



**UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS DE CHAPECÓ
CURSO DE CIÊNCIA DA COMPUTAÇÃO**

CLEITON PUTTLITZ

**INTERVALO ADAPTATIVO ENTRE CHECKPOINTS EM SIMULAÇÃO
DISTRIBUÍDA**

**CHAPECÓ
2021**

CLEITON PUTTLITZ

**INTERVALO ADAPTATIVO ENTRE CHECKPOINTS EM SIMULAÇÃO
DISTRIBUÍDA**

Trabalho de conclusão de curso apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.
Orientador: Prof. Dr. Braulio Adriano de Mello

**CHAPECÓ
2021**

Puttlitz, Cleiton

Intervalo adaptativo entre checkpoints em simulação distribuída /
Cleiton Puttlitz. – 2021.

34 f.: il.

Orientador: Prof. Dr. Braulio Adriano de Mello.

Trabalho de conclusão de curso (graduação) – Universidade Federal
da Fronteira Sul, curso de Ciência da Computação, Chapecó, SC, 2021.

1. Simulação computacional. 2. Simulação distribuída. 3. Intervalo
entre Checkpoints. I. Mello, Prof. Dr. Braulio Adriano de, orientador.
II. Universidade Federal da Fronteira Sul. III. Título.

© 2021

Todos os direitos autorais reservados a Cleiton Puttlitz. A reprodução de partes ou do todo deste trabalho só poderá ser feita mediante a citação da fonte.

E-mail: cleitonputtlitz@gmail.com

CLEITON PUTTLITZ

**INTERVALO ADAPTATIVO ENTRE CHECKPOINTS EM SIMULAÇÃO
DISTRIBUÍDA**

Trabalho de conclusão de curso apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.

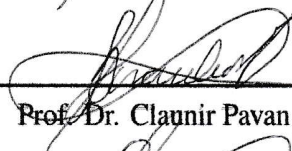
Orientador: Prof. Dr. Bráulio Adriano de Mello

Aprovado em: 21/05/2021.

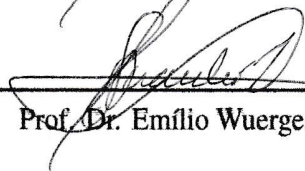
BANCA AVALIADORA



Prof. Dr. Bráulio Adriano de Mello – UFFS



Prof. Dr. Claunir Pavan – UFFS



Prof. Dr. Emílio Wuerges – UFFS

RESUMO

A simulação otimista em ambiente distribuído está sujeita a falhas de violação de tempo. *Checkpoints* podem ser usados para salvar estados da simulação e permitir que operações de *rollback* retomem a computação a partir de um estado consistente anterior a falha, sem que seja necessário voltar ao início da simulação. A falta de um controle eficaz da distância entre os *checkpoints* pode resultar em intervalos muito grandes ou muito pequenos, causando prejuízos para o desempenho da simulação. Neste trabalho, é apresentado um mecanismo capaz de ajustar ou adaptar os intervalos entre os *checkpoints* com base em um modelo que calcula o custo de recuperação para um determinado estado da simulação em caso de *rollback*.

Palavras-chave: Simulação computacional. Simulação distribuída. Intervalo entre Checkpoints.

ABSTRACT

Optimist simulation in distributed systems is subject to local constraint causality. Checkpoints may be used to save states of simulation and allow rollback operations to resume computing from a consistent state prior to failure, without having to go back to the beginning of the simulation. The lack of effective control of the distance between checkpoints can result in very large or very small intervals, prejudice the simulation perform. In this work, is presented a mechanism able to adjust or adapt the intervals between checkpoints based on a model that calculates the cost of recovery for a given state of the simulation in case of rollback.

Keywords: Computer simulation. Distributed simulation. Checkpoint interval.

LISTA DE ILUSTRAÇÕES

Figura 1 – Intervalos entre checkpoints	12
Figura 2 – Exemplo de simulação de componentes síncronos e assíncronos	15
Figura 3 – Efeito Dominó (ELNOZAHY et al., 2002)	16
Figura 4 – Exemplo de utilização de um <i>checkpoint</i> (MELLO, 2005)	17
Figura 5 – Arquitetura do DCB (MELLO, 2005)	19
Figura 6 – Exemplo custo de recuperação	23
Figura 7 – Relação da troca de mensagens entre os processos	26
Figura 8 – Linha do tempo do processo C - Checkpoints periódicos	27
Figura 9 – Linha do tempo do processo C - Estados seguros	28
Figura 10 – Linha do tempo do processo C - Custo de recuperação	28
Figura 11 – Comparação dos resultados	29
Figura 12 – Tempo total de execução	32

LISTA DE ALGORITMOS

Algoritmo 1 – Custo de recuperação	24
--	----

LISTA DE TABELAS

Tabela 1 – Resultados obtidos	29
Tabela 2 – Tempo total de execução - Checkpoints periódicos	31
Tabela 3 – Tempo total de execução - Estados seguros	31
Tabela 4 – Tempo total de execução - Custo de recuperação	31

LISTA DE ABREVIATURAS E SIGLAS

GVT Global Virtual Time

LCC Local Causality Constraint

LVT Local Virtual Time

PL Processo Lógico

SUMÁRIO

1	INTRODUÇÃO	11
2	REFERENCIAL TEÓRICO	13
2.1	SIMULAÇÃO	13
2.2	SIMULAÇÃO DISTRIBUÍDA	13
2.2.1	Sincronização	14
2.2.1.1	Simulação Síncrona	14
2.2.1.2	Simulação Assíncrona	15
2.2.2	Rollback	15
2.2.3	Checkpoints	16
2.2.3.1	Checkpoints coordenados	17
2.2.3.2	Checkpoints não coordenados	18
2.2.3.3	Checkpoints baseados em comunicação	18
2.2.3.4	Garbage Collection	18
2.3	DISTRIBUTED CO-SIMULATION BACKBONE	18
2.3.1	Visão Geral	18
2.3.2	Criação de Checkpoints	20
3	TRABALHOS RELACIONADOS	21
4	CRIAÇÃO DE CHECKPOINTS BASEADO NO CUSTO DE RECUPERAÇÃO	22
4.1	ESPECIFICAÇÃO DA SOLUÇÃO	22
4.2	IMPLEMENTAÇÃO	24
4.3	ESTUDO DE CASO	26
4.4	ANÁLISE DOS RESULTADOS	27
5	CONCLUSÃO	33
5.1	TRABALHOS FUTUROS	33
	REFERÊNCIAS	34

1 INTRODUÇÃO

A simulação computacional permite representar o comportamento de sistemas, sejam eles reais ou hipotéticos, por meio de um modelo computacional que incorpora as suas características. Com isso, a simulação pode ser usada em etapas de projeto de novos sistemas, analisar o comportamento de sistemas existentes, entre outros, antes mesmo de efetuar qualquer alteração no processo já existente ou construção de protótipos.

Simulações podem ser executadas de forma centralizada ou distribuída. Na simulação centralizada um modelo é inteiramente executado em um único recurso de processamento. Com a utilização de elementos distribuídos, os modelos são divididos em partes que se comunicam através da troca de mensagens por meio de uma rede de comunicação. Cada mensagem contém informações do evento que deve ser realizado e o instante de tempo, ou *timestamp*, em que deve ocorrer no processo de destino.

Afora o potencial computacional dos sistemas distribuídos, o gerenciamento da simulação é dificultado pela necessidade de manter a sincronização entre as partes do modelo, ou processos. A sincronização visa garantir que os eventos ocorram na ordem correta, onde todos os eventos são processados na ordem definida pelo *timestamp* (FUJIMOTO, R., 2015).

A ordem de execução de eventos internos de um processo utiliza como referência um tempo local de simulação, ou *Local Virtual Time* (LVT). A ordem de execução de eventos externos é controlado por um tempo global reconhecido em todos os processos do modelo, chamado de *Global Virtual Time* (GVT) (FUJIMOTO, R., 2015).

Os processos otimistas ou assíncronos, avançam no tempo continuamente e permitem que outro processo solicite a execução de um evento com *timestamp* menor do que o seu LVT, gerando então uma falha de violação de tempo.

Ao ser identificada uma violação de tempo, o processo com falha realiza uma operação de *rollback* para retornar para um estado consistente anterior ao tempo da mensagem recebida. *Checkpoints* podem ser usados para salvar informações que permitam a retomada da simulação em um estado consistente anterior a falha.

Em um modelo de simulação distribuído, cada processo pode desempenhar funções distintas durante a simulação e com isso ter um comportamento diverso no que se refere ao número de eventos executados, quantidade de memória utilizada e número de *rollbacks*, por exemplo. Além disso, o comportamento de um mesmo processo pode variar de acordo com as atividades realizadas durante a execução da simulação. Em virtude dessa variação, há uma dificuldade em se estabelecer um intervalo fixo ótimo para criação de *checkpoints* que seja válido para todos os processos participantes e durante todo o tempo da simulação (RÖNNGREN; AYANI, 1994).

A falta de um controle eficaz para o intervalo entre *checkpoints* pode resultar em intervalos muito grandes ou muito pequenos entre um e outro, podendo causar prejuízos para o desempenho da simulação.

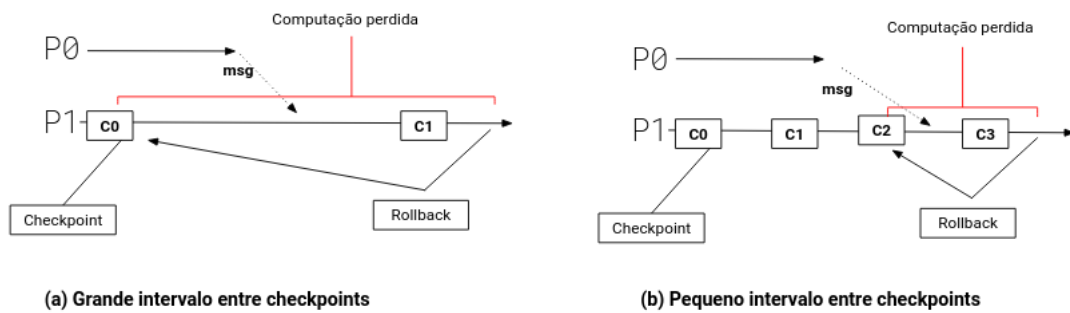


Figura 1 – Intervalos entre checkpoints

Em um intervalo muito grande, como pode ser observado na Figura 1.a, o tempo de recuperação após uma falha aumenta, visto que há um maior retrocesso em caso de falha. Um retrocesso maior pode significar um maior número de eventos que precisam ser reprocessados. No entanto, grandes intervalos entre os *checkpoints* necessitam de menos memória para o armazenamento dos *checkpoints*, visto que um menor número deles será criado.

Pequenos intervalos, exemplificados na Figura 1.b, reduzem o tempo consumido em operações de retrocesso, pois há um menor número de eventos para serem reprocessados em caso de *rollback*. Entretanto, gerar *checkpoints* muito próximos tende a gerar *checkpoints* inúteis, aumentar o espaço necessário para armazenamento e o tempo utilizado pelo sistema na criação de *checkpoints*.

As abordagens adaptativas de criação de *checkpoints* analisam dados de características de cada processo participante da simulação, coletados durante um determinado período de tempo. Estes dados são então utilizados para recalculer o intervalo de criação de *checkpoints* ideal para cada processo, podendo gerar *checkpoints* mais distantes um do outro em determinados momentos e mais frequentes em outros.

As contribuições deste trabalho agregam ao DCB, que é uma arquitetura de sistemas de simulação distribuída, funcionalidades para que antes de criar *checkpoints*, os processos decidam se o estado é conveniente analisando o custo de recuperação para o estado em caso de uma possível ocorrência de *rollback*. Um *checkpoint* somente é criado se o custo para salvar o *checkpoint* em determinado estado da simulação for menor ou igual ao custo de restaurar um *checkpoint* anterior e reprocessar os eventos até a simulação atingir novamente o estado analisado. O método implementado considera ainda a probabilidade de o *checkpoint* ser utilizado futuramente.

Os experimentos realizados mostram que a abordagem desenvolvida, ao ajustar os intervalos de criação de *checkpoints* no decorrer do tempo de simulação, pode contribuir com o aumento da performance da simulação. Reduzindo o número de *checkpoints* criados, a quantidade de *rollbacks* e de *checkpoints* inúteis, se comparado a duas soluções anteriores existentes no DCB para criação de *checkpoints*, contribui para a economia de tempo de simulação, recursos de memória e de processamento.

2 REFERENCIAL TEÓRICO

2.1 SIMULAÇÃO

A simulação computacional busca representar o comportamento de sistemas através de modelos que imitam suas características. É uma ferramenta de apoio para entender melhor como o sistema correspondente se comporta ou prever o desempenho sob algumas novas condições consideradas.

A utilização da simulação pode ser uma ferramenta útil e poderosa em diversos tipos de problemas, especialmente naqueles perigosos ou de alto custo, onde testes em situações reais seriam impraticáveis.

Os modelos normalmente são classificados em dois tipos: discretos e contínuos. Um sistema discreto é aquele onde os estados da simulação mudam em pontos separados no tempo. Em um sistema contínuo os estados mudam continuamente em relação ao tempo.

Quanto maior o número de detalhes incorporados ao modelo, melhor será a sua representação do sistema real. No entanto, o modelo pode tornar-se grande e complexo, aumentando o volume de dados e o tempo para o processamento da simulação.

2.2 SIMULAÇÃO DISTRIBUÍDA

Na simulação centralizada um modelo é inteiramente executado em um único recurso de processamento. Com a utilização de elementos distribuídos, conectados através de uma rede de comunicação, um modelo é dividido em partes cooperantes que podem estar em um mesmo local ou geograficamente dispersos.

Segundo (FUJIMOTO, R. M., 1999), a distribuição do modelo em múltiplas partes proporciona alguns benefícios, tais como:

- Redução do tempo de execução;
- A distribuição geográfica permite a inclusão de participantes de diferentes localidades mais facilmente;
- Tolerância à falhas;
- Inclusão de componentes heterogêneos, com diferentes sistemas operacionais, de diferentes fabricantes, com diferentes capacidades de processamento.

Cada processo mantém um conjunto de variáveis locais que constituem o seu estado, que pode ser alterado com a execução de eventos. 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 entre os processos. Todos os outros eventos de um processo são considerados internos (FUJIMOTO, R., 2015).

Os componentes de um sistema distribuído se comunicam e coordenam suas ações através da troca de mensagens. Uma mensagem é composta pelas informações do evento que deve ser realizado e o tempo em que este evento deve ocorrer no processo de destino. O tempo de evento adicionado nas mensagens é chamado também de *timestamp*.

2.2.1 Sincronização

O gerenciamento de tempo é responsável pela sincronização da simulação, garantindo que os eventos ocorram na ordem correta, onde todos os eventos são processados um após o outro na ordem definida pelo *timestamp* (FUJIMOTO, R., 2015).

A ordem de execução de eventos internos de um processo utiliza como referência um tempo local de simulação, ou *Local Virtual Time* (LVT). Já a ordem de execução de eventos externos precisa ser controlada por um tempo global reconhecido em todos os processos do modelo, chamado de *Global Virtual Time* (GVT).

Uma simulação distribuída pode ser classificada em síncrona ou assíncrona, de acordo com o que é realizado para a sincronização dos processos e a execução dos eventos que compõem o modelo simulado. Na simulação síncrona, somente eventos que são considerados seguros são executados. Na abordagem assíncrona, os eventos são executados sem verificar se são seguros, no entanto, mecanismos de recuperação são acionados ao ser identificado que a simulação foi executada fora de ordem.

2.2.1.1 Simulação Síncrona

Uma simulação síncrona ou conservadora somente executa um evento quando tiver a garantia de que nenhum outro evento com *timestamp* menor do que o seu LVT vai ser solicitado, não permitindo, desta forma, que ocorram violações de tempo (FUJIMOTO, R., 2015).

Os processos com LVT maior permanecem bloqueados aguardando os processos com menor LVT avançarem no tempo. Essa situação de espera pode levar a ocorrência de *deadlock* e a perda de eficiência.

Uma das formas de evitar o *deadlock* é a geração de mensagens nulas. Uma mensagem nula não corresponde a uma atividade a ser realizada no sistema simulado, servindo apenas como uma promessa do processo A de que não enviará uma mensagem ao processo B com um registro de *timestamp* menor do que o tempo da mensagem nula.

O *timestamp* da mensagem nula é gerado a partir do *lookahead*. Se o processo A está no momento da simulação T, e pode garantir que qualquer mensagem que ele enviará no futuro terá um *timestamp* de pelo menos $T + L$, diz-se que o processo A tem um *lookahead* de L. Assim, a simulação síncrona ganha mais desempenho, visto que os outros processos podem avançar sua computação até o limite desse intervalo de tempo (ELNOZAHY et al., 2002).

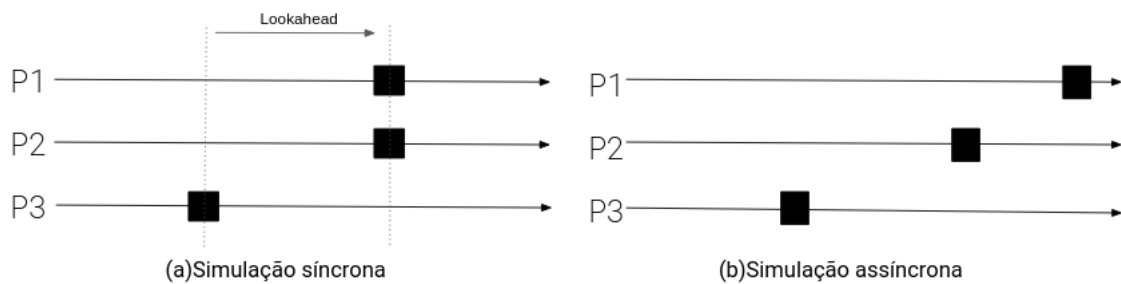


Figura 2 – Exemplo de simulação de componentes síncronos e assíncronos

Há também protocolos que permitem que o *deadlock* ocorra, porém dispõem de mecanismos de detecção e recuperação do estado de *deadlock* (FUJIMOTO, R. M., 1999).

Na Figura 2.a vemos que os processos P1 e P2 avançaram no tempo de simulação até a distância máxima permitida pelo *lookahed* enviado pelo processo P3. P1 e P2 estão bloqueados, ou seja, não estão executando nenhum evento, apenas aguardando que o processo P3 avance no tempo de simulação.

2.2.1.2 Simulação Assíncrona

Na Figura 2.b vemos um exemplo com três processos executando no modo assíncrono. Cada processo avança no tempo de simulação de maneira distinta. Essa variação pode depender da capacidade de processamento do processo ou do tipo de evento que está executando, por exemplo.

A simulação assíncrona ou otimista permite que violações de tempo ocorram, ou seja, que um componente solicite a outro a execução de um evento com *timestamp* menor do que o LVT do componente de destino (FUJIMOTO, R., 2015).

Quando um processo recebe uma mensagem com *timestamp* menor do que o seu LVT, ocorre uma violação de tempo ou LCC (*Local Causality Constraint*). Com isso, o estado da simulação torna-se inconsistente, pois os eventos foram executados fora da ordem definida pelo *timestamp*.

Com a realização de um *rollback*, o processo com erro retorna para um estado consistente anterior ao tempo de chegada desta mensagem e retoma a simulação.

2.2.2 Rollback

O *rollback* é o ato de retroceder no tempo de simulação. Isto ocorre quando há necessidade de restabelecer a consistência da simulação devido a alguma violação de tempo.

As mensagens enviadas entre os processos dificultam a recuperação pois induzem dependências entre eles. Em caso de falha de um ou mais processos em um sistema, essas dependências

podem forçar alguns dos processos que não falharam a também realizarem a reversão, como pode ser observado na Figura 3.

O processo P_2 envia a mensagem m_6 para o processo P_1 e logo em seguida ocorre uma falha no processo P_2 , obrigando-o a retornar para o ponto C, com tempo anterior ao envio da mensagem m_6 . O processo P_1 agora torna-se inconsistente, pois há a ocorrência de uma mensagem órfã, ou seja, com registro de recebimento da mensagem m_6 pelo destinatário (P_1) e o não envio da mesma pelo remetente (P_2).

Apesar de não ter apresentado falha, P_1 também deverá realizar *rollback* para um estado anterior ao recebimento da mensagem m_6 a fim de manter a simulação em um estado consistente, sem a ocorrência de mensagens órfãs. Para que isso ocorra, uma anti-mensagem é enviada para P_1 . Uma anti-mensagem indica que o remetente de uma mensagem, neste exemplo a m_6 , realizou *rollback* para um tempo anterior ao envio da mesma, e solicita que o processo de destino da mensagem também retroceda no tempo para um estado anterior ao seu recebimento.

P_1 também enviará uma anti-mensagem para P_0 , visto que retrocedeu para o ponto B com tempo anterior ao envio da mensagem m_7 , tornando o processo P_0 inconsistente. Este processo se repete até que a simulação volte para um estado consistente.

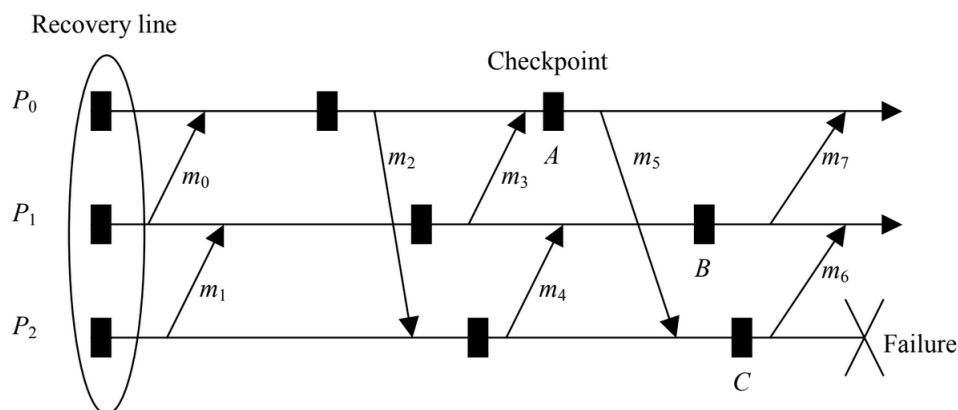


Figura 3 – Efeito Dominó (ELNOZAHY et al., 2002)

O efeito dominó ocorre em cenários em que a propagação de reversão se estende de volta ao estado inicial da simulação, perdendo todo o trabalho realizado antes de uma falha.

Em caso de falha em uma simulação assíncrona, o processo com falha usa as informações salvas em *checkpoints* para reiniciar a computação a partir de um estado intermediário, reduzindo assim a quantidade de computação perdida (ELNOZAHY et al., 2002).

2.2.3 Checkpoints

Checkpoint é uma cópia do estado atual de um processo, armazenado em uma mídia estável, contendo informações suficientes que permitam restaurar o estado do processo em um instante de tempo da simulação em caso de ocorrência de alguma falha.

A Figura 4 mostra um exemplo de utilização de *checkpoints* na recuperação de um estado consistente. Nela podemos observar que o PL0 envia uma mensagem com *timestamp* 40 para o PL1 cujo LVT é 60, causando então uma violação de tempo e obrigando PL1 a realizar *rollback*. O *checkpoint* C1 foi gerado em tempo anterior ao da mensagem recebida e é então utilizado para restaurar a consistência da simulação.

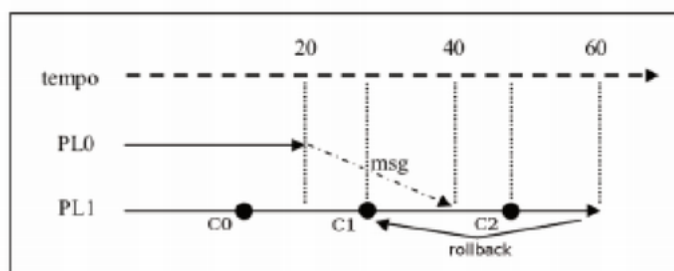


Figura 4 – Exemplo de utilização de um *checkpoint* (MELLO, 2005)

Também observando a Figura 4 podemos encontrar a ocorrência de um *checkpoint* inútil. Um *checkpoint* se torna inútil quando ele não é utilizado ou não é suficiente para restaurar a simulação em uma operação de *rollback*. No exemplo de *rollback* apresentado, o *checkpoint* C2 é inútil, visto que o processo P1 não pode utilizá-lo para restaurar a consistência da simulação. Checkpoints inúteis são indesejáveis, pois não contribuem para a recuperação do sistema em caso de falha, impactando em desperdício de processamento e armazenamento.

Os *checkpoints* são gerados regularmente durante a simulação segundo três abordagens principais: coordenados, não coordenados e baseados em comunicação (ELNOZAHY et al., 2002).

2.2.3.1 Checkpoints coordenados

A abordagem coordenada ou síncrona suspende a operação da aplicação enquanto os processos obtêm um *checkpoint* global, garantindo que os *checkpoints* locais obtidos sejam concorrentes e o *checkpoint* global resultante seja consistente.

A retomada da computação é simplificada em caso de *rollback*, pois a recuperação sempre é reiniciada com a utilização do *checkpoint* mais recente, sem custo adicional dos processos da aplicação para encontrar um *checkpoint* consistente para retomada da computação. Como desvantagem, pode-se citar o fato de interromper a execução de toda a aplicação enquanto coordena a criação de um *checkpoint* por meio de mensagens específicas (ELNOZAHY et al., 2002).

2.2.3.2 Checkpoints não coordenados

Esta abordagem permite total liberdade para cada processo na escolha de seus *checkpoints*, onde cada processo pode criar um *checkpoint* quando for mais conveniente, por exemplo, quando a quantidade de informações do estado a ser salva é pequena.

Ao não impor restrições sobre quais estados são usados como *checkpoints*, esta abordagem traz alguns prejuízos, segundo (ELNOZAHY et al., 2002):

- Está sujeita à ocorrência de *checkpoints* inúteis;
- Ocorrência do efeito dominó;
- Cada processo precisa manter vários *checkpoints*.

2.2.3.3 Checkpoints baseados em comunicação

Dois tipos de *checkpoints* são gerados com a utilização desta técnica: locais e forçados. Os *checkpoints* locais são gerados livremente pelos processos, assim como acontece na abordagem não coordenada. Os *checkpoints* forçados são gerados a partir do processamento de informações de controle contidas nas mensagens enviadas entre os processos, a fim de identificar relações de dependência entre eles.

Não há troca de mensagens de controle específicas para determinar quando um *checkpoint* forçado deve ser criado, o que reduz o *overhead* de comunicação. Além disso, também evita a criação de *checkpoints* inúteis e a não ocorrência do efeito dominó.

2.2.3.4 Garbage Collection

No decorrer do tempo de simulação, alguns *checkpoints* criados podem se tornar obsoletos, ou seja, que não serão mais úteis em uma operação de *rollback*. No entanto, estes *checkpoints* continuam consumindo espaço de armazenamento. De tempos em tempos, algoritmos de *Garbage Collection* ou *Fossil Collection* devem ser executados para que estes *checkpoints* obsoletos sejam identificados e eliminados, liberando assim espaço de armazenamento para novos *checkpoints* serem criados.

2.3 DISTRIBUTED CO-SIMULATION BACKBONE

2.3.1 Visão Geral

O Distributed Co-Simulation Backbone ou DCB, é uma arquitetura de simulação que permite executar modelos de simulação distribuídos e heterogêneos.

Como pode ser observado na Figura 5, o DCB é composto por quatro módulos principais: o *gateway*, o Expedidor do DCB (*DCB Sender* - DCBS), o Receptor do DCB (*DCB Receiver* - DCBR) e o Núcleo do DCB (*DCB Kernel* - DCBK) (MELLO, 2005).

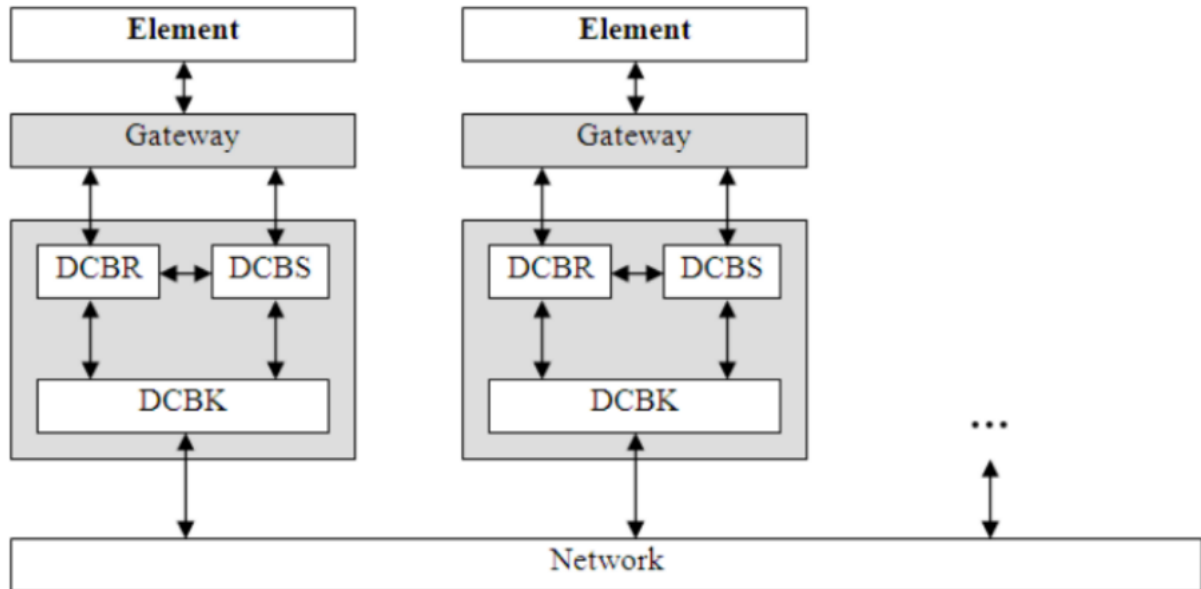


Figura 5 – Arquitetura do DCB (MELLO, 2005)

Por permitir que os elementos possam ser heterogêneos, se faz necessário a presença de uma interface de comunicação responsável por realizar a ligação entre o elemento e os demais componentes que dão suporte à simulação. Esta tarefa é realizada pelo *gateway*.

O DCBK provê os serviços de comunicação por troca de mensagens e realiza o controle para que todos os nodos possuam o mesmo tempo virtual global.

O gerenciamento das mensagens recebidas de outros elementos e enviadas para outros elementos da simulação são gerenciadas pelo DCBR e pelo DCBS, respectivamente. Ambos contribuem também com o gerenciamento do tempo da simulação.

A sincronização da simulação é realizada por meio do *timestamp* que é acoplada nas mensagens enviadas, indicando para o processo de destino o tempo em que o evento deve ser executado.

O DCB permite que os componentes avancem no tempo de modo síncrono ou assíncrono, possibilitando a criação de modelos híbridos, ou seja, com a participação de componentes síncronos e assíncronos. É permitido ainda a inclusão de elementos *untimed*, que não utilizam tempo. Estes componentes executam suas tarefas de acordo com a ordem em que foram recebidas. Elementos síncronos são conservadores e bloqueiam sua execução enquanto existirem eventos previstos para o seu tempo. Componentes assíncronos organizam seus eventos com ajuda de mecanismos de *rollback* baseados em *checkpoint*.

2.3.2 Criação de Checkpoints

Ao iniciar a simulação, todos os elementos estabelecem o primeiro *checkpoint* com LVT 0. Isso garante que caso ocorra o efeito dominó a simulação não precise ser interrompida.

Na solução desenvolvida em (CARVALHO; MELLO, 2015) os *checkpoints* são criados de forma não coordenada, levando em consideração apenas o tempo de simulação e o número de troca de mensagens entre os elementos.

Durante a simulação, novos *checkpoints* são criados se uma destas duas regras for atendida:

1. A simulação avançou 5.000 unidades de tempo desde o último *checkpoint*;
2. Desde o último *checkpoint* o número de mensagens enviadas e recebidas for maior que 10.

Com base na abordagem de criação de *checkpoints* induzidos a comunicação, (BIZZANI, 2016) incorporou ao DCB um novo método a partir da identificação de estados seguros para criar *checkpoints*. Para tanto, faz uso de um histórico de mensagens recebidas de cada elemento e de dados anexados em cada troca de mensagem entre os componentes.

A abordagem de predição de mensagens detecta instantes de tempo que são seguros para a criação de um *checkpoint*. Para isso, considera a frequência com que um elemento recebe mensagens de outros elementos, armazenando em uma fila o *timestamp* das últimas cinco mensagens recebidas de cada elemento que se comunica com ele.

Ao receber uma mensagem, o elemento calculará a média de intervalos de recebimento de mensagens deste remetente. Essa média é somada ao LVT atual para gerar uma nova predição. Quando o processo atingir esse LVT o *checkpoint* é criado.

A abordagem de índice de *checkpoint* atua na identificação de padrões no envio e recebimento de mensagens, impedindo que levem a estados inconsistentes criando *checkpoints*.

Cada elemento mantém um contador de *checkpoints*. O contador é incrementado em um a cada *checkpoint* criado e decrementado em um a cada *checkpoint* inútil encontrado durante a operação de *rollback*. Ao mandar uma mensagem esse contador é adicionado no final da mensagem. Ao receber uma mensagem, caso o índice da mensagem for maior do que o índice local, um novo *checkpoint* é gerado e o índice local é atualizado com o índice da mensagem.

3 TRABALHOS RELACIONADOS

Este capítulo apresenta uma visão geral de estudos relacionados aos objetivos deste trabalho. As soluções apresentadas buscam aumentar a performance da simulação buscando identificar os melhores momentos ou a frequência em que os *checkpoints* são criados por cada processo.

Em (FIALHO; REXACHS; LUQUE, 2012) é apresentado uma equação para calcular em tempo de execução o intervalo considerado ideal para criação de *checkpoints*. O estudo é conduzido considerando uma aplicação paralela com a criação de *checkpoints* ocorrendo de forma não coordenada por cada processo.

O método busca definir a frequência em que os *checkpoints* devem ser criados de modo a minimizar o custo dos processos com a criação de *checkpoints*. O cálculo do intervalo entre *checkpoints* considera dois pontos principais: o custo para criação de *checkpoints* e a relação de dependência entre os processos. O modelo possibilita que cada processo crie *checkpoints* em intervalos distintos.

Em (QUAGLIA, 2001) é apresentada uma técnica de criação de *checkpoints* que seleciona o momento em que será criado um *checkpoint* com base em um modelo que considera o custo de recuperar um determinado estado da simulação em caso de *rollback*.

Dado o estado atual da simulação, o modelo determina a conveniência de criar um *checkpoint* neste estado analisando a posição do último *checkpoint*, o tempo de execução dos eventos executados desde o último *checkpoint* e uma estimativa probabilística de que o *checkpoint* criado nesse estado da simulação possa ser utilizado futuramente.

O *checkpoint* somente é criado caso o custo de salvar o *checkpoint* for menor ou igual ao custo de não salvar, ou seja, carregar o *checkpoint* anterior ao estado atual e reprocessar os eventos executados desde então. Adicionalmente, o algoritmo possui a criação de *checkpoints* periódicos, devido ao fato de que a técnica permite a possibilidade de induzir um processo a nunca registrar um *checkpoint* em casos em que a probabilidade se aproximar de zero.

Em (RÖNNGREN; AYANI, 1994) é apresentado um estudo sobre os efeitos de salvar *checkpoints* com menor frequência na performance da simulação. É apresentado também um modelo que permite que cada processo adapte continuamente o intervalo dos seus *checkpoints* de acordo com o comportamento dos *rollbacks*.

O intervalo entre os *checkpoints* é calculado considerando dados coletados durante um intervalo de tempo, tais como: número de eventos executados, número de *rollbacks*, tempo para salvar e tempo para carregar um *checkpoint*, entre outros. Há também a presença de um fator que determina o quão rápido o intervalo de *checkpoint* se adaptará às mudanças nas frequências dos *rollbacks*.

4 CRIAÇÃO DE CHECKPOINTS BASEADO NO CUSTO DE RECUPERAÇÃO

Este capítulo apresenta uma estratégia para a criação de *checkpoints* que considera o custo de recuperação para um determinado estado da simulação. O algoritmo implementado foi adaptado de (QUAGLIA, 2001). São apresentadas a descrição do algoritmo, a sua implementação no DCB, o modelo utilizado para estudo de caso e por fim os resultados obtidos. Para a análise dos resultados, foi realizado um comparativo com duas soluções anteriores existentes no DCB para a criação de *checkpoints*.

4.1 ESPECIFICAÇÃO DA SOLUÇÃO

Um processo se move de um estado para outro por meio da execução de eventos. Na Figura 6 podemos observar que o processo passa do estado X para o estado Y com a execução do evento e_1 , e do estado Y para o estado S com a execução do evento e_2 .

O estado S representa o estado atual da simulação. Para cada novo estado é atribuído uma probabilidade, chamado P(S), de que este estado possa ser restaurado futuramente por uma operação de *rollback*. P(S) é calculado observando a quantidade de *rollbacks* que ocorreram e o total de eventos executados desde o início da simulação ($P(S) = \text{rollbacks} / \text{eventos}$).

Após o cálculo da probabilidade de utilização de um *checkpoint* no estado S, o método calcula o custo para gerar um *checkpoint* no estado S e o custo para recuperação do estado S em caso de *rollback*.

O cálculo do custo para que um *checkpoint* seja criado no estado S é realizado da seguinte forma:

$$C(S) = \begin{cases} \delta_s & \text{se S for gravado} \\ 0 & \text{se S não for gravado} \end{cases}$$

$C(S)$, que representa o custo para salvar o *checkpoint* no estado S, depende da quantidade de informação a ser armazenada (número de bytes) e o tempo por byte necessário pelo processo para armazenar esta quantidade de informações. Este tempo total para armazenamento é representado por δ_s . Se não for criado *checkpoint* no estado S, então $C(S) = 0$.

Já o custo de recuperação do processo para o estado S é:

$$R(S) = \begin{cases} P(S)\delta_s & \text{se S for gravado} \\ P(S)[\delta_s + \sum_{e \in E(S)} \delta_e] & \text{se S não for gravado} \end{cases}$$

Ao ser criado um *checkpoint* no estado S, o custo para recuperar o estado S consiste no custo de recarregar o *checkpoint* gerado. Este custo é também representado por δ_s . Neste cenário, o custo para salvar e o custo para restaurar um *checkpoint* é o mesmo. Se não for criado um *checkpoint* no estado S, o custo de recuperação consiste em recarregar o *checkpoint* mais

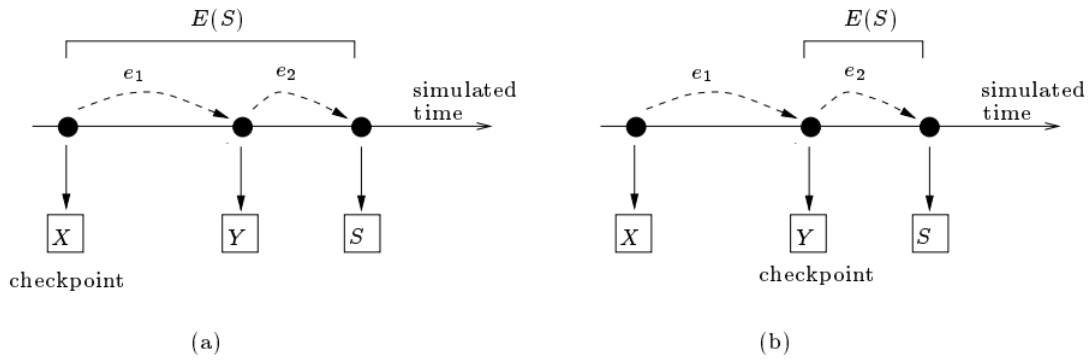


Figura 6 – Exemplo custo de recuperação
(QUAGLIA, 2001)

recente anterior a S mais o custo para reproduzir todos os eventos até a simulação chegar em S novamente.

A Figura 6 apresenta um exemplo do custo de recuperação de um estado da simulação quando não for criado um *checkpoint* no estado S. A Figura 6.a mostra que o último *checkpoint* antes de S é X. Com isso, o custo de recuperação para o estado S consiste na restauração do *checkpoint* X e no reprocessamento dos eventos e_1 e e_2 . Já na Figura 6.b o último *checkpoint* antes de S é Y. Neste caso, o custo de recuperação para o estado S é restaurar o *checkpoint* Y e reprocessar o evento e_2 .

O valor de $P(S)$, que é a probabilidade de o estado S ser recuperado futuramente, é o mesmo utilizado tanto para calcular o custo de recuperação se for criado ou não um *checkpoint* no estado S.

O conjunto $E(S)$ contém os eventos executados desde o último *checkpoint* até o estado S. Na Figura 6.a $E(S) = \{e_1, e_2\}$ e na Figura 6.b $E(S) = \{e_2\}$. δ_e determina o custo necessário para execução de um evento. Este custo é determinado pelo tipo de evento ou tarefa que é executado pelo processo.

Dependendo da tarefa ou tipo de evento a ser executado ela pode demorar mais ou menos tempo, considerando a capacidade de processamento de um mesmo processo. Para cada tipo de evento que possa ser realizado por um processo haverá um δ_e correspondente. Um grande número de eventos com δ_e baixo, ou seja, de execução rápida, pode ter um custo de reprocessamento menor do que um pequeno número de eventos com δ_e alto, onde o tempo de reprocessamento será maior.

Por fim, o custo para decidir se um *checkpoint* será ou não criado em S se dá pela soma de $C(S)$ e $R(S)$, resultando em:

$$CR(S) = \begin{cases} \delta_s + P(S)\delta_s & \text{se S for gravado} \\ P(S)[\delta_s + \sum_{e \in E(S)} \delta_e] & \text{se S não for gravado} \end{cases}$$

O *checkpoint* somente é criado no estado S caso o custo de salvar o *checkpoint* seja menor ou igual ao custo de não salvar, ou seja, restaurar o *checkpoint* anterior ao estado S e

reprocessar os eventos contidos em $E(S)$ até chegar em S novamente.

Adicionalmente, o algoritmo possui a criação de *checkpoints* periódicos, devido ao fato de que a técnica permite a possibilidade de induzir um processo a nunca registrar um *checkpoint* em casos em que a probabilidade $P(S)$ se aproximar de zero.

Se poucos *checkpoints* forem criados durante a simulação, então é possível que o número de mensagens que devem ser armazenadas seja muito grande. Em consequência disso, a quantidade de memória utilizada que é recuperada durante a execução do *Garbage Collection* pode ser pequena, fazendo com que este procedimento de coleta de *checkpoints* antigos seja invocado com mais frequência, podendo prejudicar o desempenho da simulação. Para resolver este problema, a criação de *checkpoints* periódicos ocorre em um intervalo entre 15 e 30 eventos executados entre dois *checkpoints*.

No Algoritmo 1, $CR(S)y$ representa o custo de salvar o *checkpoint* no estado S e $CR(S)n$ o custo de não salvar. *event_ex* armazena o número de eventos executados desde o último *checkpoint* e *maxdist* é o número máximo de eventos que pode haver entre um *checkpoint* e outro (valor entre 15 e 30).

Então, o método de criação de *checkpoints* apresentado cria um *checkpoint* no estado S , se o custo para salvar o *checkpoint* for menor ou igual ao custo de não salvar, ou se o número máximo de eventos desde o último *checkpoint* for atingido.

Algoritmo 1 – Custo de recuperação

```

1 if ( $CR(S)y \leq CR(S)n$ )OR(event_ex=maxdist) then
2   |   criar checkpoint;
3 else
4   |   não criar checkpoint;
5 end
6

```

4.2 IMPLEMENTAÇÃO

Para o desenvolvimento do trabalho, foi utilizado a IDE IntelliJ IDEA 2021.1 (Community Edition) para programação na linguagem Java. A linguagem Java foi escolhida por ser a mesma já utilizada na implementação do DCB.

Para a implementação do Algoritmo 1 foi criada uma nova classe chamada: *RecoveryCostCheckpoints*. Essa classe estende da classe *Component* e fornece acesso aos módulos do DCB que gerenciam as mensagens recebidas e enviadas e o tempo de simulação (DCBR, DCBS e DCBK respectivamente).

Ao receber uma mensagem, o processo executa o método *onMessage* da classe *RecoveryCostCheckpoints*, recebendo como parâmetro a mensagem recebida. Dentro desse método está incluída a chamada para o método *shouldTakeCheckpoint*. Este método é responsável por identificar se deverá ou não ser criado um *checkpoint*.

A classe *RollbackManager* é responsável por criar os *checkpoints*, controlar o LVT do processo e realizar os *rollbacks*. Ela necessitou de alterações para fornecer informações necessárias para o cálculo do algoritmo e avaliação do método. Nela foram adicionados:

- Lista para armazenar os *timestamps* em que ocorreram *rollbacks*;
- Método *getEventExec* para retornar a quantidade de mensagens enviadas e recebidas desde o início da simulação;
- Método *getEventExecSinceLastCheckpoint* para retornar a quantidade de mensagens enviadas e recebidas desde o último *checkpoint*;
- Método *getRollbacks* para retornar a quantidade de *rollbacks* ocorridos desde o início da simulação;
- Atributo para armazenar a quantidade de *checkpoints* inúteis;
- Atributo para armazenar a quantidade de mensagens reprocessadas.

O método *shouldTakeCheckpoint* da classe *RecuperationCostCheckpoints* irá executar os métodos *getEventExec* e *getRollbacks* da classe *RollbackManager* para obter a quantidade de eventos executados e a quantidade de *rollbacks*. Com isso, tem as informações necessárias para calcular a probabilidade $P(S)$.

Em seguida o método *shouldTakeCheckpoint* irá executar o método *getEventExecSinceLastCheckpoint* da classe *RollbackManager* para obter a quantidade de eventos executados desde o último *checkpoint*. Calcula então $C(S)$ e $R(S)$, obtendo os valores do custo de salvar um *checkpoint* e o de recuperar o estado atual. Por fim, o método testa se o custo de salvar o *checkpoint* é menor ou igual ao custo de não salvar, ou se a quantidade de eventos executados desde o último *checkpoint* for igual ao máximo permitido entre dois *checkpoints*. O método irá retornar *true* se uma das condições for atendida ou *false* caso contrário.

O método *onMessage* então irá verificar o retorno da chamada de *shouldTakeCheckpoint*. Caso o retorno for *true* irá chamar o método *takeCheckpoint* da classe *RollbackManager* para que seja criado um *checkpoint*. Se o retorno for *false*, o *checkpoint* não é gerado.

Na classe *RecuperationCostCheckpoints* também são definidos os parâmetros δ_s , δ_e e *maxdist* que são valorados ao instanciar a classe e não sofrem alteração até finalizar a simulação. Para os testes foram utilizados os seguintes valores:

- Número máximo de eventos entre dois *checkpoints* (*maxdist*) = 30;
- Tempo para salvar/recuperar um *checkpoint* (δ_s) = 70 microssegundos;
- Tempo de execução por evento (δ_e) = 140 microssegundos.

Os valores de δ_s e δ_e foram definidos com base na primeira configuração para teste realizado por (QUAGLIA, 2001). Esta configuração permite testar as técnicas de *checkpoint* considerando que não há nenhuma variação sobre o tipo e o tempo de execução de um evento. Ou seja, assume-se que a execução de um evento terá sempre a mesma duração em todos os processos. Da mesma forma, considera-se que a criação de *checkpoints* sempre terá a mesma quantidade de informação a ser armazenada e levará a mesma quantidade de tempo para ser armazenado em todos os processos participantes da simulação.

4.3 ESTUDO DE CASO

A fim de comparar os métodos de criação de *checkpoints*, foi desenvolvido um modelo de comunicação em que três componentes assíncronos realizam troca de mensagens, sendo eles aqui chamados de: A, B e C.

O processo A envia mensagens para os processos B e C. O processo B recebe mensagens do processo A e envia mensagens para o processo C. O processo C apenas recebe mensagens dos processos A e B.

A Figura 7 mostra um grafo com a relação da troca de mensagens entre os processos. As arestas são dirigidas, indicando a direção das mensagens enviadas por cada processo.

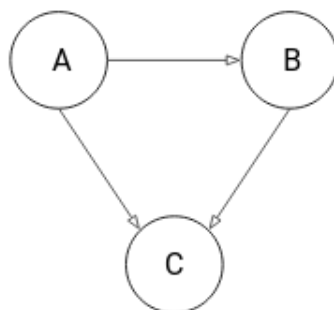


Figura 7 – Relação da troca de mensagens entre os processos

Os processos executam até que o seu LVT atinja a marca de 100.000. O LVT de cada processo inicia em zero e a cada mensagem enviada o LVT é incrementado de forma randômica.

O processo A ao enviar uma mensagem incrementa seu LVT em um intervalo que varia de 50 até 200. O processo B ao enviar uma mensagem incrementa seu LVT em um intervalo que varia de 10 até 40. Essa diferença no aumento do LVT de cada processo é motivada pela necessidade de forçar que hajam algumas ocorrências de violação de tempo no processo C, resultando em operações de *rollback*.

As simulações foram executadas em um único computador: Intel® Core™ i5-2430M CPU @ 2.40GHz × 4, com 8GB de memória RAM e sistema operacional Ubuntu 18.04.5 LTS.

Foram executadas 3 simulações até que o LVT de cada componente atingisse o valor de 100.000 para cada uma destas três técnicas de criação de *checkpoints*:

- Criação de *checkpoints* periódicos (CARVALHO; MELLO, 2015);
- Criação de *checkpoints* em estados seguros (BIZZANI, 2016);
- Criação de *checkpoints* com base no custo de recuperação (implementado neste trabalho).

4.4 ANÁLISE DOS RESULTADOS

Primeiramente vamos analisar o comportamento dos algoritmos na geração de *checkpoints* observando a linha do tempo da simulação do processo C.

Nas Figuras 8, 9 e 10 as linhas azuis representam a distância entre os *checkpoints* gerados. Essa distância é a diferença entre o *timestamp* de um *checkpoint* e o *timestamp* do último *checkpoint* criado antes dele. As linhas na cor laranja representam a quantidade de mensagens enviadas e recebidas entre dois *checkpoints* criados. Já as linhas na cor verde representam a quantidade de *rollbacks* ocorridos no decorrer do tempo de simulação.

Na Figura 8 podemos observar o comportamento da geração de *checkpoints* pela abordagem periódica. Há uma grande variação entre as distâncias entre um *checkpoint* e outro, motivada pelo fato de o aumento do LVT das mensagens não ser sempre o mesmo. No entanto, a quantidade de mensagens entre um *checkpoint* e outro se mantém constante em 10, ou seja, está de acordo com a proposta do algoritmo de criar um *checkpoint* a cada 10 mensagens enviadas e/ou recebidas.

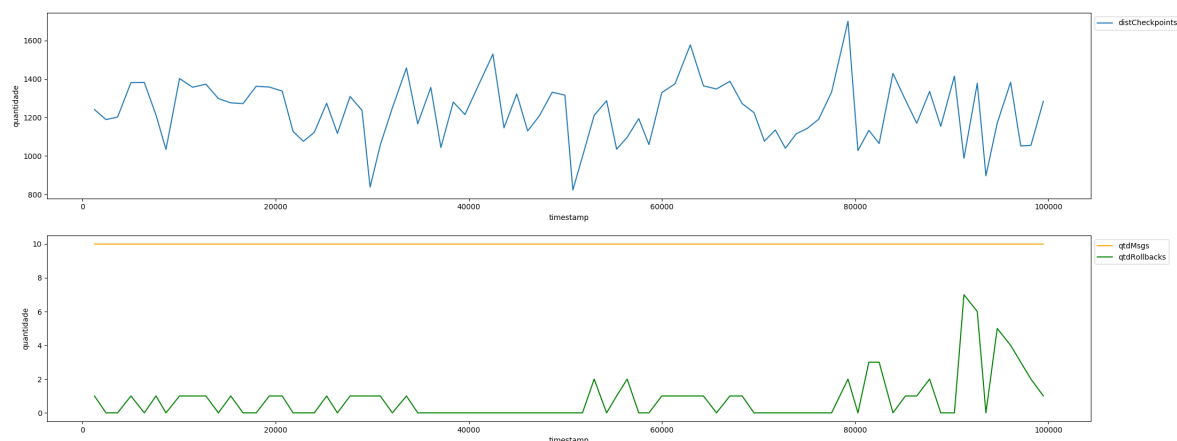


Figura 8 – Linha do tempo do processo C - Checkpoints periódicos

A Figura 9 apresenta a geração de *checkpoints* pelo método de identificação de estados seguros. Aqui ocorre uma variação menor das distâncias entre os *checkpoints* motivadas pelo fato de o algoritmo gerar um maior número deles. A cada estado seguro encontrado a abordagem cria um novo *checkpoint*. A quantidade de mensagens entre eles também diminuiu em relação a

criação de *checkpoints* periódicos. A quantidade de *rollbacks* ocorridos, no entanto, aumentou, devido a geração de um maior número de *checkpoints* inúteis.

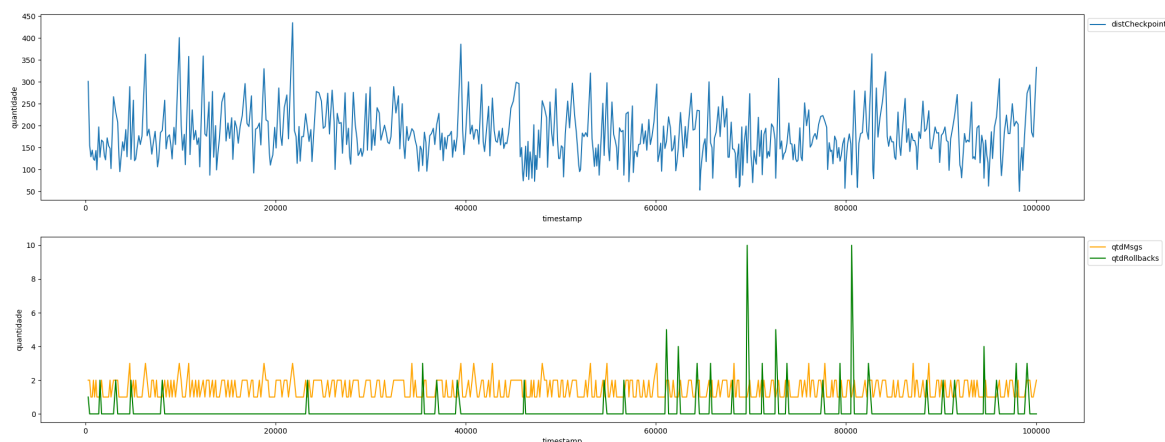


Figura 9 – Linha do tempo do processo C - Estados seguros

Na Figura 10 vemos o comportamento da geração de *checkpoints* utilizando a abordagem que calcula o custo de recuperação. O processo apresenta uma variação da distância entre os *checkpoints* e no número de mensagens entre um e outro no decorrer do tempo da simulação. Em alguns momentos há um aumento na distância e o no número de mensagens entre os *checkpoints*. Em outros, esse número é reduzido. Essa variação é devida a ocorrência dos *rollbacks*.

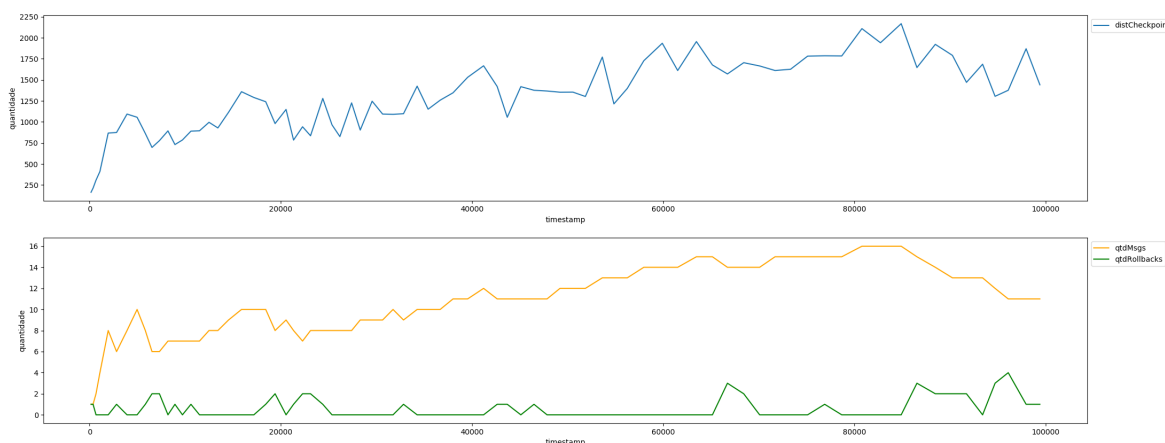


Figura 10 – Linha do tempo do processo C - Custo de recuperação

Se em um intervalo de tempo o número de *rollbacks* aumenta, a abordagem gera *checkpoints* com mais frequência, diminuindo a distância entre os *checkpoints* e o número de mensagens entre eles. Quando o número de *rollbacks* diminui, o algoritmo gera *checkpoints* mais distantes um do outro e com um maior número de mensagens entre eles.

Este comportamento pode ser benéfico para a simulação. Criando um maior número de *checkpoints* em instantes de tempo onde há maiores chances de ocorrer *rollback*, contribui para reduzir o tempo de reprocessamento. Já ao criar *checkpoints* mais distantes em momentos onde a ocorrência de *rollbacks* tem menor probabilidade de ocorrer, permite que a simulação ganhe

mais agilidade, não precisando ser interrompida para criação *checkpoints* que poderão se tornar inúteis.

Durante a execução algumas outras informações são extraídas da simulação, sendo estes valores também utilizados como comparação dos métodos:

- Total de mensagens recebidas e enviadas;
- Total de *checkpoints* gerados;
- Total de *rollbacks* realizados;
- Total de *checkpoints* inúteis;
- Total de mensagens reprocessadas.

Na Tabela 1 e na Figura 11 podemos observar os valores obtidos para cada método para os parâmetros especificados. Estes valores correspondem ao total dos processos A, B e C que estavam em execução.

Algoritmo	Qtd. Msg	Checkpoints	Inúteis	Rollbacks	Reprocessados
Checkpoints periódicos	1.757	182	29	66	441
Estados Seguros	1.772	1230	66	98	92
Custo de recuperação	1.764	112	10	38	247

Tabela 1 – Resultados obtidos

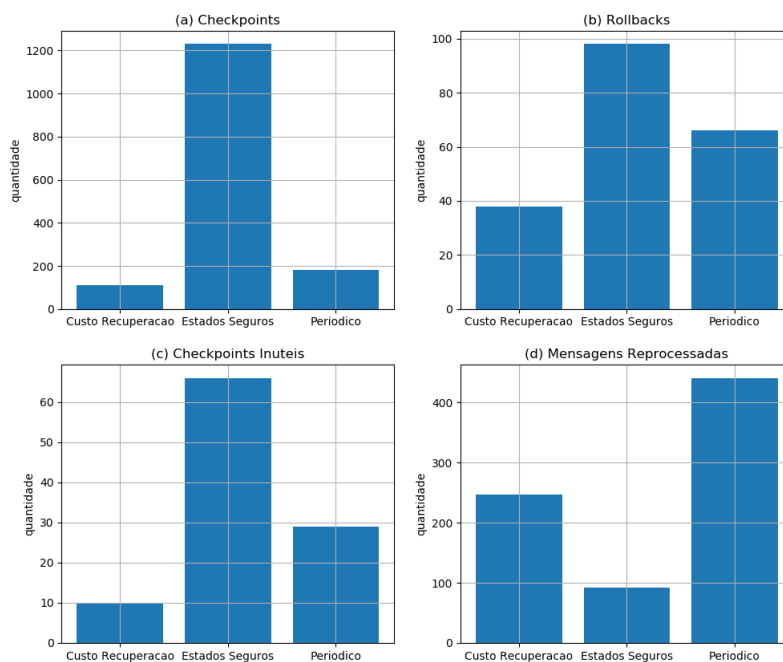


Figura 11 – Comparação dos resultados

A quantidade de mensagens varia levemente entre os métodos. Isso se deve ao fato de que os processos executam até que o seu LVT atinja a marca de 100.000. A cada mensagem

enviada o LVT é incrementado com o uso de uma distribuição uniforme, e dependendo dos valores sorteados, pode haver um maior ou menor número de mensagens.

A quantidade de *checkpoints* gerados pelo algoritmo que considera o custo de recuperação foi o menor dentre todos os algoritmos testados. Isso se deve ao fato de gerar *checkpoints* mais distantes um do outro quando a probabilidade de ocorrência de *rollback* é baixa, conforme já demonstrado na Figura 10.

A quantidade de *rollbacks* realizados teve variação entre os métodos. O aumento do número de *rollbacks* é devido a grande quantidade de *checkpoints* inúteis gerados, especialmente na criação de *checkpoints* em estados seguros.

Em relação ao total de *checkpoints* criados, a técnica implementada neste trabalho gerou 8,93% de *checkpoints* inúteis. Já a abordagem que cria *checkpoints* periódicos criou 15,93%. Ao criar *checkpoints* em estados seguros, 5,37% deles foram considerados inúteis.

O número de mensagens reprocessadas devido a ocorrência de *rollback* foi de 5,19% ao criar *checkpoints* em estados seguros, 14,00% ao considerar o custo de recuperação, e 25,10% ao criar *checkpoints* de forma periódica. Este baixo número de mensagens reprocessadas obtido pelo método de estados seguros se deve ao fato de o algoritmo gerar um maior número de *checkpoints*, e com isso, um menor número de mensagens entre um *checkpoint* e outro, exemplificado na Figura 9.

Um maior número de mensagens reprocessadas pelo método que calcula o custo de recuperação em relação a abordagem de criação de *checkpoints* em estados seguros, leva a crer que o algoritmo considerou que o reprocessamento de alguns intervalos era mais benéfico para a simulação do que criar um maior número de *checkpoints*. Ao contrário do que ocorreu na abordagem de estados seguros, onde há um maior número de *checkpoints* gerados.

Nas Tabelas 2, 3 e 4 estão os tempos de execução, em microssegundos, para cada uma das abordagens testadas. O tempo de simulação foi dividido entre as atividades executadas por cada processo:

- Envio e recebimento de mensagens;
- Criação de *checkpoints*;
- Reprocessamento de mensagens;
- Recuperação de um *checkpoint* em decorrência de *rollback*.

Para o cálculo do tempo de simulação, foram considerados os mesmos parâmetros utilizados na implementação do algoritmo de criação de *checkpoints* que considera o custo de recuperação, onde o tempo para executar ou reprocessar um evento é de 140 microssegundos e o tempo para criar ou restaurar um *checkpoint* é de 70 microssegundos.

O tempo de processamento das mensagens é semelhante entre os modelos testados, variando apenas pela quantidade de mensagens processadas. A técnica implementada reduz o tempo da simulação em tarefas como a criação de *checkpoints*, reprocessamento e *rollbacks*

em relação a criação de *checkpoints* periódicos. Se comparado com a abordagem que utiliza os estados seguros, há redução no tempo gasto na criação de *checkpoints* e nas operações de *rollback*. A diminuição do tempo gasto nestas tarefas, possibilita que a simulação dedique uma maior parte do tempo na execução de eventos (mensagens), fazendo com que o tempo total de simulação de um modelo diminua.

	Quantidade	Tempo por evento	Tempo Total	% Total
Mensagens	1.757	140	245.980	75,67
Checkpoint	182	70	12.740	3,92
Reprocessamento	441	140	61.740	18,99
Rollback	66	70	4.620	1,42
Total			325.080	100

Tabela 2 – Tempo total de execução - Checkpoints periódicos

	Quantidade	Tempo por evento	Tempo Total	% Total
Mensagens	1.772	140	248.080	70,82
Checkpoint	1230	70	86.100	24,58
Reprocessamento	66	140	9.240	2,64
Rollback	98	70	6.860	1,96
Total			350.280	100

Tabela 3 – Tempo total de execução - Estados seguros

	Quantidade	Tempo por evento	Tempo Total	% Total
Mensagens	1.764	140	246.960	84,56
Checkpoint	112	70	7.840	2,69
Reprocessamento	247	140	34.580	11,84
Rollback	38	70	2.660	0,91
Total			292.040	100

Tabela 4 – Tempo total de execução - Custo de recuperação

Na Figura 12, é apresentado um gráfico com os percentuais do tempo total de execução que a simulação dedicou a cada uma destas tarefas, obtido das tabelas acima.

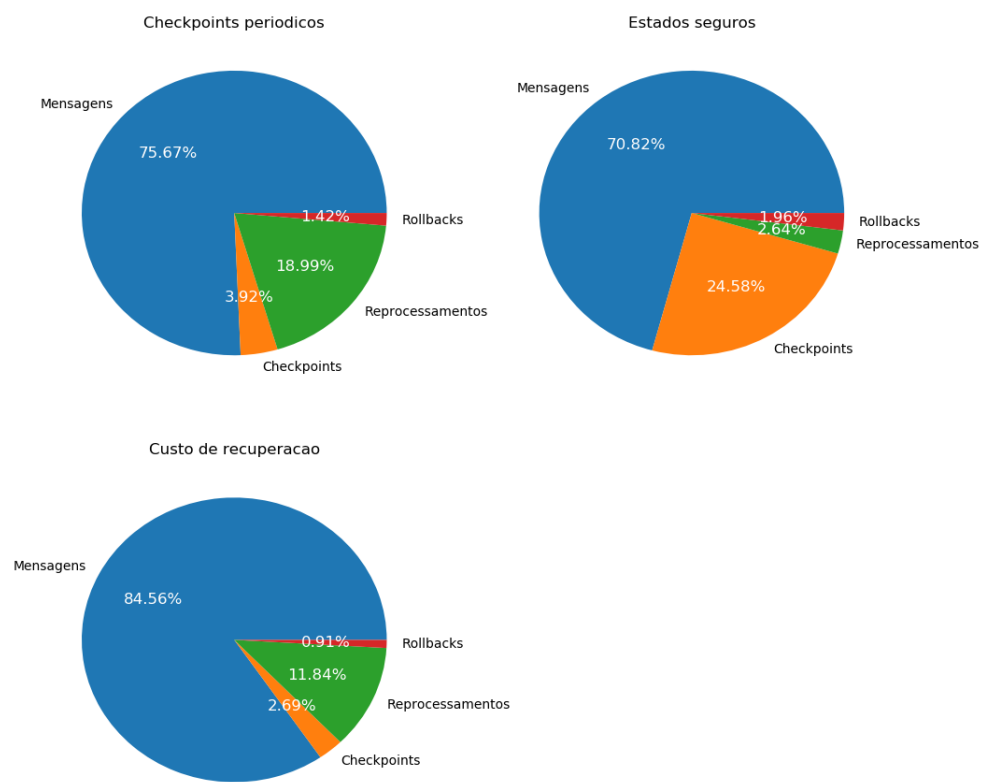


Figura 12 – Tempo total de execução

5 CONCLUSÃO

As contribuições deste trabalho agregam ao DCB funcionalidades para que o processo decida se o estado atual da simulação é conveniente para a criação de um *checkpoint*, analisando o custo de recuperação para o estado em caso de uma possível ocorrência de *rollback*. Os experimentos realizados mostram que, em comparação com duas abordagens anteriores existentes no DCB para criação de *checkpoints*, houve diminuição do número de *checkpoints* gerados, quantidade de *rollbacks* e *checkpoints* inúteis no cenário analisado.

Um maior número de mensagens reprocessadas pelo método, em comparação a criação de *checkpoints* periódicos, sugere que o algoritmo considerou que em algumas situações o reprocessamento de alguns intervalos era mais benéfico para a simulação do que criar um maior número de *checkpoints*.

Levando em conta o histórico de simulação de cada processo, a abordagem pode contribuir com o aumento da performance de simulações, identificando os instantes mais adequados para criar *checkpoints* com base no custo de recuperação do estado, reduzindo o número de *checkpoints* criados, economizando tempo, recursos de memória e processamento para geração e armazenamento dos mesmos.

5.1 TRABALHOS FUTUROS

Estudos futuros podem ser realizados aplicando o método de criação de *checkpoints* baseado no custo de recuperação em uma situação real de simulação, como por exemplo, em um cenário de vacinação em massa da população em uma pandemia.

A aplicação do algoritmo em um ambiente distribuído e com um maior número de processos participantes, variando entre os processos os parâmetros utilizados pelo algoritmo para o tempo de execução de um evento (δ_e), salvamento e recuperação de um *checkpoint* (δ_s) e quantidade máxima de eventos entre *checkpoints* (*maxdist*).

REFERÊNCIAS

- BIZZANI, Guilherme. **Identificação de estados seguros para reduzir a criação de Checkpoints sem valor**. [S.l.: s.n.], 2016.
- CARVALHO, F. M. M.; MELLO, B. A. Hybrid synchronization in the dcb based on uncoordinated checkpoints. Leicester, 2015.
- ELNOZAHY, E. N. (Mootaz) et al. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 34, n. 3, p. 375–408, set. 2002. ISSN 0360-0300.
- FIALHO, Leonardo; REXACHS, Dolores; LUQUE, Emilio. Defining the Checkpoint Interval for Uncoordinated Checkpointing Protocols, abr. 2012.
- FUJIMOTO, Richard. Parallel and Distributed Simulation. In: PROCEEDINGS of the 2015 Winter Simulation Conference. Huntington Beach, California: IEEE Press, 2015. (WSC '15), p. 45–59. ISBN 9781467397414.
- FUJIMOTO, Richard M. PARALLEL AND DISTRIBUTED SIMULATION. In: 1999 Winter Simulation Conference Proceedings: Pointe Hilton Squaw Peak Resort, Phoenix, AZ, USA; 5-8 December 1999. [S.l.: s.n.], 1999. v. 2, p. 122.
- MELLO, Braulio Adriano de. **Co-simulação distribuída de sistemas heterogêneos**. 2005. Tese (Doutorado) – Universidade Federal do Rio Grande do Sul.
- QUAGLIA, F. A cost model for selecting checkpoint positions in time warp parallel simulation. **IEEE Transactions on Parallel and Distributed Systems**, v. 12, n. 4, p. 346–362, 2001.
- RÖNNGREN, Robert; AYANI, Rassul. Adaptive Checkpointing in Time Warp. In: PROCEEDINGS of the Eighth Workshop on Parallel and Distributed Simulation. Edinburgh, Scotland, United Kingdom: Association for Computing Machinery, 1994. (PADS '94), p. 110–117. ISBN 1565550277.