	0.307
12	11999.7	0.002	6000.1	0.002	5999.6	0.005	2910.2	0.248	5230.3	0.098	2777.5	0.152
13	11999.7	0.001	6000.1	0.001	5999.5	0.003	2914.1	0.317	5288.5	0.074	2793.6	0.316
14	11999.8	0.002	6000.2	0.002	5999.6	0.003	2911.4	0.358	5325.0	0.184	2797.3	0.342
15	11999.8	0.002	6000.2	0.002	5999.6	0.004	2911.6	0.240	5370.0	0.082	2807.3	0.211
16	11999.7	0.002	6000.1	0.001	5999.6	0.005	2916.0	0.166	5400.6	0.170	2815.8	0.144

Tabela A.10: Médias e desvios padrão, cenário AP - 1

	FDAS-Partner		BHMR		BCS		BCS-Aftersend		BCS-Partner		HMNR	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
2	398.1	2.256	398.1	2.256	404.9	2.199	403.3	2.065	398.1	2.256	398.1	2.256
3	1671.1	0.675	1537.0	1.373	562.3	1.852	550.3	1.784	540.6	1.589	535.5	1.514
4	2148.3	0.867	2131.0	0.874	664.9	0.830	646.6	0.719	637.1	1.015	630.5	0.944
5	2358.2	0.572	2353.0	0.603	723.8	2.086	698.4	2.278	691.0	2.316	683.9	2.266
6	2474.2	0.578	2471.7	0.550	772.5	1.272	743.8	1.282	736.2	1.442	728.4	1.453
7	2563.9	0.416	2562.7	0.437	809.5	1.655	776.7	1.518	768.8	1.423	760.9	1.302
8	2609.0	0.311	2607.9	0.317	839.9	1.519	800.9	1.410	794.1	1.436	786.5	1.384
9	2642.9	0.205	2642.4	0.205	856.3	0.837	818.0	0.727	812.7	0.708	805.8	0.666
10	2672.3	0.061	2671.7	0.056	881.9	1.288	839.0	1.163	834.0	1.238	827.5	1.205
11	2688.9	0.318	2688.2	0.317	898.7	0.955	854.1	1.078	849.2	1.116	842.8	1.047
12	2706.9	0.137	2706.6	0.138	915.9	0.928	869.1	0.812	864.3	0.867	858.1	0.904
13	2730.7	0.333	2730.4	0.338	938.5	1.015	887.0	0.980	882.4	0.969	876.4	0.920
14	2739.0	0.348	2738.7	0.346	947.8	1.691	895.1	1.509	890.4	1.550	884.7	1.447
15	2754.3	0.249	2754.0	0.245	961.2	1.265	907.0	1.224	902.9	1.228	897.5	1.190
16	2764.3	0.143	2764.1	0.145	969.3	1.320	914.2	0.972	910.3	0.972	904.8	0.968

Tabela A.11: Médias e desvios padrão, cenário AP - 2

	Lazy-BCS		Lazy-BCS-Aftersend		Lazy-BCS-Partner		BQF		BQC	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
2	399.7	2.331	399.0	2.226	398.1	2.256	398.7	2.133	398.1	2.256
3	553.5	1.811	541.9	1.684	533.1	1.499	541.3	1.765	762.5	1.767
4	656.2	1.108	638.9	0.870	629.8	1.193	638.6	0.898	1173.8	2.303
5	713.8	1.864	689.3	2.053	682.3	2.075	689.0	2.044	1549.4	0.926
6	764.3	0.967	736.2	1.088	728.8	1.212	736.0	1.137	1846.4	0.696
7	801.1	1.695	769.5	1.525	761.6	1.435	769.2	1.540	2098.2	0.652
8	830.4	1.474	793.1	1.310	786.7	1.357	792.9	1.292	2253.0	0.591
9	848.1	1.156	810.6	0.956	805.5	0.943	810.3	0.918	2380.6	0.236
10	876.6	1.082	834.3	0.988	829.4	1.050	834.2	1.009	2468.9	0.361
11	892.2	0.744	848.3	0.897	843.5	0.920	848.3	0.894	2530.5	0.283
12	909.3	0.902	862.9	0.806	858.1	0.857	863.0	0.816	2582.4	0.290
13	931.8	1.116	881.5	1.040	876.9	1.064	881.5	1.024	2628.0	0.385
14	943.6	1.913	891.7	1.720	887.2	1.761	891.7	1.715	2656.0	0.338
15	956.9	1.380	903.4	1.348	899.3	1.358	903.3	1.351	2687.5	0.302
16	964.1	1.305	909.4	0.921	905.6	0.915	909.4	0.920	2708.9	0.132

Tabela A.12: Médias e desvios padrão, cenário AP - 3

	CASBR		CAS		CBR		NRAS		FDI		FDAS	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
4	71995.8	0.005	36000.6	0.002	35995.2	0.012	17063.3	0.282	28238.7	0.363	15619.4	0.487
10	71997.3	0.003	36000.5	0.001	35996.8	0.006	17338.6	0.428	27908.2	0.378	15843.1	0.509
16	71997.8	0.002	36000.6	0.002	35997.2	0.005	17460.3	0.527	27771.2	0.328	15916.1	0.472
22	71997.7	0.002	36001.1	0.003	35996.6	0.004	17589.2	0.514	27695.0	0.342	16010.0	0.638
28	71998.6	0.002	36000.8	0.003	35997.8	0.004	17629.1	0.355	27641.9	0.380	16055.2	0.487
34	71998.2	0.002	36000.3	0.001	35997.9	0.004	17682.0	0.279	27646.3	0.412	16107.8	0.440
40	71997.1	0.003	36000.5	0.001	35996.6	0.006	17649.2	0.317	27485.9	0.352	16036.9	0.336
46	71997.5	0.002	36000.4	0.001	35997.1	0.004	17681.9	0.304	27526.3	0.220	16070.7	0.374
52	71998.7	0.002	36000.8	0.001	35997.9	0.004	17738.5	0.515	27458.7	0.265	16110.2	0.647
58	71997.8	0.003	36000.6	0.001	35997.2	0.006	17763.1	0.332	27491.0	0.431	16137.9	0.476
64	71997.9	0.003	36000.4	0.001	35997.5	0.005	17800.4	0.452	27486.3	0.461	16190.3	0.436
70	71996.5	0.005	36000.2	0.001	35996.3	0.010	17744.2	0.262	27426.4	0.475	16134.2	0.390
76	71997.6	0.002	36000.4	0.002	35997.2	0.003	17803.0	0.374	27399.7	0.400	16155.7	0.437
82	71998.5	0.003	36000.4	0.002	35998.1	0.004	17765.8	0.623	27403.5	0.325	16153.5	0.682
88	71998.1	0.003	36000.5	0.002	35997.6	0.004	17801.7	0.325	27350.4	0.385	16143.1	0.443
94	71998.3	0.003	36000.8	0.003	35997.5	0.005	17819.8	0.387	27441.7	0.298	16196.1	0.416
100	71998.6	0.002	36000.4	0.002	35998.2	0.003	17807.1	0.426	27425.9	0.407	16193.2	0.534
106	71997.6	0.003	36000.5	0.002	35997.1	0.006	17845.2	0.309	27383.7	0.419	16198.5	0.397
112	71998.4	0.003	36001.3	0.003	35997.1	0.006	17817.3	0.364	27354.0	0.490	16173.5	0.530
118	71997.6	0.002	36000.5	0.002	35997.1	0.003	17798.2	0.378	27308.7	0.261	16151.7	0.355

Tabela A.13: Médias e desvios padrão, cenário AI - 1

	FDAS-Partner		BHMR		BCS		BCS-Aftersend		BCS-Partner		HMNR	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
4	14563.9	0.526	14550.7	0.520	7046.9	0.768	6294.2	0.795	6132.0	0.828	5985.0	0.811
10	14781.3	0.364	14766.7	0.356	5066.5	1.110	4820.3	1.166	4757.1	1.229	4699.8	1.158
16	14867.1	0.305	14849.3	0.329	4457.0	1.780	4296.8	1.632	4257.8	1.582	4218.7	1.594
22	14955.4	0.594	14936.8	0.588	4081.0	2.504	3960.8	2.301	3929.7	2.335	3896.9	2.290
28	14980.1	0.559	14965.3	0.553	3770.7	0.940	3675.9	0.964	3648.7	0.929	3619.7	0.901
34	15018.8	0.642	15004.5	0.645	3522.1	1.847	3442.1	1.830	3418.8	1.756	3395.8	1.744
40	14956.6	0.400	14939.8	0.396	3345.5	2.082	3269.7	2.213	3249.0	2.280	3227.2	2.253
46	15028.0	0.296	15011.5	0.288	3145.0	0.989	3083.0	0.959	3064.3	0.938	3046.0	0.929
52	15032.6	0.677	15016.5	0.699	2969.4	2.171	2919.0	2.090	2903.6	2.058	2887.5	1.995
58	15064.5	0.500	15047.4	0.496	2814.6	2.246	2767.7	2.295	2754.7	2.255	2738.7	2.399
64	15114.0	0.390	15098.0	0.376	2711.8	0.918	2666.6	0.885	2654.0	0.895	2640.7	0.936
70	15069.0	0.584	15052.2	0.585	2621.5	1.920	2585.6	2.014	2575.1	2.058	2562.8	2.015
76	15114.7	0.461	15099.6	0.457	2486.8	1.869	2454.0	1.961	2445.1	2.006	2433.5	1.972
82	15108.0	0.667	15090.2	0.647	2379.7	2.412	2349.1	2.461	2341.0	2.490	2331.5	2.475
88	15085.7	0.507	15068.6	0.500	2268.8	3.917	2237.9	3.918	2231.1	3.896	2221.2	3.811
94	15140.5	0.525	15123.9	0.539	2211.6	2.514	2183.8	2.658	2176.0	2.678	2167.6	2.654
100	15149.5	0.535	15131.6	0.540	2136.3	1.602	2112.5	1.656	2105.3	1.738	2097.4	1.746
106	15128.7	0.408	15110.3	0.414	2073.4	2.603	2052.4	2.623	2046.1	2.624	2037.4	2.582
112	15133.6	0.546	15115.9	0.539	1991.4	2.617	1969.1	2.723	1962.8	2.756	1955.4	2.789
118	15087.5	0.369	15067.8	0.377	1922.5	3.019	1903.1	2.886	1897.5	2.983	1890.5	2.948

Tabela A.14: Médias e desvios padrão, cenário AI - 2

	Lazy-BCS		Lazy-BCS-Aftersend		Lazy-BCS-Partner		BQF		BQC	
	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio	Média	Desvio
4	5519.9	0.550	5185.7	0.609	5105.8	0.652	5168.3	0.561	11351.2	0.769
10	4853.1	1.238	4639.7	1.283	4583.4	1.350	4636.9	1.321	11185.9	0.866
16	4434.2	1.820	4276.8	1.672	4238.5	1.633	4276.5	1.669	11042.6	0.773
22	4074.9	2.528	3955.1	2.307	3924.1	2.330	3955.1	2.307	10949.9	0.990
28	3770.8	0.939	3676.0	0.963	3648.8	0.929	3676.0	0.963	10868.7	1.225
34	3522.1	1.847	3442.1	1.830	3418.8	1.756	3442.1	1.830	10716.8	1.839
40	3345.5	2.082	3269.7	2.213	3249.0	2.280	3269.7	2.213	10615.0	0.939
46	3145.0	0.989	3083.0	0.959	3064.3	0.938	3083.0	0.959	10572.3	1.342
52	2969.4	2.171	2919.0	2.090	2903.6	2.058	2919.0	2.090	10506.7	1.387
58	2814.6	2.246	2767.7	2.295	2754.7	2.255	2767.7	2.295	10379.5	1.463
64	2711.8	0.918	2666.6	0.885	2654.0	0.895	2666.6	0.885	10261.2	0.904
70	2621.5	1.920	2585.6	2.014	2575.1	2.058	2585.6	2.014	10252.1	1.846
76	2486.8	1.869	2454.0	1.961	2445.1	2.006	2454.0	1.961	10087.9	1.713
82	2379.7	2.412	2349.1	2.461	2341.0	2.490	2349.1	2.461	10051.8	1.767
88	2268.8	3.917	2237.9	3.918	2231.1	3.896	2237.9	3.918	9841.0	1.833
94	2211.6	2.514	2183.8	2.658	2176.0	2.678	2183.8	2.658	9841.5	2.466
100	2136.3	1.602	2112.5	1.656	2105.3	1.738	2112.5	1.656	9759.6	2.804
106	2073.4	2.603	2052.4	2.623	2046.1	2.624	2052.4	2.623	9620.9	1.441
112	1991.4	2.617	1969.1	2.723	1962.8	2.756	1969.1	2.723	9715.1	2.051
118	1922.5	3.019	1903.1	2.886	1897.5	2.983	1903.1	2.886	9512.0	2.999

Tabela A.15: Médias e desvios padrão, cenário AI - 3

Bibliografia

[1] L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain, and A. D. Mel. An analysis of communication-induced checkpointing. In *29th IEEE Symposium on Fault-Tolerant Computing (FTCS)*, pages 242–249, June 1999.

[2] Ö. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.

[3] R. Baldoni, J.-M. Hélary, A. Mostefaoui, and M. Raynal. A communication-induced checkpoint protocol that ensures rollback dependency trackability. In *27th IEEE Symposium on Fault Tolerant Computing (FTCS)*, pages 68–77, June 1997.

[4] R. Baldoni, F. Quaglia, and B. Ciciani. A VP-accordant checkpointing protocol preventing useless checkpoints. In *17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 61–67, 1998.

[5] R. Baldoni, F. Quaglia, and P. Fornara. An index-based checkpoint algorithm for autonomous distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):181–192, Feb. 1999.

[6] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *4th IEEE Symposium on Reliability in Distributed Software and Database Systems*, pages 207–215, Oct. 1984.

[7] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1989. ISBN 0-26-203141-8.

[9] E. N. Elnozahy, D. B. Johnson, and Y.-M. Wang. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Oct. 1996.

[10] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *10th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 134–141, May 1990.

[11] I. C. Garcia. Estados globais consistentes em sistemas distribuídos. Master's thesis, Instituto de Computação, Universidade Estadual de Campinas, July 1998.

[12] I. C. Garcia and L. E. Buzato. Progressive construction of consistent global checkpoints. In *19th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 55–62, May 1999.

[13] I. C. Garcia, G. M. D. Vieira, and L. E. Buzato. RDT-Partner: An efficient checkpointing protocol that enforces rollback-dependency trackability. In *19º Simpósio Brasileiro de Redes de Computadores (SBRC)*, May 2001.

[14] R. Gerth. *Concise Promela Reference.* Eindhoven University, Germany, Aug. 1997. http://cm.bell-labs.com/cm/cs/what/spin/Man/Quick.html.

[15] J.-M. Hélary, A. Mostefaoui, R. Netzer, and M. Raynal. Communication-based prevention of useless checkpoints in distributed computations. Technical Report 1105, IRISA, May 1997.

[16] J.-M. Hélary, A. Mostefaoui, R. Netzer, and M. Raynal. Preventing useless checkpoints in distributed computations. In *16th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 183–190, Oct. 1997.

[17] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991. ISBN 0-13-539925-4.

[18] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[19] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient distributed detection of conjunctions of local predicates in asynchronous computations. In *8th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 588–594, Oct. 1996.

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[21] D. Manivannan and M. Singhal. A low-overhead recovery technique using quasi-synchronous checkpointing. In *16th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 100–107, May 1996.

[22] D. Manivannan and M. Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):703–713, July 1999.

[23] R. H. B. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, Feb. 1995.

[24] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[25] M. Raynal and M. Singhal. Logical time: Capturing causality in distributed systems. *IEEE Computer*, 29(2):49–56, Feb. 1996.

[26] D. L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, Mar. 1980.

[27] K. Venkatesh, T. Radhakrishnan, and H. F. Li. Optimal checkpointing and local recording for domino-free rollback recovery. *Information Processing Letters*, 25(5):295–303, July 1987.

[28] G. M. D. Vieira, I. C. Garcia, and L. E. Buzato. Systematic analysis of index-based checkpointing algorithms using simulation. In *IX Brazilian Symposium on Fault-Tolerant Computing (SCTF)*, pages 31–42, Mar. 2001.

[29] L. Wall, R. L. Schwartz, and S. Talbot. *Programming Perl.* O'Reilly & Associates, 2nd edition, Sept. 1996. ISBN 1-56-592149-6.

[30] Y.-M. Wang. Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Transactions on Computers*, 46(4):456–468, Apr. 1997.

[31] Y.-M. Wang and W. K. Fuchs. Lazy checkpoint coordination for bounding rollback propagation. In *12th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 78–85, Oct. 1993.

[32] J. Xu and R. H. B. Netzer. Adaptive independent checkpointing for reducing rollback propagation. In *5th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 754–761, Dec. 1993.

[33] F. Zambonelli. On the effectiveness of distributed checkpoint algorithms for domino-free recovery. In *7th IEEE Symposium on High Performance Distributed Computing (HPDC)*, July 1998.