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

JACKSON HENRIQUE HOCHSCHEIDT

INTERVALO FLEXÍVEL PARA CRIAÇÃO DE CHECKPOINTS EM SIMULAÇÃO DISTRIBUÍDA

CHAPECÓ
2019

JACKSON HENRIQUE HOCHSCHEIDT

INTERVALO FLEXÍVEL PARA CRIAÇÃO DE 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
2019

Hochscheidt, Jackson Henrique

Intervalo Flexível para Criação de Checkpoints em Simulação Distribuída / Jackson Henrique Hochscheidt. – 2019.
27 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, 2019.

1. Simulação Distribuída. 2. Checkpoints. 3. Intervalo Flexível de Checkpoint. 4. Rollback. 5. Sobrecarga. I. Mello, Prof. Dr. Braulio Adriano de, orientador. II. Universidade Federal da Fronteira Sul. III. Título.

E-mail: jackson94h@gmail.com

JACKSON HENRIQUE HOCHSCHEIDT

INTERVALO FLEXÍVEL PARA CRIAÇÃO DE 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

Este trabalho de conclusão de curso foi defendido e aprovado pela banca avaliadora em: 5 de julho de 2019.

BANCA AVALIADORA

Prof. Dr. Braulio Adriano de Mello – UFFS

Prof. Dr. Claunir Pavan – UFFS

Prof. Dr. Emilio Wuerges – UFFS

RESUMO

Modelos de simulação com componentes assíncronos estão sujeitos a violação de tempo. *Checkpoints* podem ser utilizados para executar operações de *rollback* e restaurar a simulação para um estado anterior consistente. Em modelos distribuídos, os componentes podem gerar checkpoints de maneira independente a fim de reduzir o *overhead* de comunicação, o que pode ocasionar a criação de *checkpoints inúteis*. Checkpoints inúteis são indesejados, pois implicam em desperdício de processamento e armazenamento. Com o objetivo de reduzir checkpoints inúteis e o número de tempo gasto em rollback, este trabalho apresenta um algoritmo para calcular intervalos flexíveis entre checkpoints. A estratégia foi implementada no DCB (Distributed Co-Simulation Backbone). Os resultados obtidos mostraram que a implementação reduziu em quase 10% o tempo de rollback da simulação, como também, uma redução de quase 24% a quantidade de checkpoints inúteis criados.

Palavras-chave: Simulação Distribuída. Checkpoints. Intervalo Flexível de Checkpoint. Rollback. Sobrecarga.

ABSTRACT

Simulation models with asynchronous components are subject to time violation. *Checkpoints* can be used to perform *rollback* operations and restore the simulation to a previous consistent state. In distributed models, components can independently checkpoints to reduce communication *overhead*, which can lead to the creation of *useless checkpoints*. Useless checkpoints are unwanted as they entail wasted processing and storage. In order to reduce useless checkpoints and the number of time spent in rollback, this paper presents an algorithm to calculate flexible intervals between checkpoints. The strategy was implemented in the Distributed Co-Simulation Backbone (DCB). The results showed that the implementation reduced the simulation rollback time by almost 10%, as well as a reduction of almost 24% in the number of useless checkpoints created.

Keywords: Distributed Simulation. Checkpoints. Flexible Checkpoints Interval. Rollback. Overhead.

LISTA DE ILUSTRAÇÕES

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

A simulação é uma ferramenta para estudo, análise e avaliação de sistemas reais ou hipotéticos. Em simulação, modelos são usados para representar o comportamento dos sistemas. Sobre os modelos podem ser realizados experimentos com a finalidade de entender seu comportamento e também avaliar estratégias para auxiliar na tomada de decisões (20) (21).

Modelos de simulação podem ser particionados em componentes distintos e distribuídos. Distribuir os componentes do modelo, seja em uma arquitetura multiprocessada ou em computadores conectados por uma rede, pode reduzir o tempo de execução da simulação (2).

A distribuição dos componentes requer protocolos de sincronização entre eles, que podem utilizar um tempo virtual global (GVT) único para todos os componentes da simulação. Cada componente também mantém um tempo virtual local (LVT) (12).

Na simulação, os componentes se comunicam através da troca de mensagens. Quando um componente é assíncrono, ou seja, quando avança seu tempo local sem limitação pelo tempo global, existe a possibilidade de ocorrer violações de tempo (LCC), levando a simulação a um estado inconsistente. Quando isso ocorre, deve haver alguma maneira de retornar a um estado anterior consistente. Isso é feito utilizando o recurso de checkpoints, que permite que um componente possa retornar a um estado anterior consistente (através de operações de rollback).

Protocolos otimistas de sincronização permitem que os processos lógicos (LPs) executem eventos fora da ordem de *timestamp*, fornecendo um meio de recuperação de falhas, baseados em pontos de recuperação e mecanismos de *rollback* (10).

Esses protocolos podem utilizar as informações obtidas pelas trocas de mensagens de simulação entre os componentes para reduzir o *overhead*, uma vez que diminui a complexidade do gerenciamento, embora exista a possibilidade de se criar *checkpoints inúteis*.

Estabelecer intervalos de tempo flexíveis entre *checkpoints* é uma das formas de evitar a criação de *checkpoints inúteis* e buscar uma melhor distribuição temporal dos checkpoints. Esta estratégia pode contribuir com a redução da demanda de processamento e memória, possibilitando que a simulação possa ser executada em uma arquitetura com baixa capacidade de armazenamento e processamento, e também, a redução do tempo de execução de modelos complexos, resultando em uma simulação mais eficiente.

Foi definido um algoritmo para o cálculo destes intervalos flexíveis de tempo para criação de checkpoints. Após a definição do algoritmo, foi feita a sua implementação no DCB e foram feitos experimentos com a estratégia já existente para o DCB(1, 16) e com a solução proposta neste trabalho, visando comparação e análise dos resultados. A solução com intervalos flexíveis apresentou algumas vantagens em relação ao número de checkpoints inúteis criados.

A sequência deste trabalho está organizado da seguinte forma: no capítulo 2 são apresentados conceitos teóricos necessários para o entendimento do trabalho. A especificação da solução, sua implementação no DCB, o estudo de caso e a análise dos resultados são feitas no capítulo 3. E no capítulo 4 são feitas as considerações finais e trabalhos futuros.

2 REFERENCIAL TEÓRICO

2.1 SIMULAÇÃO

A simulação é um recurso computacional, com o qual, tenta-se imitar um dado sistema ou processo, que existe no mundo real ou que seja hipotético (que se pretende implantar no mundo real). Através de um modelo computacional, tenta-se entender o comportamento do sistema, e também, extrair informações que auxiliem na tomada de decisões (20) (21).

Um modelo de simulação é baseado em eventos que ocorrem ao longo do tempo, representando estados do sistema que está sendo simulado. Quando as mudanças de estados ocorrem continuamente no tempo, a simulação é dita contínua. Já em uma simulação discreta, um estado do sistema muda apenas quando ocorre um evento (4). Sempre que se usar o termo simulação neste trabalho, estará se fazendo referência a simulação discreta.

2.2 SIMULAÇÃO DISTRIBUÍDA

Uma simulação pode ser executada de forma paralela ou distribuída, que dentre os principais benefícios, está a redução do tempo de execução da simulação (8). Isso ocorre porque os componentes do modelo estão distribuídos em múltiplos processadores ou em vários computadores distribuídos geograficamente, interconectados por uma rede de comunicação (6).

Basicamente, uma simulação paralela ou distribuída é composta por simulações sequenciais, chamadas de *processos lógicos (PLs)* ou *componentes*. Cada PL possui sua própria noção de tempo de simulação, também chamado de *Local Virtual Time (LVT)*, e os PLs se comunicam através da troca de mensagens (17).

As mensagens de cada PL devem ser processadas na ordem correta, para que os resultados da simulação estejam corretos, ou seja, deve haver sincronização entre as mensagens dos PLs. Os métodos de sincronização são divididos em duas categorias: *conservadora* e *otimista*.

2.3 SIMULAÇÃO CONSERVADORA

Também conhecida como simulação síncrona, ela trata dos componentes síncronos do modelo, e deve prevenir o acontecimento de violações de restrição de causalidade local (*local causality constraint - LCC*). Para isso, deve determinar quando é seguro processar um evento, ou seja, garantir que o PL não receberá eventos com um *timestamp* menor que o seu LVT. Enquanto não houver essa garantia, nenhum evento poderá ser processado (8), e o PL deve ser bloqueado, podendo levar a situações de impasse (*deadlock*) se precauções não forem tomadas (7).

2.4 SIMULAÇÃO OTIMISTA

Também conhecida como simulação assíncrona, ela trata dos componentes assíncronos do modelo. Diferente dos protocolos conservadores que previnem o acontecimento de LCCs, os protocolos otimistas permitem que ocorram essas violações, mas fornecem mecanismos de *rollback* e recuperação de tais violações (9). Realizar uma operação de *rollback* envolve a restauração do estado do PL para um estado anterior ao do acontecimento da violação, para isso são necessários *checkpoints* (8).

2.5 CHECKPOINTS

Existem várias técnicas para recuperação de falhas, como checkpointing, registro de histórico de mensagens, replicação ativa, entre outros (11). Neste trabalho será abordado apenas a técnica de utilização de *checkpoints*, por ser um dos mais conhecidos e também, pelo fato de que a política atual para recuperação de falhas do DCB utiliza deste método.

Um *checkpoint* é uma cópia do estado atual de um processo. Ele salva informações suficientes em armazenamento estável, de modo que, caso ocorra uma falha, seja possível restaurar um estado do processo anterior a esta falha (14). As técnicas de recuperação de falhas baseadas em checkpoint podem ser classificadas em três categorias: *checkpointing* não-coordenado, coordenado e induzido por comunicação (3).

2.5.1 Checkpointing não-coordenado

Esta técnica permite ao processo criar um checkpoint quando lhe for mais conveniente, dando-lhe o máximo de autonomia para tomar essa decisão.

Porém, essa técnica apresenta algumas desvantagens, como a possibilidade do *efeito dominó*, também chamado *rollback em cascata*, que é a ocorrência de sucessivas operações de rollback para se alcançar um estado anterior consistente, que pode levar a perda de uma quantidade significativa de esforço computacional, como por exemplo, retroceder até o ponto inicial de uma simulação. Outra desvantagem é a possibilidade de um checkpoint criado ser inútil, ou seja, de não fazer parte de um estado global consistente (3).

2.5.2 Checkpointing coordenado

O checkpointing coordenado exige que os processos organizem seus checkpoints de maneira coordenada, para formar um estado global consistente. Com esta técnica, a recuperação não é suscetível ao efeito dominó, e também, não exige que cada processo armazene vários checkpoints, reduzindo o overhead de armazenamento.

Porém, como essa abordagem exige troca de informações entre os processos para formar um estado global consistente, a simulação fica bloqueada enquanto este estado não é formado. Existem algumas abordagens para tentar evitar esse bloqueio, como a coordenação de checkpoint sem bloqueio, checkpointing com relógios sincronizados, checkpointing e comunicação confiável e coordenação mínima de checkpoint (3).

2.5.3 Checkpointing induzido por comunicação

Nessa técnica existem dois tipos de checkpoints: local e forçado. Checkpoints locais são tomados de forma independente por cada processo, enquanto que checkpoints forçados são tomados através de informações específicas acopladas nas mensagens de simulação recebidas por cada processo.

Nessa classe de protocolos, não existe a troca de mensagens especiais 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, como também, o efeito dominó.

2.6 INTERVALO DE CHECKPOINT FLEXÍVEL

Esta seção apresenta algumas propostas de solução para intervalos de checkpoints flexíveis, que tenham propósitos semelhantes ao deste trabalho.

Em (13) os autores propuseram um algoritmo de seleção de intervalo de checkpoint, que determina rapidamente o intervalo de checkpoint ideal durante e execução de uma simulação, utilizando informações como o overhead de salvamento de um checkpoint e o tempo de execução de eventos. A proposta é encontrar rapidamente um valor ótimo para o intervalo de checkpoint e, então, este valor é utilizado como intervalo de checkpoint durante o restante da simulação. Comentários sobre a escolha deste valor e efeitos de sua escolha são feitas pelos autores.

Já em (19), é apresentado um método para implementar *checkpointing* adaptativo em cada processo lógico, ou seja, cada PL pode continuamente adaptar o intervalo de checkpoint, utilizando informações observadas durante a simulação, como o número de rollbacks, o tempo médio de um checkpoint, o tempo médio para restaurar um estado e o tempo médio de execução de um evento. Devido a essa possibilidade de adaptação contínua do intervalo, resultados desse trabalho mostraram que o overhead para esse método é baixo, além de melhorar o desempenho da simulação, já que cada PL pode adaptar seu intervalo individualmente.

No trabalho de (18) os autores apresentam alguns mecanismos de otimização para simulações otimistas, que tentam reduzir a sobrecarga de memória e processamento. São feitas análises entre estes mecanismos e quais os efeitos de se usar cada um deles. Um desses mecanismos é o *dynamic checkpointing*, que periodicamente ajusta o tamanho do intervalo entre checkpoints tentando equilibrar a relação custos com salvamento de estados versus custos de

reexecução de eventos. Resultados do trabalho mostram que utilizar *dynamic checkpointing* acelerou em 20% o tempo de simulação.

Já no trabalho de (22), é descrito um modelo de avaliação de tempo discreto com um número finito de checkpoints disponíveis, onde os autores propõem uma técnica de checkpointing adaptativa, que elimina desse conjunto finito de checkpoints disponíveis, aquele checkpoint cuja sua remoção faz com que esse conjunto tenha um arranjo de tempo mais efetivo, ou seja, os checkpoints desse conjunto ficam melhor distribuídos no tempo.

A proposta de (1) tenta identificar se um estado é seguro para criar um checkpoint. Quando um estado é identificado como seguro, existe a tentativa de criação de um checkpoint. Tentativa porque antes de efetivar a criação do checkpoint, o protocolo proposto em (16) verifica se o checkpoint será útil, utilizando de algumas métricas e cálculos probabilísticos. O checkpoint é criado caso seja considerado útil, do contrário, é ignorado. Porém, nenhuma das duas propostas apresenta uma solução específica para estabelecer um limite de tempo mínimo e máximo ou a flexibilização do intervalo entre ckeckpoints.

2.7 DISTRIBUTED CO-SIMULATION BACKBONE - DCB

O DCB é uma arquitetura de simulação que tem como objetivo principal oferecer suporte à execução distribuída de modelos heterogêneos de simulação (15).

A estrutura básica do DCB constitui de quatro módulos principais: o *gateway*, o embaixador do DCB (EDCB), o embaixador do federado (EF) e o Núcleo do DCB (NDCB).

Como os componentes (no DCB também chamados de federados) do modelo podem ser heterogêneos, é necessário que haja uma interface que faça a comunicação entre tais componentes e os módulos do DCB (EF, EDCB, NDCB), fazendo a tradução de dados de um formato origem para um formato destino.

O *EF* tem como propósito geral o gerenciamento de mensagens recebidas de outros federados, sejam eles remotos ou locais. É ele quem realiza a decodificação de pacotes recebidos e participa das atividades de gerenciamento do tempo de simulação.

Já o *EDCB* tem como propósito geral o gerenciamento de mensagens que são enviadas do federado que ele representa. E juntamente com o EF, faz o gerenciamento do tempo virtual local (*LVT*).

E o *NDCB* por sua vez, tem como propósito geral oferecer serviços de comunicação por troca de mensagens entre os federados, além de manter um valor único em todos os nodos para o tempo virtual global (*GVT*).

2.7.1 Sincronização no DCB

No DCB é utilizado o tempo de evento (*timestamp*) para fazer a sincronização das mensagens. O tempo de evento enviado em uma mensagem indica o tempo em que um evento

deve ocorrer no federado destino.

O DCB suporta sincronização híbrida, que permite a cada federado avançar seu tempo local no modo síncrono ou assíncrono, além da cooperação com federados *notime*, que não utilizam tempo explicitamente no seu modelo.

Federados síncronos utilizam de protocolos que não permitem a execução de um evento fora da ordem crescente de *timestamp*, garantindo que não ocorram violações de tempo. Já os componentes *notime*, organizam suas mensagens de acordo com a ordem em que foram recebidas. Os federados assíncronos, por sua vez, utilizam protocolos que permitem a execução de um evento fora da ordem de *timestamp*, e caso ocorra uma violação de tempo, realiza uma operação de rollback.

O DCB utiliza políticas de checkpoints para prover tolerância a falhas ocasionadas por violações de tempo. A criação de checkpoints é feita de modo não-coordenado, com base em informações obtidas na comunicação entre os componentes (1) (16).

3 IMPLEMENTAÇÃO E ESTUDO DE CASO

Neste capítulo, inicialmente são apresentadas as definições e conceitos sobre *ciclos* e *eventos* utilizados. Em seguida, é descrita a especificação e o funcionamento da solução proposta para intervalos flexíveis entre checkpoints. Posteriormente, a implementação da solução no DCB, o estudo de caso e a análise dos resultados colhidos no estudo de caso são apresentadas.

3.1 MODELO DE SELEÇÃO DE INTERVALO DE CHECKPOINT

A solução implementada neste trabalho tem como base o modelo de checkpointing descrito em (13), que propôs o algoritmo 1, para selecionar intervalos de checkpoint dinâmicos (ou flexíveis) durante a simulação. Intervalos de checkpoints flexíveis, podem diminuir a sobrecarga associada ao salvamento e restauração de estados de um processo.

No modelo de checkpointing proposto em (13), um processo considera três fases: *re-execução de eventos*; *execução normal de eventos*; e *rollback*. As fases estão organizadas dentro de *ciclos de computação*. Estas quatro definições são utilizadas na solução e, portanto, são explicadas a seguir. Ao final da seção é apresentado o algoritmo original e as alterações feitas na implementação da solução.

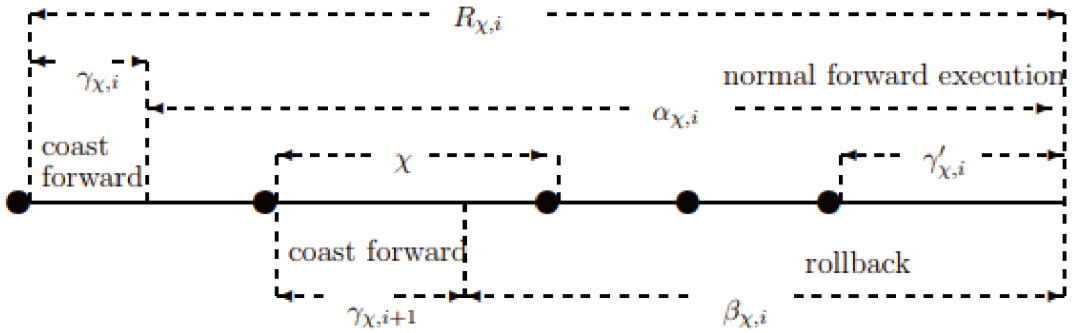

Figura 1 – *i*-ésimo ciclo de computação (13)

3.1.1 Ciclo de Computação

Um ciclo de computação é definido como o período entre dois rollbacks consecutivos. Desta forma, o período de tempo entre dois retrocessos que ocorreram em um processo, é chamado de ciclo de computação. Um ciclo de computação é dividido em três fases: *eventos re-executados*, *eventos executados normalmente*, e *rollback* ou *eventos desfeitos*.

O comprimento, ou o tamanho de um ciclo de computação, é dado pelo número de eventos re-executados, devido à ocorrência de um rollback, somados com os eventos executados normalmente durante um determinado ciclo de computação.

Considerando o i-ésimo ciclo de computação, o comprimento deste ciclo é definido, conforme a figura 1, da seguinte forma:

$$R_{\chi,i} = \gamma_{\chi,i} + \alpha_{\chi,i} \tag{3.1}$$

3.1.2 Eventos re-executados

Quando ocorre uma LCC, para que seja possível restaurar a simulação a um estado consistente é preciso re-executar os eventos de um determinado período de tempo. Este período de tempo é delimitado pelo checkpoint necessário para retomar a simulação depois da ocorrência de uma LCC e pelo *timestamp* da mensagem que causou a violação. Ou seja, os eventos que estão neste intervalo de tempo devem ser re-executados. Os eventos re-executados são denotados por $\gamma_{\chi,i}$, onde χ é o intervalo de checkpoint e i indica o i-ésimo ciclo de computação. Que pode ser observado na Figura 1 pelo período nomeado como *coast forward*.

3.1.3 Eventos executados normalmente

Os eventos que são considerados *eventos executados normalmente*, são todos os eventos de um ciclo de computação exceto aqueles que compõem o período de tempo de re-execução de eventos. Os eventos executados normalmente, denotado por $\alpha_{\chi,i}$, no i-ésimo ciclo de computação com intervalo de checkpoint χ, conforme a Figura 1.

3.1.4 Eventos desfeitos

Neste modelo de checkpointing, os eventos que estão no intervalo compreendido pela *fase de rollback*, conforme a Figura 1, denotado por $\beta_{\chi,i}$, são considerados os eventos desfeitos. Sendo assim, a quantidade de eventos que foram desfeitos devido a ocorrência de uma LCC, compreendem o comprimento de um rollback. Denota-se por $\beta_{\chi,i}$, os eventos desfeitos no i-ésimo ciclo de computação, onde o intervalo de checkpoint é χ.

3.2 ALGORITMO DE SELEÇÃO DE INTERVALO DE CHECKPOINT

Esta seção apresenta um algoritmo para calcular intervalos de checkpoints. Uma simplificação do algoritmo é descrita.

A primeira iteração do algoritmo 1, linha 6, inicia definindo o intervalo de checkpoint, χ = 1. Depois que N eventos tenham sido *committed* [1] em um processo, são obtidas as seguintes informações: δ_e (tempo esperado de execução de um evento); δ_s (overhead de salvar um estado de um processo); k_1 (quantidade de rollbacks que ocorreram em um processo enquanto χ =

[1] De acordo com (5), eventos *committed* são aqueles que tem um LVT menor que o GVT, uma vez que um evento é considerado *committed* apenas quando o GVT avança no tempo.

Algoritmo 1 – Algoritmo de Seleção de Intervalo de Checkpoint (13)

```
1  N ← eventoscommitted
2  i ← 1
3  iteracao ← 1
4  i é atualizado verificando a quantidade de eventos committed
5  if i = N then
6      if iteracao = 1 then
7          χ = 1
8          calcula ᾱ
9          calcula χ⁻
10         calcula χ⁺
11         define χ
12     else
13         Tem-se i amostras para R_χ
14         Usa ajuste de curva para predizer o comportamento da curva R_χ sobre χ
15         Escolhe-se χ que minimiza T_χ
16     end
17     incrementa iteracao em uma unidade
18 end
```

1); e R_1 (a soma dos comprimentos dos ciclos de computação que tenham como intervalo de checkpoint $\chi = 1$).

A partir destas informações, é calculado $\overline{\alpha}_\chi = \frac{R_\chi}{k_\chi}$, linha 8 do algoritmo 1, que é a média de eventos executados normalmente, quando o intervalo de checkpoint era χ. Onde R_χ é a soma dos comprimentos dos ciclos de computação cujo intervalo de checkpoint era χ, e k_χ é a soma da quantidade de rollbacks que ocorreram enquanto o intervalo de checkpoint era χ.

Duas equações fazem uso das informações acima para definir os intervalos mínimo e máximo de checkpoint. São elas, respectivamente:

$$\chi^- = \left\lfloor \sqrt{\frac{(\overline{\alpha}_1 - 1)\delta_s}{\delta_e}} \right\rfloor \tag{3.2}$$

$$\chi^+ = \left\lceil \sqrt{\frac{(2\overline{\alpha}_1 + 1)\delta_s}{\delta_e}} \right\rceil \tag{3.3}$$

A equação 3.2, é a linha 9 do algoritmo 1, e a equação 3.3, é a linha 10 do algoritmo 1.

Então é selecionado um valor para o intervalo de checkpoint entre 3.2 e 3.3. Cada iteração subsequente inicia após outros N eventos terem sido *committed*. Uma iteração utiliza informações da iteração anterior, partindo da 1ª iteração, que juntamente com as amostras armazenadas dos ciclos de computação, tentam predizer o comportamento de uma curva R_χ por χ utilizando alguma técnica de ajuste de curva.

O valor de χ selecionado na *i*-ésima iteração, é o intervalo de checkpoint que minimiza T_χ, equação 3.4, que é o tempo necessário para finalizar uma quantidade definida de eventos em

uma simulação (13).

$$T_\chi = R_\chi \left(\delta_e + \frac{\delta_s}{\chi} \right) \tag{3.4}$$

Esse algoritmo é encerrado na iteração em que é encontrado um valor tido como ótimo (um mesmo valor para o intervalo de checkpoint que já tenha sido encontrado dentre as i iterações), onde i, pelos estudos dos autores varia entre 2 e 3. Sendo assim, após no máximo 3 iterações o algoritmo termina e o valor ótimo encontrado é utilizado durante o restante da simulação.

Como a proposta deste trabalho é tornar flexível o intervalo de tempo entre checkpoints, e também encontrar um limite mínimo e máximo para o intervalo, o algoritmo 1 foi simplificado, executando apenas as operações apresentadas para a 1ª iteração do algoritmo 1. Sendo assim, o algoritmo 2 foi então, implementado como solução neste trabalho.

Algoritmo 2 – Algoritmo de Seleção de Intervalo de Checkpoint Simplificado

```
1  N ← eventoscommitted
2  i ← 1
3  iteracao ← 1
4  i é atualizado verificando a quantidade de eventos committed
5  if i = N then
6  |   χ = 1
7  |   calcula α̅
8  |   calcula χ⁻
9  |   calcula χ⁺
10 |   define χ
11 |   incrementa iteracao em uma unidade
12 end
```

A solução permite calcular continuamente o intervalo de checkpoint durante todo o percurso da simulação. Diferentemente do algoritmo 1, que após um número pequeno de iterações, é encerrado. Isso por que o algoritmo 1 encontra valores ótimos locais para o intervalo de checkpoint, e segundo os autores, esse intervalo selecionado pode se tornar um valor ótimo global, caso a simulação tenha um comportamento padrão. Como nem sempre a simulação terá um comportamento padrão, dificilmente poderá se encontrar um valor ótimo global. Devido a esse fato, e seguindo uma recomendação dos autores, o algoritmo 2 é executado continuamente durante toda a simulação, atualizando os valores para os intervalos mínimo e máximo e também o valor para o intervalo de checkpoint.

3.3 IMPLEMENTAÇÃO NO DCB

Esta seção apresenta as alterações feitas no DCB, para que pudesse ser implementado o algoritmo 2. Além da simplificação do algoritmo 1, os valores dos parâmetros δ_s, δ_e e N, apresentados na seção 3.2 foram alterados. Visto que, em experimentos realizados em (13), δ_s

variou entre 1,23 e 9,23 , $\delta_e = 45,06$ e N iniciou-se com um valor grande, e no decorrer da execução do algoritmo foi diminuindo.

A sobrecarga associada ao salvamento de um estado de um processo (δ_s) e a sobrecarga associada ao tempo de execução de um evento (δ_e) não depende do DCB, mas sim do recurso computacional em que a simulação será executada. Desta forma, ambas, (δ_s) e (δ_e) ficaram definidas com o valor 1. Ou seja, para o DCB a sobrecarga de salvar um estado e de executar um evento tem custo igual a 1.

Já o parâmetro N ficou definido com o valor 25. Isso porque na versão atual do DCB não há tratamento para o gerenciamento do GVT para componentes assíncronos, não sendo possível verificar a quantidade de eventos *committed*. Então, definiu-se que a cada 25 eventos executados, seria chamado o algoritmo 2 para calcular o intervalo de checkpoint.

Sendo assim, para que fosse possível a implementação do algoritmo 2, foi criada uma nova classe, *CicloDeComputacao* no DCB, de acordo com a Figura 2. Uma instância dessa classe armazena informações de um ciclo de computação. Ou seja, para cada rollback que ocorrer na simulação, será armazenado uma instância dessa classe, com a quantidade de eventos re-executados, eventos executados normalmente, bem como o comprimento do ciclo de computação, e o valor corrente do intervalo de checkpoint quando ocorreu o rollback.

```
1  public class CicloDeComputacao {
2      private int totalEventosReexecutados;
3      private int totalEventosExecutadosNormalmente;
4      private int intervaloCheckpoint;
5      private int comprimentoCicloDeComputacao;
6  }
```

Figura 2 – Classe criada - CicloDeComputacao

Com essas informações armazenadas é possível fazer os cálculos para definir intervalos mínimo e máximo para criação de checkpoints. Na classe *Chat.java*, foi criado um método, chamado *lin()*, onde são calculados os valores para os intervalo mínimo e máximo de checkpoint. No método, conforme o algoritmo 2, é calculada a média de eventos executados durante os ciclos de computação que tenham como intervalo de checkpoint χ, linha 7, (conforme a figura 3, pelo método *calculaAlpha*), percorre os ciclos de computação para extrair as informações necessárias; depois são calculados os intervalos mínimo e máximo, respectivamente, nas linhas 8 e 9; e por fim, é definido o intervalo de checkpoint, linha 10, delimitado pelos intervalos mínimo e máximo calculados anteriormente.

O método *lin()* é chamado após N eventos terem sido executados em um processo, conforme explicado anteriormente. Sempre que o LVT do componente é atualizado (através da Thread *AtualizaLVT* , é feita a verificação *eventos executados* >= *N*, que em caso afirmativo, chamada o método *lin()*.

Estruturas condicionais foram adicionadas no método *setReceivedText()*, linha 4 na Figura 4, e no método *run()* da Thread *CheckpointAnalyzer*, linhas 3 e 4 na Figura 5, sempre antes

```
public void lin() {
    Double alpha1 = calculaAlpha(intervaloCheckpoint);
    intervaloMax = (int)Math.ceil(Math.sqrt(((2*alpha1+1)*overheadSalvarEstado)/tempoEsperadoExecucaoEvento));
    intervaloMin = (int)Math.floor(Math.sqrt(((alpha1-1)*overheadSalvarEstado)/tempoEsperadoExecucaoEvento));
    Random gerador = new Random();
    intervaloCheckpoint = gerador.nextInt(intervaloMax - intervaloMin + 1) + intervaloMin;
}
public Double calculaAlpha(int intervaloCheckpoint){
    int contSomaRollbacks = 0;
    int contSomaComprimentos = 0;
    for(CicloDeComputacao c : ciclos){
        if(c.getIntervaloCheckpoint() == intervaloCheckpoint ){
            contSomaRollbacks++;
            contSomaComprimentos = contSomaComprimentos + c.getComprimentoCicloDeComputacao();
        }
    }
    return Math.ceil( (double) contSomaComprimentos / contSomaRollbacks ) ;
}
```

Figura 3 – Implementação do algoritmo 2 através do método *lin()*

```
public void setReceivedText(String value) {
    ...
    if (indexReceivedINT > CkpIndex) {
        if(contadorDeMensagens >= intervaloMin && contadorDeMensagens <= intervaloMax){
            Integer aux = Integer.parseInt(chatLVT);
            aux += 100;
            this.setCheckpoint(aux.toString());
            CkpIndex = indexReceivedINT;
        }
    }
    ...
}
```

Figura 4 – Alteração no código do método *setReceivedText()*

de chamar o método que tenta criar um checkpoint, *setCheckpoint()*. Ou seja, sempre antes de tentar criar um checkpoint, é verificado se já foram executados eventos suficientes para a criação de um checkpoint.

```
public void run() {
    ...
    if(outerClass.contadorDeMensagens >= outerClass.intervaloMin
        && outerClass.contadorDeMensagens <= outerClass.intervaloMax){
        if(outerClass.contadorDeMensagens > 10) {
            outerClass.setCheckpoint(outerClass.chatLVT);
        }
        if (Integer.valueOf(outerClass.chatLVT) - Integer.valueOf(ultimoCheckpoint) >= 200) {
            if(!(outerClass.checkpointsArray.contains(outerClass.chatLVT))) {
                outerClass.setCheckpoint(outerClass.chatLVT);
            }
        }
    }
    ...
}
```

Figura 5 – Alteração no código do método *run()* da Thread *CheckpointAnalyzer*

3.4 ESTUDO DE CASO

Esta seção apresenta a especificação de um modelo de troca de mensagens usado no desenvolvimento de um estudo de caso em que a solução proposta é experimentada. Os resultados dos experimentos são comparados com a implementação anterior do DCB, realizada em (16), na análise dos resultados.

O modelo de troca de mensagens é composto por cinco processos, no DCB, chamados de *Chat5*, *Chat6*, *Chat7*, *Chat8* e *Chat9*, todos evoluindo no tempo de forma assíncrona. A topologia do modelo utilizado nos experimentos é a mesma utilizada em (16). Na Figura 7, observa-se a topologia, onde *P0*, *P1*, *P2*, *P3*, *P4*, correspondem respectivamente, aos processos *Chat5*, *Chat6*, *Chat7*, *Chat8* e *Chat9*.

Os valores das arestas correspondem a probabilidade de um processo se comunicar com outro processo em um determinado intervalo de tempo, que segundo (16), torna a realização de eventos externos probabilística e o intervalo entre a execução discreto. Na Figura 7, a esquerda tem-se a topologia usada em (16) e a direita tem-se a topologia utilizada nos experimentos deste trabalho. A diferença está nas probabilidades de um processo se comunicar com outro. No DCB, estas alterações foram feitas no método *run()* da Thread *MensagensAutomaticas*, conforme exemplo da Figura 6.

```
1  public void run() {
2      ...
3      switch (id) {
4      num = gerador.nextInt(100);
5      case 5:
6          if (num < 50) { //envia msg para chat 6 }
7          if (num < 80) { //envia msg para chat 7 }
8          break;
9      case 6:
10         if (num < 50) { // envia msg para chat 7 }
11         if (num < 80) { // envia msg para chat 8 }
12         break;
13     case 7:
14         if (num < 50) { // envia msg para chat 8 }
15         break;
16     case 8:
17         if (num < 50) { // envia msg para chat 9 }
18         break;
19     }
20     ...
21 }
```

Figura 6 – Alteração no código do método *run()* da Thread *MensagensAutomaticas*

Os processos incrementam seus LVTs de maneira diferente ao executar eventos, para que pudessem ser criadas situações de rollbacks, a fim de testar a implementação da solução. Utilizando os mesmos valores que em (16), o *Chat6* aumenta 90 unidades de tempo a cada evento, o *Chat7* aumenta 95 e o *Chat8* aumenta 100, forçando alguns rollbacks (pelo 6, no 7 e no 8, e pelo 7, no processo 8).

Foram executadas cinco simulações para cada implementação, a versão anterior do DCB e a implementação deste trabalho. Cada simulação foi executada por 500.000 unidades de

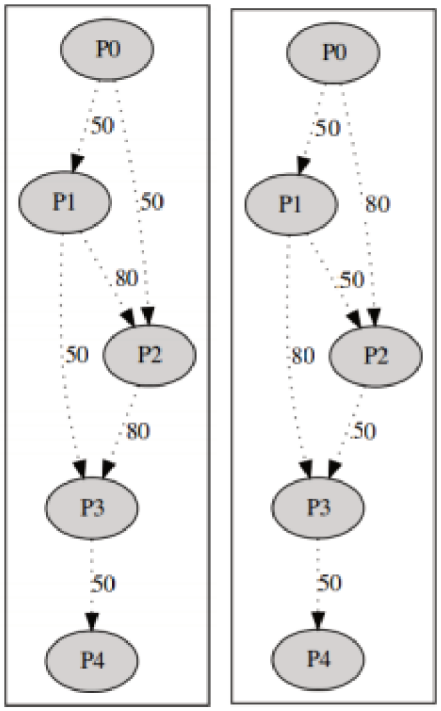

Figura 7 – Grafo de relação de troca de mensagens entre os processos do modelo (16)

tempo, em um único computador Intel®Core™i5-8400 CPU @ 2.80GHz.

3.5 ANÁLISE DOS RESULTADOS

Os resultados obtidos, contemplam a média das cinco execuções para cada implementação. Para fazer a análise dos resultados, foram contabilizadas para cada processo: a quantidade de mensagens enviadas e recebidas; a quantidade de tentativas de checkpoints; a quantidade de checkpoints criados e quantos checkpoints inúteis foram criados; a quantidade de rollbacks e o tempo total de rollback; e o tempo total de simulação.

Inicia-se a análise observando que o número de mensagens trocadas (mensagens enviadas e recebidas) entre os componentes têm se mantido entre as duas implementações, conforme observa-se no gráfico da Figura 8. Mostrando que o modelo de intervalo flexível para criação de checkpoints não degradou o desempenho da simulação no que diz respeito a quantidade de eventos executados, ou seja, o método não trás sobrecarga à execução da simulação.

Outra informação que se observa no estudo de caso é a redução no número de tentativas de checkpoint, conforme o primeiro gráfico da Figura 9. O somatório da média de tentativas de checkpoints de cada componente, teve uma redução de 6,71%. A média da quantidade de checkpoints efetivamente criados também reduziu. Como pode-se observar no segundo gráfico da figura 9, a soma da média de checkpoints criados por todos os componentes reduziu 9,41%. Outra métrica que teve redução foi o número de checkpoints inúteis, com redução de 23,64%. Esses dados demonstram que o modelo de intervalo flexível de checkpoint, juntamente com os

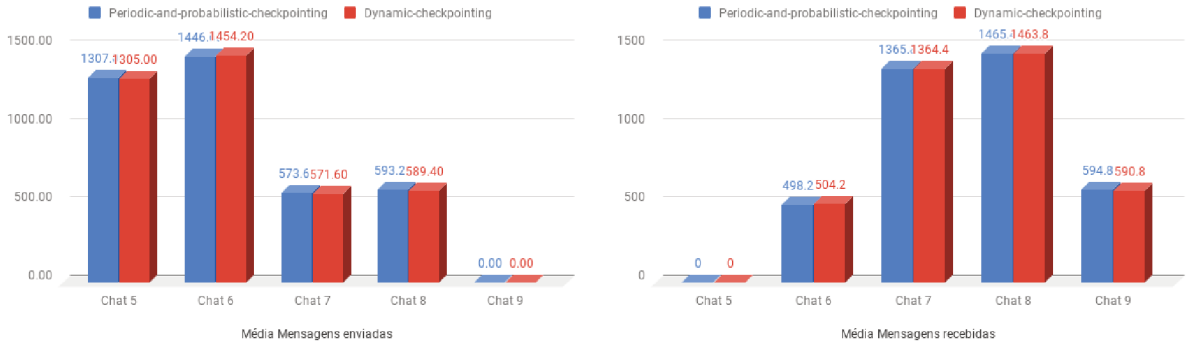

Figura 8 – Média de mensagens enviadas e recebidas por cada componente da simulação

modelos de checkpoint já implementados no DCB, têm uma melhora na distribuição temporal entre os checkpoints.

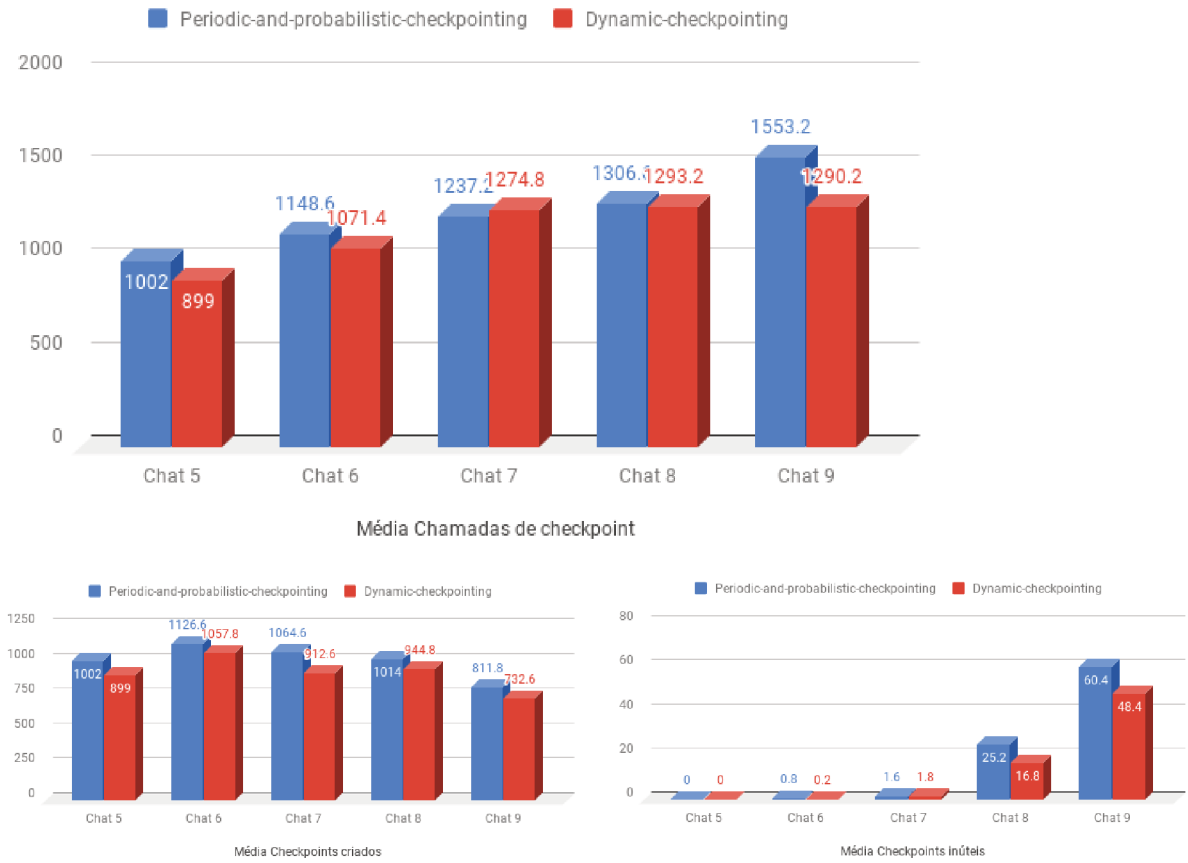

Figura 9 – Média de: tentativas de checkpoints, checkpoints criados, checkpoints inúteis

Em relação aos rollbacks, na média houve um aumento de 32,97% na quantidade de rollbacks realizados. Porém, em relação a média de tempo de rollbacks, houve uma redução de 9,75%, conforme os gráficos da figura 10. Embora tenha ocorrido um aumento significativo na quantidade de rollbacks realizados, que de acordo com (13), esse modelo de checkpointing tende a aumentar a quantidade de rollbacks, o tempo total gasto em rollbacks reduziu. Percebendo

assim que, a distribuição temporal dos checkpoints foi satisfatória, o que ocasionou uma redução no tempo de rollback.

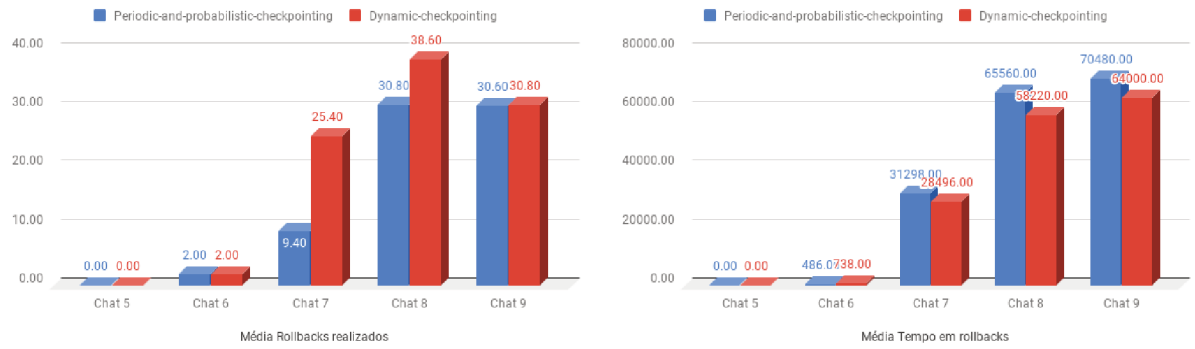

Figura 10 – Média de rollbacks realizados e tempo total de rollbacks

Como consequência de uma melhor distribuição temporal dos checkpoints, e da redução no tempo total de rollback, houve uma redução no tempo total simulado, de 1,03%, conforme o gráfico da Figura 11.

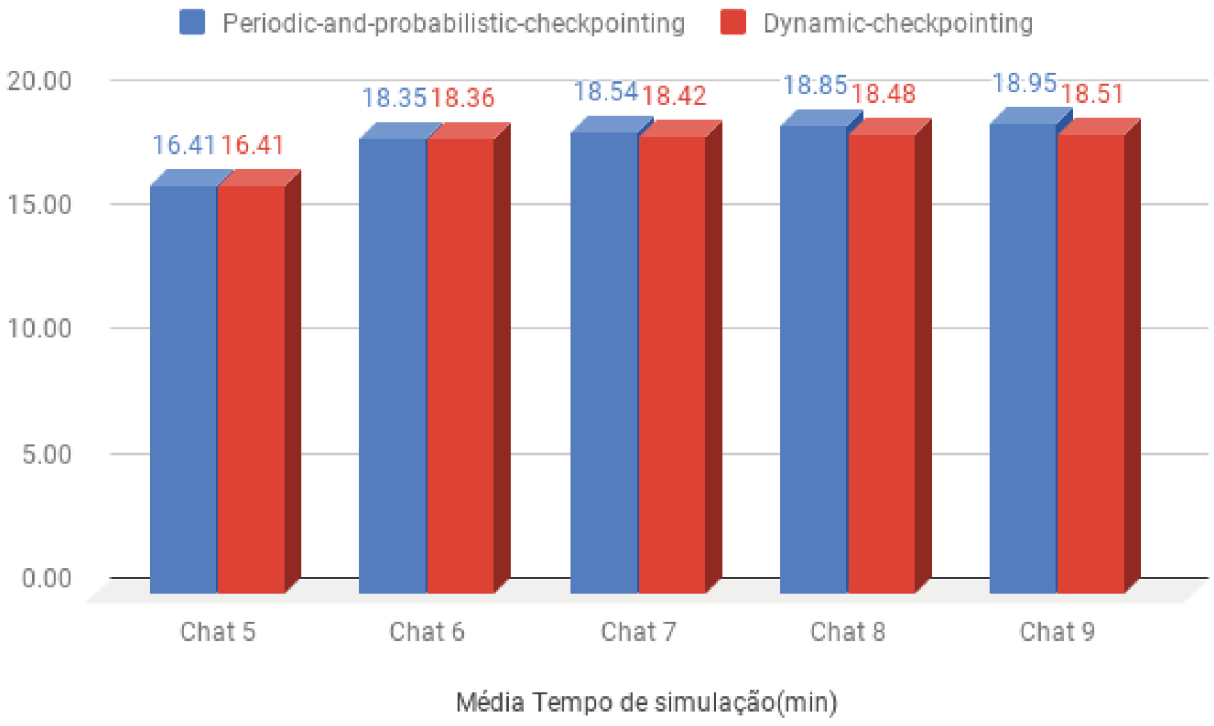

Figura 11 – Média de tempo de simulação

4 CONCLUSÃO

Este trabalho apresentou um modelo para criação de checkpoints que torna flexível o intervalo de tempo entre checkpoints. A solução reduz a sobrecarga de salvamento de estados dos componentes, sem degradar a simulação em relação ao tempo gasto em rollbacks e consequentemente, o tempo total de simulação.

A implementação do modelo de checkpointing diminuiu tanto o número de tentativas de checkpoints, quanto a quantidade de checkpoints criados, e também de checkpoints inúteis, em relação a implementações anteriores de checkpointing no DCB (16, 1).

A redução mais significativa foi a dos checkpoints inúteis, 23,64% a menos que na implementação anterior. Essa redução, além de diminuir a sobrecarga de se salvar um estado de um processo, faz com que, por exemplo, em um ambiente com capacidade de armazenamento limitado, seja ele volátil ou não-volátil, possa executar simulações mais complexas sem que ocorram problemas críticos com gargalos.

Como trabalhos futuros, pode-se utilizar o DCB para executar um modelo mais realístico, a fim de validar a solução proposta neste trabalho, bem como as soluções já existentes no DCB. Também podem ser feitos estudos, com a solução deste trabalho, alterando os valores para os parâmetros utilizados, na tentativa de encontrar valores ideias para os parâmetros.

Poderiam ser criadas outras topologias para o modelo de troca de mensagens, com valores de probabilidade diferentes de um processo se comunicar com outro, e com o valor a ser incrementado no LVT do processo após a execução de cada evento. Isso para que pudesse ser alterado o comportamento da simulação e assim, verificar para quais outras topologias o intervalo flexível de checkpoints trás benefícios ou não. Uma topologia em anel, onde todos os processos enviam e recebem mensagens, poderia ser testada. Outra topologia que poderia ser utilizada, seria um anel fortemente conectado, onde cada processo está conectado com os demais processos, enviando e recebendo mensagens de todos. Ou seja, topologias em que tenha uma quantidade considerável de ciclos, onde a probabilidade de ocorrer rollbacks é alta, para que possa ser verificado se a distribuição temporal dos checkpoints, através dos intervalos flexíveis, trás benefícios à execução da simulação.

REFERÊNCIAS

1 BIZZANI, G.; MELLO, B. A. **Identificação de estados seguros para reduzir a criação de checkpoints sem valor**. [S.l.: s.n.], 2016. Trabalho de Conclusão de Curso, Universidade Federal da Fronteira Sul.

2 CHANDY, K. Mani; MISRA, Jayadev. Asynchronous distributed simulation via a sequence of parallel computations. **Communications of the ACM**, ACM, v. 24, n. 4, p. 198–206, 1981.

3 ELNOZAHY, E. N. (Mootaz) et al. A Survey of Rollback-recovery Protocols in Message-passing Systems. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 34, n. 3, p. 375–408, set. 2002. ISSN 0360-0300. DOI: 10.1145/568522.568525. Disponível em: <http://doi.acm.org/10.1145/568522.568525>.

4 FERSCHA, Alois; TRIPATHI, Satish K. **Parallel and distributed simulation of discrete event systems**. [S.l.], 1998.

5 FLEISCHMANN, J.; WILSEY, P. A. Comparative analysis of periodic state saving techniques in time warp simulators. In: PROCEEDINGS 9th Workshop on Parallel and Distributed Simulation (ACM/IEEE). [S.l.: s.n.], jun. 1995. p. 50–58. DOI: 10.1109/PADS.1995.404317.

6 FUJIMOTO, Richard M. **Parallel and distributed simulation systems**. [S.l.]: Wiley New York, 2000. v. 300.

7 FUJIMOTO, Richard M. Parallel Discrete Event Simulation. **Commun. ACM**, ACM, New York, NY, USA, v. 33, n. 10, p. 30–53, out. 1990. ISSN 0001-0782. DOI: 10.1145/84537.84545. Disponível em: <http://doi.acm.org/10.1145/84537.84545>.

8 _____. Parallel Simulation: Parallel and Distributed Simulation Systems. In: PROCEEDINGS of the 33Nd Conference on Winter Simulation. Arlington, Virginia: IEEE Computer Society, 2001. (WSC '01), p. 147–157. ISBN 0-7803-7309-X. Disponível em: <http://dl.acm.org/citation.cfm?id=564124.564145>.

9 _____. Research Challenges in Parallel and Distributed Simulation. **ACM Trans. Model. Comput. Simul.**, ACM, New York, NY, USA, v. 26, n. 4, 22:1–22:29, maio 2016. ISSN 1049-3301. DOI: 10.1145/2866577. Disponível em: <http://doi.acm.org/10.1145/2866577>.

10 _____. Time Management in The High Level Architecture. **SIMULATION**, v. 71, n. 6, p. 388–400, 1998. DOI: 10.1177/003754979807100604. eprint: https://doi.org/10.1177/003754979807100604. Disponível em: <https://doi.org/10.1177/003754979807100604>.

11 JOHNSON, David Bruce. **Distributed system fault tolerance using message logging and checkpointing**. 1990. Tese (Doutorado) – Rice University.

12 LAMPORT, Leslie. Time, Clocks, and the Ordering of Events in a Distributed System. **Commun. ACM**, ACM, New York, NY, USA, v. 21, n. 7, p. 558–565, jul. 1978. ISSN 0001-0782. DOI: 10.1145/359545.359563. Disponível em: <http://doi.acm.org/10.1145/359545.359563>.

13 LIN, Yi-Bing et al. Dynamic Checkpoint Interval Selection in Time Warp Simulation. Citeseer, 2001.

14 MANDAL, Partha Sarathi; MUKHOPADHYAYA, Krishnendu. Concurrent checkpoint initiation and recovery algorithms on asynchronous ring networks. **Journal of Parallel and Distributed Computing**, v. 64, n. 5, p. 649–661, 2004. ISSN 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2004.03.013. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0743731504000401>.

15 MELLO, Braulio Adriano de. **Co-simulação distribuída de sistemas heterogêneos**. 2005. Tese (Doutorado) – Universidade Federal do Rio Grande do Sul.

16 PARIZOTTO, R.; MELLO, B. A. **Métricas de Checkpoints Probabilísticos em Simulação heterogênea Distribuída**. [S.l.: s.n.], 2016. Trabalho de Conclusão de Curso, Universidade Federal da Fronteira Sul.

17 PREISS, Bruno R; MACINTYRE, Ian D; LOUCKS, Wayne M. On the trade-off between time and space in optimistic parallel discrete-event simulation. In:

18 RADHAKRISHNAN, Radharamanan et al. A Comparative Analysis of Various Time Warp Algorithms Implemented in the WARPED Simulation Kernel. In: p. 107–. DOI: 10.1109/SIMSYM.1996.492158.

19 RÖNNGREN, Robert; AYANI, Rassul. Adaptive Checkpointing in Time Warp. **SIGSIM Simul. Dig.**, ACM, New York, NY, USA, v. 24, n. 1, p. 110–117, jul. 1994. ISSN 0163-6103. DOI: 10.1145/195291.182577. Disponível em: <http://doi.acm.org/10.1145/195291.182577>.

20 SHANNON, Robert E. Introduction to simulation. In: ACM. PROCEEDINGS of the 24th conference on Winter simulation. [S.l.: s.n.], 1992. p. 65–73.

21 _____. Introduction to the art and science of simulation. In: IEEE COMPUTER SOCIETY PRESS. PROCEEDINGS of the 30th conference on Winter simulation. [S.l.: s.n.], 1998. p. 7–14.

22 SUZUKI, Ryo; FUKUMOTO, Satoshi; IWASAKI, Kazuhiko. Adaptive Checkpointing for Time Warp Technique with a Limited Number of Checkpoints. In: ICDCS Workshops. [S.l.: s.n.], 2002.