FLÁVIO MARQUES MIGOWSKI CARVALHO

CONTROLE DE CHECKPOINTS PARA EXECUÇÃO OTIMISTA DE MODELOS HETEROGÊNEOS NO DCB

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

Orientador

Prof. Doutor Braulio Adriano de Mello

LAVRAS
MINAS GERAIS – BRASIL
2009

FLÁVIO MARQUES MIGOWSKI CARVALHO

CONTROLE DE CHECKPOINTS PARA EXECUÇÃO OTIMISTA DE MODELOS HETEROGÊNEOS NO DCB

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

APROVADA em 18 de setembro de 2009.

Prof. Doutor Cláudio Fabiano Motta Toledo.

Profa. Doutora Marluce Rodrigues Pereira.

Prof. Doutor Braulio Adriano de Mello
UFLA
(Orientador)

LAVRAS
MINAS GERAIS – BRASIL

DEDICATÓRIA

Dedico este trabalho primeiramente a Deus, por me dar toda a força necessária, não só para sua realização, mas principalmente, para conseguir superar todos problemas. Dedico também a minha família, que sempre me apoiou nos momentos difíceis e dividiu também as alegrias comigo. Meu pai, Josemar, que nos iniciou neste caminho e me mostrou como um pai deve honrar sua família, servindo de exemplo com seu indiscutível caráter. Minha mãe, Ângela, que é um exemplo de fé, abnegação e dedicação à família. Meu irmão, Bruno, que tem um coração imenso e possuidor de um humor sem igual. Minha irmã, Paula, que tem uma ingenuidade angelical e também um carinho pelo próximo imensurável. E por último meus avós, Reynaldo (em memória) e Milta, que são nossos eternos companheiros e nos enchem de alegria e dão bons conselhos adquiridos com suas experiências de vida. Também não posso esquecer dos meus GRANDES amigos: emo, maicow, rogger, shun, Carol e Ariana (não tão grandes fisicamente, porém enormes de espírito), Teresa, nikos, trombada, mestre, lespa, véio entre outros que ficarão para sempre em minha memória. O que teria sido do tempo de faculdade sem estas pessoas para alegrarem meu dia? Sendo com um comentário infeliz sobre a minha pessoa, ou com uma piadinha medíocre, ou um belo porre independente do dia da semana. Dedico também aos professores que tive a oportunidade de estar em contato e adquirir um pouco de seus vastos conhecimentos. Finalizando, dedico aos funcionários que me foram prestativos nas horas necessitadas, e as vezes realizaram mais do que seu trabalho. Como exemplo disso cito apenas dois de vários: Ângela e Deivison, que trabalharam na secretaria do Departamento de Ciência da Computação durante a maior parte dos meus estudos.

RESUMO

Em uma simulação híbrida, elementos otimistas podem tornar o sistema inconsistente ao retornarem no tempo, pois os elementos conservadores não podem voltar no tempo. Para permitir apenas a regressão temporal de elementos otimistas são necessários *checkpoints*. *Checkpoints* são marcas de tempo estabelecidas durante a simulação. A presente monografia apresenta a especificação e a implementação de *checkpoints* não-coordenados como mecanismo para suporte à simulação de elementos otimistas no *Distributed Co-Simulation Backbone* (DCB). Durante a simulação, caso exista a necessidade de retorno no tempo (*rollback*) de elementos otimistas, estes voltarão para os *checkpoints* criados até então. Foram utilizados *checkpoints* não-coordenados com o objetivo de obter resultados úteis e não ter perda de desempenho do sistema. Com a solução de *checkpoints* não-coordenados, os elementos têm a autonomia do estabelecimento e gerenciamento dos *checkpoints*. Com esta autonomia o sistema não teve perda de desempenho, pois não há *overhead* de mensagens para a escolha dos *checkpoints*.

SUMÁRIO

LISTA DE FIGURAS.....v
1. INTRODUÇÃO.....1
1.1. Contextualização e Motivação.....1
1.2. Objetivo.....2
1.3. Estrutura do trabalho.....3
2. REFERENCIAL TEÓRICO.....4
2.1. Simulação de Sistemas.....4
2.2. Simulação Distribuída.....4
2.3. Simulação Heterogênea.....5
2.3.1. Modelos Híbridos.....6
2.4. High Level Architecture.....7
2.4.1. Características do HLA.....8
2.5. Distributed Co-Simulation Backbone.....9
2.5.1. Sincronização no DCB.....10
2.5.2. Mensagens Nulas.....11
2.5.3. Anti-mensagens.....11
2.6. Soluções para rollback.....11
2.6.1. Estado consistente.....12
2.6.2. Rollback baseado em log.....13
2.6.3. Rollback baseado em checkpoints.....14
2.7. Checkpoints não-coordenados.....14
2.7.1. Grafos de dependência e cálculo de linha de recuperação.....15
2.7.2. Efeito dominó.....16
2.8. Checkpoints coordenados.....17
2.8.1. Coordenação não bloqueante de checkpoints.....18

2.8.2. Coordenação mínima de checkpoints.....19
2.9. Checkpoints induzidos à comunicação.....19
3. METODOLOGIA.....21
4. RESULTADOS.....22
4.1. Problemas e especificações do DCB.....22
4.1.1. Problema do efeito dominó.....23
4.1.2. Problema de checkpoints inúteis e inconsistência.....24
4.1.3. Problemas na sincronização.....26
4.1.4. Anti-mensagens.....27
4.1.5. Rollback.....28
4.1.6. Checkpoints.....31
4.1.7. Nova estrutura do DCB.....32
4.2. Estudo de caso.....34
4.2.1. Mecanismos de atualização do LVT.....38
4.2.2. Checkpoints no estudo de caso.....39
4.2.3. Exemplo de execução.....39
5. CONCLUSÕES.....41
5.1. Trabalhos Futuros.....42
6. REFERENCIAL BIBLIOGRÁFICO.....43

LISTA DE FIGURAS

Figura 2.1: Visão funcional de uma federação.....8
Figura 2.2: Arquitetura do DCB.....9
Figura 2.3: Exemplo de execução(a), grafo de dependência(b) e grafo de checkpoints(c).....15
Figura 2.4: Rollback, linha de recuperação e efeito dominó.....17
Figura 4.1: Exemplo de checkpoint inútil.....24
Figura 4.2: Exemplo de inconsistência.....25
Figura 4.3: Problema no rollback.....26
Figura 4.4: Rollback no DCB.....29
Figura 4.5: Rollback e anti-mensagens.....30
Figura 4.6: Estrutura do DCB.....33
Figura 4.7: Exemplo de elemento síncrono.....35
Figura 4.8: Exemplo de elemento assíncrono.....36
Figura 4.9: Exemplo de elemento untimed.....37
Figura 4.10: Atualização do LVT.....38
Figura 4.11: Exemplo de execução.....40

1. INTRODUÇÃO

1.1. Contextualização e Motivação

Uma alternativa para realizar testes em produtos ou previsões sobre seu comportamento é através da simulação. Isso ocorre devido a grande redução de custos na maioria das vezes. A simulação é aplicável a quase todas áreas do conhecimento, apenas deixando de fora as áreas que ainda não são possíveis de serem modeladas computacionalmente. Embora não seja trivial simular fielmente a realidade, é possível obter uma boa aproximação, gerando resultados úteis diante dos propósitos que condizem ao uso da simulação. No cenário de sistemas de simulação, surgiu a simulação distribuída que, de acordo com as características dos elementos que a compõem, pode ser classificada como sistema homogêneo ou heterogêneo.

Uma simulação é considerada heterogênea quando os seus elementos diferem nas suas interfaces, linguagem de descrição e tecnologias de construção (Mello 2009, Souza, Sperb, Mello e Wagner 2005, Reynolds 1988). Na simulação heterogênea existem alguns tópicos que devem ser considerados com devida atenção, pois eles estão ligados diretamente com o desempenho da simulação. São eles: tolerância a falhas, reconfiguração dinâmica, requisitos de tempo real e abordagens otimizadas para o processo de sincronização e comunicação dos elementos da simulação (Reynolds 1988).

Este trabalho utiliza a arquitetura de co-simulação distribuída heretogênea chamada de *Distributed Co-Simulation Backbone* (DCB). O DCB trata dos principais tópicos sobre simulação heterogênea, ele foi inspirado na *High Level Architecture* (HLA). A HLA é um padrão IEEE para arquitetura de sistemas de simulação distribuídos (Fujimoto 2000).

O DCB é distribuído pois sua execução pode ser distribuída lógica e

fisicamente. É heterogêneo pois permite que elementos desenvolvidos em diferentes linguagens e com diferentes noções temporais sejam implementados e trabalhem em conjunto. Embora voltado para o suporte de modelos híbridos no tempo, o DCB não possui os mecanismos necessários ao gerenciamento de *checkpoints*.

Neste cenário, para serem feitos estudos correspondentes à simulações otimistas (assíncronas). Foi necessário estudo de soluções algorítmicas sobre o estabelecimento de *checkpoints* e a implementação da solução que mais se adequava ao DCB.

Os *checkpoints* são necessários para a simulação de elementos otimistas, pois eles são as marcas no tempo que guardam as características do elemento naquele momento. Então, ao voltarem para um *checkpoint* voltam a ter as mesmas características que tinham quando o *checkpoint* foi estabelecido. O ato de voltar no tempo é chamado de *rollback*. Portanto para suportar a simulação de elementos assíncronos, o presente trabalho especificou e implementou mecanismos no DCB para que os elementos pudessem estabelecer e gerenciar *checkpoints*.

Este trabalho apresenta e implementa no DCB uma solução de estabelecimento de *checkpoints* durante a simulação de elementos otimistas: os *checkpoints* não-coordenados. Com os *checkpoints* não-coordenados, os elementos são os responsáveis por estabelecer seus próprios *checkpoints*.

1.2. Objetivo

Especificar e implementar *checkpoints* no DCB utilizando a solução de *checkpoints* não-coordenados, prepararando os módulos do DCB para que possam realizar *rollback* durante a simulação de elementos otimistas. Isso faz com que o *Gateway* de um elemento otimista seja o coordenador de uma ação de *rollback* acessando simultaneamente os métodos específicos do elemento para a

escolha do *checkpoint* e a eliminação das mensagens recebidas, e os métodos específicos dos módulos do DCB que envolvem o disparo das anti-mensagens.

1.3. Estrutura do trabalho

A estrutura deste trabalho está definida da seguinte forma: na Seção 2 são apresentados os principais conceitos necessários para o entendimento da arquitetura utilizada no trabalho, o DCB. Na Seção 3 é mostrada a metodologia utilizada durante a elaboração da pesquisa, bem como o estudo de caso. Os resultados do trabalho e do estudo de caso são apresentados na Seção 4. Por fim na Seção 5 são expostas as conclusões e perspectivas para novos trabalhos, e na Seção 6 a bibliografia utilizada.

2. REFERENCIAL TEÓRICO

2.1. Simulação de Sistemas

Simulação de sistemas é a utilização de meios artificiais para a representação da essência do comportamento de um sistema real, com a finalidade de obter informações relativas a determinado evento ou conjunto de eventos. Eventos esses que isolado ou simultaneamente, podem ocorrer dentro dos limites que caracterizam o objeto de estudo, consiste no que se ponde denominar de simulação de sistemas (Mertens 1976).

A simulação computacional é um processo de experimentos em sistemas ou fenômenos reais, realizados através de modelos computadorizados, os quais representam características observadas em sistemas reais (Nascimento 2005 *apud* Chung 2004). Portanto, nada mais é do que a passagem para um modelo computacional das principais características de um sistema real o qual se deseja simular.

2.2. Simulação Distribuída

A simulação pode ser feita de duas maneiras distintas: sequencial e distribuída. A segunda vem sido desenvolvida principalmente com o intuito de diminuir o tempo de uma simulação sequencial(Bruschi e Santana 2002). Na simulação distribuída existem duas principais abordagens: SRIP (*Single Replication in Parallel*) e a abordagem MRIP (*Multiple Replication in Parallel*). Na simulação distribuída, um único modelo de simulação tem seus processos lógicos executados em ambientes computacionais distribuídos (Mello 2005 *apud* Fujimoto 1991, Ferscha 1995).

A principal diferença entre as aborgagens é o particionamento dos processos lógicos. Na abordagem SRIP os processos lógicos são particionados e

mapeados em um processador, comunicando-se com os outros através de mensagens. Nesse caso, é necessária a implementação de um protocolo que possa sincronizar os tempos de simulação, uma vez que os processos podem estar dependendo de eventos que ocorrem em outro processo(Bruschi e Santana 2002).

No MRIP as replicações de um mesmo programa são executadas independentemente em paralelo. Existe um analisador global que recebe os resultados de cada execução e faz as médias finais, se estas satisfazem a precisão, a simulação, então, é encerrada. Como por exemplo no cálculo do valor pi (π), existem algoritmos que fazem o cálculo distribuído, cada processador faz um cálculo para o valor e envia para um processador raíz. O processador raíz faz a média dos valores recebidos e, com base em um valor já estabelecido, calcula o erro (precisão), se o erro encontrado for menor, então o algoritmo para a execução.

2.3. Simulação Heterogênea

A modelagem de um sistema de simulação pode ser feita de duas maneiras: homogênea e heterogênea. A principal diferença entre elas está na implementação enquanto a homogênea utiliza apenas de uma linguagem de programação, a heterogênea utiliza mais de uma. Diferem também na suas interfaces ou nas suas tecnologias de construção (Mello 2009, Souza, Sperb, Mello e Wagner 2005). Entretanto, segundo Reynolds (1988), na simulação heterogênea existem alguns tópicos que devem ser tratados com atenção, tais como tolerância a falhas, reconfiguração dinâmica, requisitos de tempo real e abordagens otimizadas para o processo de sincronização e comunicação dos elementos da simulação.

Tolerância a falhas é a capacidade que o sistema possui de contornar e continuar sua execução mesmo tendo ocorrido uma falha. Reconfiguração

dinâmica envolve um conjunto de atividades, tais como balanceamento de carga, processamento de recursos, atrasos de comunicação, entre outros. Simulações são requeridas para executarem pelo menos em tempo real. Essa é uma afirmação nebulosa e existem várias interpretações, mas será assumido que isto significa representar fielmente como se acontece no mundo real com os elementos simulados. E por último é necessário que a comunicação seja otimizada, afinal não é desejado ter perda de desempenho (Reynolds 1988).

A principal vantagem da simulação heterogênea é também o seu principal problema: modelar cada módulo do sistema em uma linguagem específica e apropriada(Amory, Morais, Oliveira, Hessel e Calazans 2000). A vantagem ocorre por poder utilizar a melhor linguagem que se adeque ao módulo, mas apresentou problema por ser complexa a definição da semântica de interação.

2.3.1. Modelos Híbridos

Na simulação heterogênea distribuída existem basicamente três tipos de componentes em relação ao tempo: síncronos, assíncronos e componentes que não modelam explicitamente a passagem do tempo (denominados *untimed*) (Mello 2005). Os elementos síncronos evoluem no tempo de forma contínua, simulando a realidade, portanto não existe a possibilidade de retorno a um tempo passado. Diferentemente, os assíncronos podem realizar o chamado *rollback*, que é o retorno no tempo de simulação, e refazer a simulação (Higaki, Shima, Tachikawa e Takizawa, 1997).

A simulação híbrida no tempo é a simulação que suporta a utilização de diferentes elementos com diferentes noções temporais simultaneamente. Como por exemplo elementos conservadores e otimistas durante uma mesma simulação.

Na simulação síncrona (conservadora) a ordem do envio e recebimento

de mensagens deve ser garantida através da utilização de um protocolo que assegure isto. Para garantir a ordem de ocorrência dos eventos, no modo síncrono, o controle geralmente é mantido centralizado mediante o uso de um relógio global GVT (*Global Virtual Time*) (Ferscha 95). A sincronização e o funcionamento dos processos de envio e recebimento de mensagens serão explicados com mais detalhes na Seção 2.5.1.

A simulação assíncrona (otimista) dispende de maior atenção para a implementação de seus protocolos, pois estes permitem que eventos violem regras definidas no modo conservador. Por ser mais flexível, a simulação otimista é mais difícil de ser implementada.

Pouco se acha na literatura acerca dos elementos *untimed*. Eles organizam as mensagens recebidas de acordo com a ordem de seu recebimento (Mello 2005).

2.4. High Level Architecture

O D*epartment of Defense* (DoD) dos Estados Unidos da América em 1995 apresentou resultados de uma pesquisa realizada num processo de esforço conjunto envolvendo o governo, o ambiente acadêmico e a indústria sobre simulação interativa distribuída(Calvin e Weatherly 1996). Denominada de *High Level Architecture* (HLA), este resultado se tornou padrão pela IEEE em se tratando de simulação.

A arquitetura HLA foi inicialmente concebida dentro da comunidade de simulação interativa distribuída, considerando as necessidades especiais que foram observadas em treinamentos militares(Mello e Wagner 2001).

O HLA é um padrão IEEE e propõe mecanismos e regras que auxiliam na interoperabilidade de simuladores heterogêneos distribuídos(Amory, Morais, Oliveira, Hessel e Calazans 2000). Ele não define uma implementação específica, nem a utilização de qualquer software ou linguagem de programação

particular.

A arquitetura HLA fornece uma interface padronizada para simulações distribuídas e oferece suporte para o desenvolvimento de simulações baseadas em componentes, chamado de federados que trabalhando em conjunto formam uma federação(Sperb 2003).

2.4.1. Características do HLA

No HLA existe o conceito de federados, que podem ser considerados como uma extensão do conceito de objeto. Quando um conjunto de federados é combinado com a RTI (*RunTime Infrastructure Services* – Infraestrutura de Serviços em Tempo de Execução) é formada então uma federação. A RTI é responsável pelo controle de operações de troca de mensagens entre os federados e federações (Kuhl, Weatherly e Dahmann 2000, Amory, Morais, Oliveira, Hessel e Calazans 2000, Mello 2005).

Na Figura 2.1 podemos ver em alto nível como é a estrutura de uma federação do HLA.

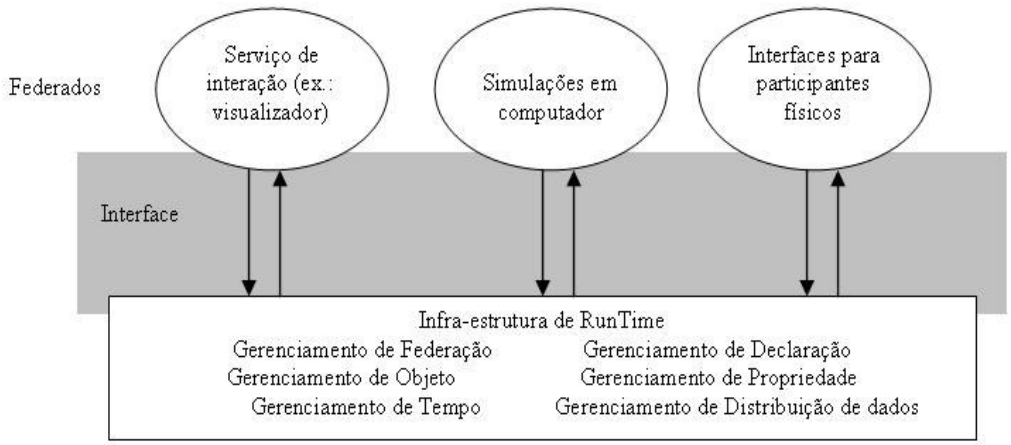

Figura 2.1: Visão funcional de uma federação.

Fonte: Mello, 2005.

Como pode ser observado na Figura 2.1, os federados não entram em contato diretamente com a RTI, mas sim através de uma interface.

2.5. Distributed Co-Simulation Backbone

O DCB tem o propósito geral de dar suporte à execução distribuída de modelos heterogêneos (Mello 2005). O DCB é distribuído em módulos, onde cada módulo executa tarefas específicas, aumentando a coesão e diminuindo o acoplamento.

No DCB os federados são chamados de elementos e estes participam de federações, numa relação de N elementos para 1 federação. Um elemento comunica-se apenas com seu Gateway, não tendo acesso, aos mecanismos internos de funcionamento do DCB. Portanto o Gateway faz a interface do elemento com os outros módulos (DCBS, DCBR, DCBK).

O DCB ainda é constituído por outros três módulos: o Expedidor do DCB (*DCB Sender* - DCBS), o Recebedor do DCB (*DCB Receiver* - DCBR) e o Núcleo do DCB(*DCB Kernel* - DCBK). A Figura 2.2 mostra a arquitetura do DCB.

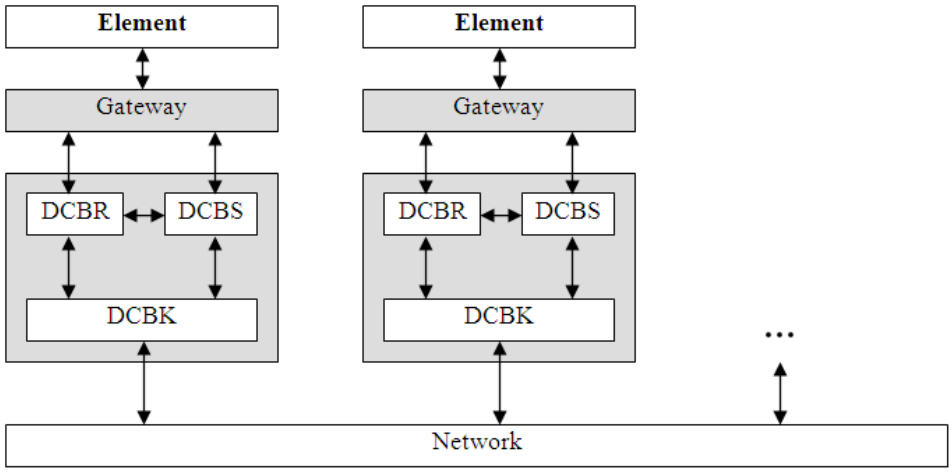

Figura 2.2: Arquitetura do DCB.

Fonte: Mello, 2009.

O DCBR gerencia mensagens recebidas de outros federados, sejam eles remotos ou locais. É ele quem decodifica os pacotes recebidos e participa das atividades de gerenciamento do tempo de simulação. Já o DCBS gerencia as mensagens emitidas pelo federado que ele representa. Em conjunto com o DCBR, também mantém gerenciamento do tempo virtual local (LVT - *Local Virtual Time*) e do modo de sincronização utilizado por cada elemento para cooperar com a federação (Mello 2005).

2.5.1. Sincronização no DCB

O DCB faz a sincronização das mensagens através do tempo de evento (*timestamp*), garantindo que o evento ocorra apenas neste tempo (Mello 2005). A ordem dos eventos externos é controlada por um tempo global (GVT - *Global Virtual Time*)(Mello 2005).

Elementos síncronos apenas poderão enviar mensagens para outros elementos síncronos se o *timestamp* da mensagem for maior do que o GVT, para garantir a não ocorrência de violação de tempo (Mello 2005). Elementos que não implementam o tempo, conhecidos como *untimed* organizam as mensagens de acordo com a ordem em que foram recebidas. Por último, elementos assíncronos não tinham sido implementados antes deste trabalho. Estes podem transitar no tempo de simulação, voltando à tempos passados e podendo ultrapassar o GVT.

Um federado foi criado para exercer a função de sincronização, o Fedgvt. Ele é responsável por calcular o valor do GVT utilizando os valores dos LVT's dos federados como parâmetro. Portanto os DCBR's enviam para o Fedgvt os seus LVT's ao tentarem avançar no tempo, e o Fedgvt retorna para eles o valor do novo LVT e se necessário atualiza o GVT(Mello 2005). O valor do GVT é calculado em cima do menor LVT adicionado um *lookahead* de 100 unidades de tempo, ou seja GVT = menor LVT + 100.

Porém um problema foi detectado durante a realização deste trabalho so

bre *checkpoints* no DCB. Ao realizar o *rollback* o elemento assíncrono se torna o menor LVT, com isso o GVT recua, e como o elemento síncrono não pode ultrapassar o GVT (o assíncrono e *untimed* podem) este também recua, o que de fato está errado. Foram feitos estudos e leituras sobre a sincronização de elementos conservadores e otimistas e os resultados serão apresentados na Seção 4.1.3.

2.5.2. Mensagens Nulas

Mensagens nulas são utilizadas para evitar situações de impasse (*deadlock*.) nos modelos síncronos(Mello 2005). Uma mensagem nula pode ser definida como uma promessa de um Processo Lógico, para os demais, de que ele não irá solicitar a execução de nenhum evento com tempo (futuro) menor que o valor fornecido na mensagem nula (Mello 2005 apud Fujimoto 2001).

2.5.3. Anti-mensagens

Anti-mensagens são mensagens que são enviadas quando ocorre violação de tempo (Ferscha 1996, Elnozahy et al, 1999*).* Estas mensagens são enviadas para os elementos que receberiam as mensagens num instante de tempo passado, para que eles realizem o *rollback* e possam receber estas mensagens.

2.6. Soluções para rollback

O *rollback* é o ato de retroceder no tempo. Isto ocorre quando há necessidade, devido a alguma violação de tempo. A violação no tempo é caracterizada quando um elemento tenta enviar uma mensagem para um outro elemento e o destinatário se encontra num tempo posterior ao do recebimento da mensagem. Portanto o elemento destinatário recebe uma anti-mensagem e realiza o chamado *rollback*, descartando as mensagens enviadas e recebidas neste intervalo de tempo perdido e posteriormente recebe a mensagem que gerou

o *rollback* e volta a evoluir no tempo, continuando a simulação. Nem todos elementos podem realizar o *rollback*, apenas os otimistas e os elementos *untimed*.

Na literatura foram encontradas dois principais métodos para realizar o *rollback* durante uma simulação: o método de guardar *logs* e o método de estabelecer *checkpoints*. O método escolhido neste trabalho para a implementação no DCB foi de criação e controle de *checkpoints*, pois se são mais fáceis de serem implementadas são menos restritivas (Elnozahy et al, 1999, Higaki, Shima, Tachikawa e Takizawa, 1997).

2.6.1. Estado consistente

Estado consistente de um sistema, é um estado em que não existe mensagens órfãs. Podem existir mensagens em trânsito, que são aquelas que já foram enviadas porém ainda não entregues no destino, seja porque o elemento que a recebeu ainda não chegou ao *timestamp* (ou tempo de evento) ou porque ainda está em trânsito na rede de comunicação (Elnozahy et al, 1999, Higaki, Shima, Tachikawa e Takizawa, 1997).

Timestamp é o tempo em que o elemento precisa alcançar para poder efetivamente receber uma mensagem específica, ou seja, um LVT. A mensagem pode estar armazenada no Embaixador do DCB porém ainda não entregue ao elemento pois este não alcançou o LVT que foi especificado pelo elemento emissor ao enviar esta mensagem.

Um problema na realização de *rollback* quando utilizados elementos síncronos e assíncronos, é o fato de que quando um elemento otimista voltar no tempo, ele pode perder uma mensagem que enviou para um elemento conservador, porém, o elemento conservador não pode retornar no tempo anterior ao recebimento desta mensagem. Portanto precisa ser feito um bloqueio à realização de *rollbacks* para que não se caracterize este tipo de estado

inconsistente.

2.6.2. Rollback baseado em log

O *rollback* baseado em *log* torna explícito o fato de que a execução de um elemento pode ser modelada como a sequência de intervalos de estados determinísticos, iniciando com a execução de um evento não-determinístico. Este evento pode ser o recebimento de uma mensagem de outro elemento, entretanto o envio de mensagem não é um evento não-determinístico (Elnozahy et al, 1999).

Os três principais tipos de protocolos para *rollback* baseado em *log* são:

- Rollback pessimista baseado em log: são protocolos que garantem que mensagens órfãs nunca são criadas durante uma falha. Estes protocolos simplificam a recuperação, coleta de lixo e a entrega da saída, porém com o custo do aumento do overhead para garantir o desempenho livre de falhas (Elnozahy et al, 1999);
- Rollback otimista baseado em log: são protocolos que reduzem o overhead para garantir o desempenho livre de falhas, porém deixa que sejam criadas mensagens órfãs durante uma falha. A possibilidade de se ter mensagens órfãs complica a recuperação, coleta de lixo e a entrega da saída (Elnozahy et al, 1999);
- Rollback causal baseado em log: são protocolos que combinam as vantagens do baixo overhead de desempenho e rápida entrega da saída, porém pode ser necessários recuperação e coleta de lixo complexo (Elnozahy et al, 1999).

Esses podem ser ditos os principais paradigmas de rollback baseado em log, apesar de que existem técnicas que melhoram o que cada paradigma peca (Elnozahy et al, 1999). Como não é o foco do trabalho, não serão apresentadas estas soluções.

2.6.3. Rollback baseado em checkpoints

Na ocorrência de uma falha, *rollback* baseado em *checkpoint* restaura o estado do sistema de acordo com o conjunto de *checkpoints* (Elnozahy et al, 1999). *Rollback* baseado em *checkpoints* são menos restritivos e mais simples de implentar do que o *rollback* baseado em *log*, porém não garantem que a execução antes da falha seja regenerada deterministicamente após um *rollback* (Elnozahy et al, 1999, Higaki, Shima, Tachikawa e Takizawa, 1997).

As técnicas de *rollback* podem ser classificadas em três categorias: *checkpoints* não-coordenados, *checkpoints* coordenados e *checkpoints* induzidos à comunicação (Elnozahy et al, 1999). Como estas divisões podem ser mais detalhadas e criadas subcategorias, serão apresentadas nas próximas seções de modo mais aprofundado.

2.7. Checkpoints não-coordenados

Checkpoints não-coordenados permitem aos processos o máximo de autonomia na decidisão de quando estabelecer *checkpoints*. A maior vantagem disso é que cada processo pode estabelecer *checkpoints* quando este achar mais conveniente (Elnozahy et al, 1999*)*. Os *checkpoints* não-coordenados também podem ser chamados de *checkpoints* assíncronos (Higaki, Shima, Tachikawa e Takizawa, 1997), porém neste trabalho serão abordados os não-coordenados.

Existem três principais desvantagens no *checkpoints* não-coordenados: o efeito dominó, os *checkpoints* inúteis e o problema de lixo. O efeito dominó pode causar a perda de grande quantidade de trabalho, e ainda há a possibilidade de voltar até o início da computação (Higaki, Shima, Tachikawa e Takizawa, 1997). Os *checkpoints* inúteis são aqueles que nunca farão parte de um estado consistente global, são indesejados pois aumentam o *overhead* e não contribuem no avanço da linha de recuperação. Por último, a não coordenação força cada

processo manter vários *checkpoints* e periodicamente invocar um algorítmo de coleta de lixo para coletar os *checkpoints* que não são mais úteis (Elnozahy et al, 1999).

2.7.1. Grafos de dependência e cálculo de linha de recuperação

Existem duas abordagens propostas na literatura para determinar a linha de recuperação no *rollback* baseado em *checkpoints*. Ambas resultam na mesma linha de recuperação, portanto são equivalentes (Elnozahy et al, 1999).

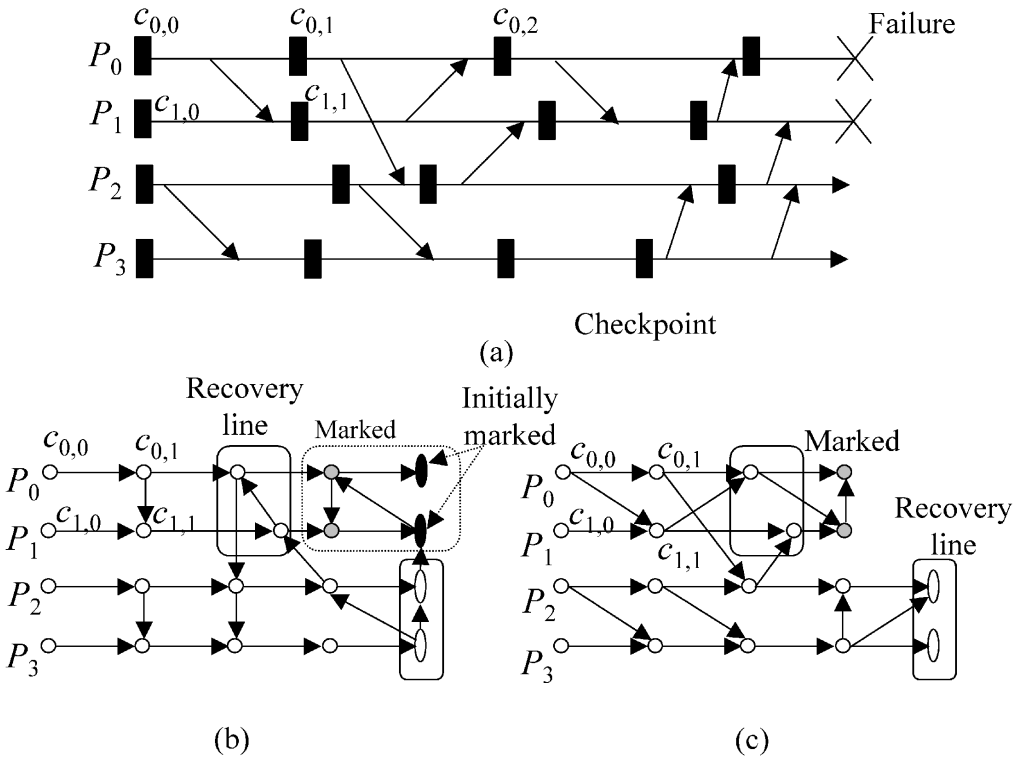

Figura 2.3: Exemplo de execução(a), grafo de dependência(b) e grafo de *checkpoints*(c).

Fonte: Elnozahy et al, 1999.

É construído um grafo onde os nós são os checkpoints e as arestas são

colocadas quando há troca de mensagens entre os elementos que obtém cada *checkpoint*, como mostrado na Figura 2.3.

Na Figura 2.3(a) é mostrada a execução da simulação, onde ocorrem falhas nos processos P_0 e P_1. As barras representam os *checkpoints* dos processos e as setas as trocas de mensagens. Na abordagem de grafo de dependência, essa execução será montada como mostrada na figura 2.3(b), onde se houver uma mensagem que saiu entre os *checkpoints* $c_{i,x-1}$ e $c_{i,x}$, onde *i* representa o número do processo e *x* representa o número *do checkpoint*,a aresta que corresponde à mensagem enviada sairá de $c_{i,x}$, e na linha do tempo do processo que recebeu a mensagem, entre os *checkpoints* $c_{j,y-1}$ e $c_{j,y}$, receberá a ponta da aresta em $cj,_y$.

Na abordagem de grafo de *checkpoints*, existe apenas uma mudança em relação ao grafo de dependência. Se houver uma mensagem que saiu entre os *checkpoints* $c_{i,x-1}$ e $c_{i,x}$, a aresta que corresponde à mensagem enviada sairá de $c_{i,x-1}$, ao invés de $c_{i,x}$, como se fosse no grafo de dependência. Portanto podemos observar o grafo de *checkpoints* na Figura 2.3(c).

A linha de recuperação, mostra os *checkpoints* que cada processo deve chegar para retornar a execução. Como a falha mostrada no exemplo não afeta os processos P_2 e P_3, estes, continuam sua execução normalmente, enquanto P_0 e P_1, realizam o *rollback* para os *checkpoints* circulados pela linha de recuperação.

2.7.2. Efeito dominó

O efeito dominó ocorre quando um elemento recebeu uma mensagem num instante de tempo passado e o elemento que enviou essa mensagem realiza *rollback* e volta para um tempo anterior ao envio desta mensagem, portanto ela se tornou uma mensagem órfã. Com isso há a necessidade do primeiro elemento retroceder num tempo anterior ao recebimento desta mensagem para poder descartá-la, e consequentemente poderão surgir outras mensagens órfãs gerando outros *rollbacks* (Higaki, Shima, Tachikawa e Takizawa, 1997).

O efeito dominó é mostrado na Figura 2.4, ele ocorre quando a retroação de um processo faz outros retroagirem, e ao passo que os outros retroagem, acontece um fenômeno em cascata que faz com que todos processos continuem retroagindo até o início da simulação, não utilizando os *checkpoints* salvos (Elnozahy *el al*).

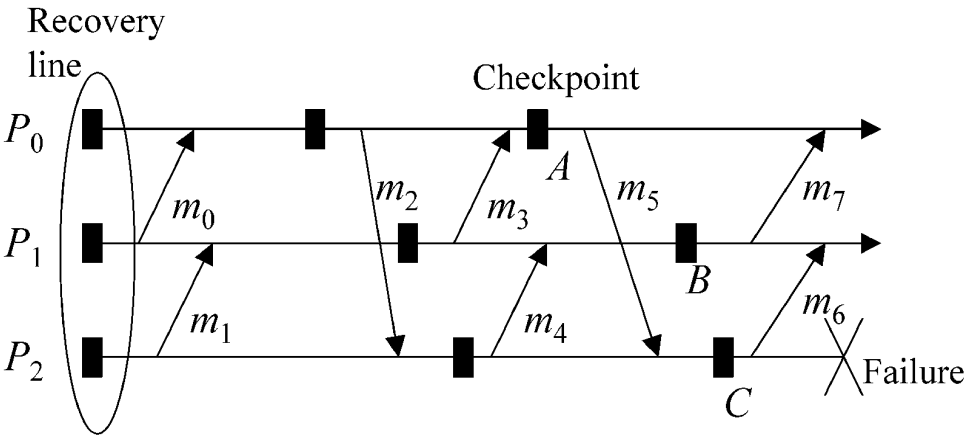

Figura 2.4: *Rollback*, linha de recuperação e efeito dominó.

Fonte: Elnozahy *el al*, 1999.

Na Figura 2.4 ocorre uma falha após o envio de m_6, fazendo com que P_2 volte para *C*. Porém como m_6 foi perdida, P_1 tem que retornar para *B* e consequentemente m_7 também se torna uma mensagem órfã, fazendo com que P_0 retorne para *A*. Podemos observar que este fenômeno ocorrerá, no exemplo, sempre que um elemento atingir um *checkpoint*, fazendo com que outro se torne inconsistente pelo fato de ter uma mensagem órfã. Portanto este exemplo irá desencadear no *rollback* dos processos até o primeiro *checkpoint*, no início da simulação.

2.8. Checkpoints coordenados

Checkpoints ordenados necessitam que os processos orquestrem seus checkpoints a fim de formar um estado consistente global. *Checkpoints*

coordenados simplifica a recuperação e não é suscetível ao efeito dominó, pois todos processos reiniciam do *checkpoint* mais recente. Também fazem com que cada processo mantenha apenas um *checkpoint* permanente, reduzindo o *overhead* de armazenamento e eliminando a necessidade de coleta de lixo (Elnozahy et al, 1999). Os *checkpoints* coordenados também podem ser encontrados na literatura como *checkpoints* síncronos, apesar de que neste trabalho serão apenas chamado de *checkpoints* coordenados.

2.8.1. Coordenação não bloqueante de checkpoints

Prevenir um processo de receber mensagens da aplicação que possam tornar o *checkpoint* inconsistente é um problema fundamental dos *checkpoints* coordenados. Portanto é necessário que haja uma sincronização na hora de estabelecer os *checkpoints*, pois caso contrário, podem ser estabelecidos *checkpoints* inconsistentes (Elnozahy et al, 1999).

Checkpoints inconsistentes são caracterizados pela recepção de uma mensagem m_0 antes do estabelecimento do *checkpoint*, e essa mensagem recebida, foi enviada após o *checkpoint* do processo emissor, ou seja, caso o processo emissor realize um *rollback*, o processo receptor terá que realizar um *rollback* antes do *checkpoint*, pois senão m_0 será uma mensagem órfã.

Algumas estratégias para que isso não ocorra são:

- Considerando que os canais são FIFO (*First In First Out*), o problema pode ser evitado pois a requisição para o estabelecimento de *checkpoints* chegará antes de alguma mensagem, e o processo é forçado a estabelecer este *checkpoint*;
- Se os canais não são FIFO, o iniciador (processo que coordena o estabelecimento de *checkpoints*) envia em *broadcast* um marcador para todos processos, cada processo vai estabelecer seu *checkpoint* e enviá-lo para todos processos em *broadcast*, cada processo

estabelece um *checkpoint* antes de receber uma mensagem da aplicação, portanto não existira mensagens órfãs.

2.8.2. Coordenação mínima de checkpoints

Checkpoints coordenados requerem que todos processos participem em todos *checkpoints*. Porém é desejado reduzir o número de processos envolvidos numa Seção de *checkpoints* coordenada. Isto pode ser feito, pois os processos que necessitam de novos *checkpoints* são apenas aqueles que comunicaram com o iniciador diretamente ou indiretamente desde o último *checkpoint (*Elnozahy et al, 1999).

O protocolo seguinte, de duas fases que realiza a coordenação mínima de *checkpoints*. Durante a primeira fase, o iniciador identifica todos os processos os quais se comunicou desde o último *checkpoint* e os envia uma requisição. Cada processo, ao receber a requisição, identifica todos os processos que se comunicou desde o último *checkpoint* e envia para eles uma requisição, e assim sucessivamente, até que nenhum processo possa ser identificado. Durante a segunda fase, todos os processos identificados na primeira fase estabelecem um *checkpoint*. O resultado é um *checkpoint* consistente que envolve apenas os processos participativos. Neste protocolo, após o processo estabelecer o *checkpoint* ele não pode enviar nnehuma mensagem até que a segunda fase termine com sucesso, porém pode receber mensagens (Elnozahy et al, 1999).

2.9. Checkpoints induzidos à comunicação

Checkpoints induzidos à comunicação evitam o efeito dominó sem a necessidade de coordenação dos *checkpoints*. Nestes processos existem dois tipos de *checkpoints*, local e forçado.

Checkpoints locais podem ser estabelecidos independentemente, enquanto os forçados devem ser estabelecidos para garantir o processo eventual

da linha de recuperação. Os protocolos para estabelecimento de *checkpoints* forçados previnem a criação de *checkpoints* inúteis, que nunca farão parte de um estado consistente global. *Checkpoints* inúteis não são desejáveis, pois além de não contribuírem para a linha de recuperação do sistema em caso de falhas, também consomem recursos e causam *overhead* de desempenho (Elnozahy et al, 1999).

3. METODOLOGIA

A pesquisa deste trabalho é classificada da seguinte forma: **Quanto à natureza** é uma pesquisa tecnológica, pois os conhecimentos adquiridos foram utilizados para aplicação prática durante a implementação efetiva de elementos assíncronos suportados pelo DCB. **Quanto aos objetivos** é uma pesquisa aplicada pois teve como finalidade a busca de solução para problema concreto, registro e análise do comportamento de *checkpoints*. **Quanto aos procedimentos** é uma pesquisa experimental pois buscou a descoberta de novos métodos através de ensaios e estudos em laboratório, visando o controle de algumas variáveis que poderiam intervir no experimento. Entende-se como pesquisa em laboratório aquela onde ocorre a possibilidade de se controlar as variáveis que possam intervir no experimento.

Primeiramente foi resgatado o que foi feito pelo autor em um trabalho de iniciação científica entitulada “Especificação e prototipação do conceito de anti-mensagens para o suporte à operações de rollback no DCB”.

Posteriormente foram estudadas soluções algorítmicas para a localização e estabelecimento de *checkpoints*. Foi feito também um estudo de viabilidade dessas soluções algorítmicas em relação às características do DCB. Iniciando, portanto, a revisão bibliográfica do trabalho.

Após este estudo foram feitas a especificação e a implementação das soluções estudadas seguidas por uma bateria de teste para o seu aperfeiçoamento. Para a validação das modificações do DCB foi feito um estudo de caso utilizando elementos conservadores, otimistas e *untimed*.

Finalizando o trabalho, foram aferidas e apresentadas conclusões acerca do estudo sobre *checkpoints* em conjunto com a arquitetura DCB.

4. RESULTADOS

Este capítulo apresenta os resultados do trabalho de especificação e implementação dos mecanismos de controle e estabelecimento de *checkpoints* no DCB.

Na Seção 4.1 e suas sub-seções são apresentados os principais problemas encontrados durante o desenvolvimento do trabalho seguidos de suas soluções.

Já as Seções 4.1.6 e 4.1.7 mostram o foco deste trabalho: *rollback* e *checkpoints*. São apresentadas as soluções para gerenciamento de elementos otimistas no DCB.

Finalizando, na Seção 4.2 é apresentado o estudo de caso utilizado com propósito de validação.

4.1. Problemas e especificações do DCB

Checkpoints são marcas estabelecidas no tempo de simulação por elementos. Estas marcas são utilizadas pelo próprio elemento caso ele necessite voltar no tempo, portanto ele só pode voltar para os tempos correspondentes aos *checkpoints*. Entretanto, a função dos *checkpoints* vai além, pois eles também devem representar um estado seguro para o elemento. O elemento pode retroagir para este *checkpoint* e mesmo que dispare anti-mensagens para outros elementos, a idéia principal é que ele mesmo após disparar anti-mensagens não receba nenhuma anti-mensagem para retroceder para um *checkpoint* anterior ao atual.

A implementação dos *checkpoints* só estará completa com a identificação de estados seguros, que é um trabalho que está sendo desenvolvido em paralelo. Com isso os mecanismos implementados por este trabalho se tornarão mais eficientes, como será exemplificado nas próximas seções.

Os *checkpoints* não-coordenados foram escolhidos como solução para a implementação no DCB. A escolha do tipo de *checkpoint* se deu pois, apesar de suas desvantagens citadas na Seção 2.7, não é necessário comunicação com outros processos para o estabelecimento de *checkpoints*. Isso implica em redução no *overhead* de mensagens, e por ser um sistema que pode ser distribuído fisicamente, isto se torna essencial para garantir um bom desempenho (Higaki, Shima, Tachikawa e Takizawa, 1997).

Com *checkpoints* não-coordenados, os elementos ficam responsáveis por estabelecer seus próprios *checkpoints*. Portanto em detrimento dos *checkpoints* coordenados, os quais possuem um elemento que fica com a responsabilidade de sincronizar todo o processo, nos *checkpoints* não-coordenados cada elemento tem que possuir na sua implementação algoritmos que estabeleçam e gerenciem os *checkpoints* (Elnozahy et al, 1999, Higaki, Shima, Tachikawa e Takizawa, 1997).

Na implementação dos *checkpoints* não-coordenados no DCB, os elementos ficam apenas com a responsabilidade de estabelecimento dos *checkpoints*, pois o *rollback* é fruto de uma cooperação entre o elemento, através do *Gateway*, e os módulos DCBS e DCBR, que será explicada com maior detalhamento na próxima Seção.

4.1.1. Problema do efeito dominó

O tratamento para o efeito dominó já estava implementado no DCB. No DCBS e no DCBR encotram-se estruturas para o armazenamento de mensagens. O DCBS controla as mensagens enviadas e o DCBR as recebidas. O descarte de mensagens recebidas é de responsabilidade do elemento, pois seu comportamento ao descartar as mensagens recebidas não está ligado ao DCB, apesar de estarem armazenadas no DCBR. Portanto esta é uma obrigação dos elementos assíncronos no DCB, tratar as mensagens recebidas a serem

descartadas durante um *rollback*. Os elementos podem ter acesso as mensagens recebidas através do *Gateway* chegando ao DCBR. O DCBS é responsável por detectar as mensagens que se tornarão órfãs durante *o rollback*, e disparar anti-mensagens para os elementos que receberam estas mensagens, para retornarem a um tempo anterior de seu recebimento.

Este problema de efeito dominó é inevitável em se tratando de simulações de elementos otimistas, porém o DCB é capaz de aceitar o efeito dominó e prosseguir a simulação.

4.1.2. Problema de *checkpoints* inúteis e inconsistência

O problema de *checkpoint*s inúteis não foi considerado neste trabalho, pois está sendo feito um trabalho relacionado, que analisa estados consistentes. A identificação de estados consistentes, facilita a eliminação de *checkpoints* inúteis e consequentemente o lixo que é o acúmulo desses *checkpoints*.

Para a identificação de um *checkpoint* inútil é necessário saber se o tempo no qual o *checkpoint* se encontra possui um estado considerado consistente para o respectivo elemento. Portanto, primeiro se faz necessária a análise da consistência do estado da simulação no tempo em que se encontra o *checkpoint*. Na Figura 4.1 é mostrado um caso em que isto ocorre. *Async* representa elementos assíncronos, *sync* elementos síncronos e *GVT* corresponde ao sentido crescente da linha do tempo de simulação controlada pelo GVT.

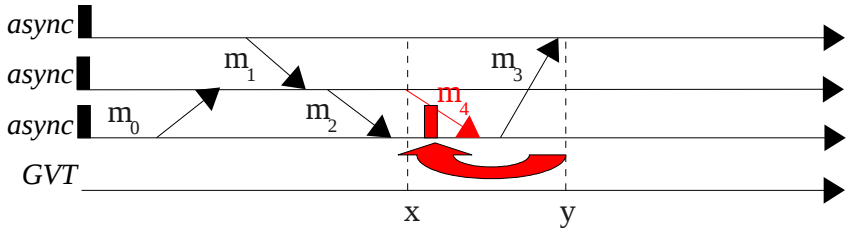

Figura 4.1: Exemplo de checkpoint inútil.

Fonte: Desenvolvido pelo autor.

Com o recebimento da mensagem m_4, o elemento que possui o *checkpoint* marcado em vermelho, que se encontra no tempo *y*, terá que realizar o *rollback* até este ponto, descartando a mensagem m_3. Ao descartar m_3, o primeiro elemento, que também se encontra no tempo y, terá que retornar ao último *checkpoint* colocado antes de m3, que se encontra no início da simulação, descartando assim m_1. Com m_1 descartada, o segundo elemento, que se encontra no tempo *x*, tem que voltar ao *checkpoint* anterior, ou seja, o *checkpoint* 0, com isso m_2 é também descartada. Fazendo com que também o terceiro elemento retorne ao início da simulação, descartando m_0. Portanto o *checkpoint* marcado de vermelho não contribuiu para a simulação, pode ser considerado como inútil.

Caso um *checkpoint* seja inconsistente, quando um elemento precisar retornar a ele, ao eliminar as mensagens enviadas neste período irá desencadear um efeito dominó que fará o próprio elemento retornar a um *checkpoint* num tempo anterior. Com isso o primeiro *checkpoint* não contribuiu em nada na simulação, pelo contrário, mais processamento foi gasto desnecessariamente, portanto ele pode ser eliminado sem prejuízo para a simulação.

A Figura 4.2 é um exemplo de inconsistência no envio de mensagens.

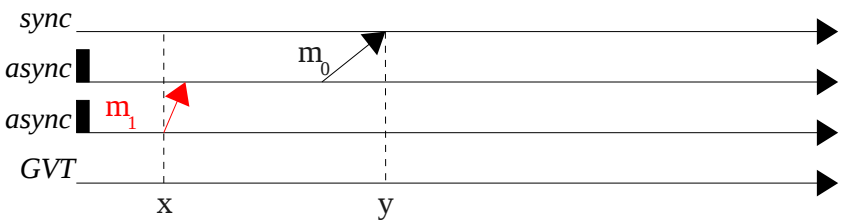

Figura 4.2: Exemplo de inconsistência.

Fonte: Desenvolvido pelo autor.

O terceiro elemento, que se encontra no tempo *x*, envia m_1 para o segundo elemento. Ao receber m_1, o segundo elemento, que se encontra no tempo *y*, precisa descartar m_0, porém m_0 foi enviada para um elemento síncrono, e este elemento não pode simplismente realizar um *rollback* para um tempo

anterior a *y* e descartá-la. Portanto a mensagem m_1 não pode ser entregue ao segundo elemento, não pode haver nenhum *checkpoint* anterior a m_0. Pois mesmo que exista algum *checkpoint*, este será inútil, será inalcançável e não contribuirá para a simulação. Então este *checkpoint* pode ser removido sem prejuízo para a simulação.

Este é um problema a ser analisado através da análise de estados seguros. Quando o tratamento de estados seguros estiver concluído, o DCB suportará de maneira eficiente uma simulação que contenha elementos otimistas.

4.1.3. Problemas na sincronização

Durante a implementação do estudo de caso ocorreram alguns problemas para sincronizar elementos otimistas, conservadores e *untimed p*or causa do *rollback* dos elementos otimistas.

Quando um elemento otimista retorna no tempo, ele deve atualizar seu LVT para um instante passado, porém ao fazer isso automaticamente o Fedgvt colocava-o na lista de LVTs e este novo LVT acabava sendo o menor. Portanto, um novo cálculo do GVT era realizado utilizando como base este LVT (GVT = menorLVT + 100). Com isso o GVT retornava no tempo forçando o elemento conservador a voltar também, pois este não pode ultrapassar o valor do GVT. Como demostrado na Figura 4.3.

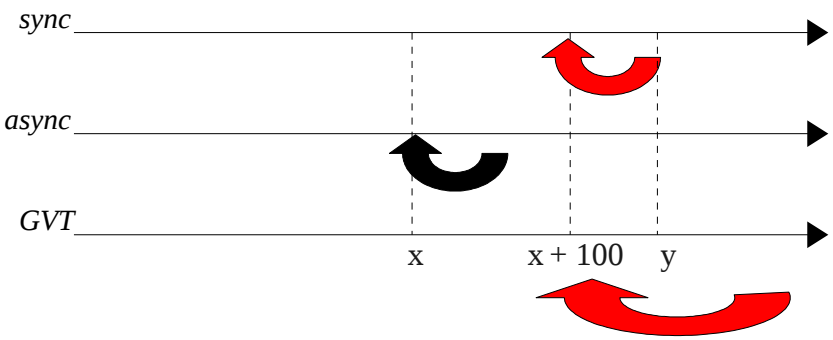

Figura 4.3: Problema no rollback.

Fonte: Desenvolvido pelo autor.

Na Figura 4.3 o elemento assíncrono identificado como *async* realiza um *rollback* e volta para o tempo *x*, se tornando assim o menor LVT da simulação, pois o elemento síncrono se encontra no tempo *y* que é maior que *x*. Portanto o GVT volta para o menor LVT (*x*) + 100, forçando o elemento síncrono a voltar para este tempo também, pois o mesmo não pode ultrapassar o valor do GVT. Porém o elemento síncrono não pode voltar no tempo.

A solução encontrada para tal problema, foi deixar os elementos otimistas fora do cálculo do GVT, assim como os elementos *untimed* já faziam. Portanto o cálculo do GVT pelo Fedgvt se baseia apenas nos elementos conservadores, deixando os elementos otimistas e *untimed* com a liberdade necessária para transitarem no tempo passado. Apesar disso, o GVT ainda se torna referência no momento de envio de mensagens para elementos conservadores, para evitar violações de tempo.

4.1.4. Anti-mensagens

As anti-mensagens foram implementadas pouco antes deste trabalho, num trabalho de iniciação científica desenvolvido pelo autor. As anti-mensagens foram feitas com estrutura igual ao de uma mensagem comum do DCB, possuindo os seguintes principais campos:

- *FederationSource*: que representa o número da federação de onde a anti-mensagem saiu;
- *FederateSource*: que representa o número do elemento que disparou a anti-mensagem;
- *FederationDestination*: que representa a federação destino;
- *FederateDestination*: que representa o elemento destino;
- *AttributeID*: que representa o número do atributo de destino, que em todos elementos otimistas corresponde a 444.3;

- *Value*: valor da mensagem, ou seja o corpo do texto;
- LVT: *timestamp* o qual o federado deve retornar;
- *Operation*: vem entitulada *AntiMessage*;

Com isso obtém-se uma redução no *overhead* de comunicação, pois ao invés de enviar duas mensagens, primeiramente uma anti-mensagem apenas para avisar o retorno no tempo e outra com o corpo do texto, é somente enviado uma mensagem com o LVT de retorno juntamente com o corpo da mensagem.

Portanto a anti-mensagem é uma mensagem comum só que identificada de forma diferente. Ao receber uma anti-mensagem, esta mensagem vai para a primeira posição na estrutura que armazena as mensagens. Ao atingir o *timestamp* das mensagens armazenadas na estrutura, são entregues para o elemento e removidas da estrutura. Porém, a anti-mensagem é retirada logo após o seu armazenamento e enviada para o *Gateway* do elemento.

O *Gateway* do elemento recebe anti-mensagem e a identifica através do valor de seu atributo que foi escolhido para todos elementos otimistas como 444.3. Em seguida inicia o processo de retorno no tempo, o *rollback*.

4.1.5. Rollback

O *rollback* no DCB é uma ação conjunta do elemento e alguns módulos do DCB. Os módulos em questão são o DCBS e o DCBR. São acessados pelo elemento através do *Gateway*. Os módulos são imprescindíveis, pois neles se encontram armazenadas as mensagens enviadas e recebidas, respectivamente, e o DCBS é o responsável pela detecção de mensagens órfãs e disparo de anti-mensagens.

O *Gateway* é a parte do DCB que tem contato direto com o elemento, o único que pode chamar métodos do elemento. Ao identificar uma anti-mensagem ele inicia o processo de *rollback* no elemento. Inicia o processo de *rollback* parando a evolução do tempo do elemento. Após isso, é escolhido pelo

elemento o *checkpoint* de acordo com o LVT contido na anti-mensagem. Após isso o DCBS dispara as anti-mensagens, caso necessário. O elemento então inicia o processo interno de descarte das mensagens recebidas. Na Figura 4.4 é mostrado detalhadamente o como acontece esse processo.

```
1.        A0 = DCBR.getAttributeReceived("444.3");
2.        if (A0 != null) {
3.           DCBR.AttributeRemove(A0);
4.           Fed.rollback = true;
5.           checkpoint = Fed.getCheckpoint(A0.LVT);
6.           if ( checkpoint != null) {
7.              DCBS.antiMessageTrigger(checkpoint);
8.              Fed.rollback(checkpoint);
9.              Fed.setChatLVT(updateLVT(A0.LVT));
10.             Fed.setReceivedText(A0.Value);
11.             Fed.rollback = false;
12.          }
13.       }
```

Figura 4.4: Rollback no DCB.

Fonte: Desenvolvido pelo autor.

Na Figura 4.4 é mostrada a implementação que cada *Gateway* de cada elemento otimista possui para realizar o *rollback*. Cada elemento deve possuir uma variável pública chamada *rollback*, é uma variável que indica se o elemento está ou não em processo de *rollback*. Isto se faz necessário, pois durante um *rollback* a evolução no tempo é interrompida enquanto o *rollback* é executado. Na linha 4 a variável *rollback*, que é um *flag* para interromper a evolução no tempo do elemento, é sinalizada como *true*, portanto terá início o processo de *rollback*, e após o término do *rollback* ela volta a ter o valor *false* (linha 11) e o elemento volta a avançar no tempo.

Ainda no trecho de código da Figura 4.4 pode-se observar a atribuição da mensagem a ser entregue para uma variável auxiliar na linha 1, e sua efetiva remoção na linha 3.

No elemento é realizada uma busca em todos *checkpoints* (linha 5), procurando o maior valor de *checkpoint* menor que o *timestamp* da mensagem. Se houver um valor, o rollback continua (linha 6), caso contrário é interrompido e a mensagem recebida é descartada sem ter sido entregue ao elemento. Se encontrou um *checkpoint*, o DCBS inicia o processo de disparo de anti-mensagens para os elementos que possuiriam mensagens órfãs após o *rollback (*linha 7). Na linha 8 é onde o elemento fica responsável por descartar as mensagens recebidas após o *checkpoint* que lhe é passado como parâmetro.

Após realizar o *rollback*, o elemento tenta evoluir para o tempo da mensagem A_0 (linha 9) para poder recebê-la (linha 10).

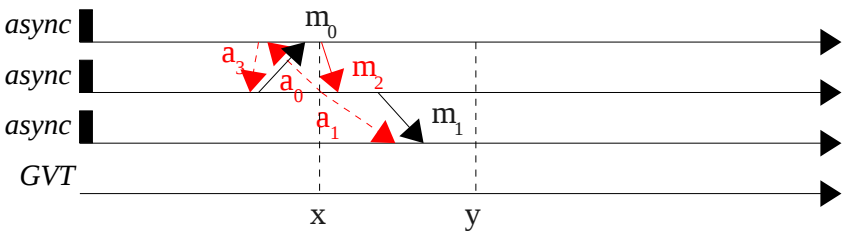

Figura 4.5: Rollback e anti-mensagens.

Fonte: Desenvolvido pelo autor.

Na Figura 4.5 é mostrado como ocorre o *rollback*. Utilizando o exemplo com três elementos otimistas, o segundo envia duas mensagens, m_0 para o primeiro elemento e m_1 para o terceiro elemento. Após receber m_0, o primeiro elemento envia uma mensagem m_2 num tempo passado do segundo elemento, que se encontra agora no tempo y, portanto ele inicia o processo de *rollback* e dispara duas anti-mensagens (a_0 e a_1) para os elementos que receberam suas mensagens m_0 e m_1 para eles então as discartarem, pois se tornarão mensagens

órfãs. No exemplo da figura, a mensagem m_2 também se torna órfã no momento em que o primeiro elemento realizar o *rollback*, portanto este dispara uma anti-mensagem (a_3) para o segundo elemento para que ele descarte a mensagem m_2. Por fim, este exemplo resulta no descarte de todas mensagens enviadas.

O elemento que realizou o *rollback* recebe do *Gateway* a mensagem que gerou o *rollback* e volta a evoluir no tempo da mesma forma que antes do início do *rollback*, como mostrado na Figura 4.4.

4.1.6. Checkpoints

A escolha de *checkpoints* não-coordenados em detrimento dos *checkpoints* coordenados foi feito, além das vantagens comentadas nas seções anteriores, principalmente para garantir consistência na estrutura do DCB, que não precisou ser alterada de forma significativa. Uma das mais drásticas mudanças no uso de *checkpoints* coordenados seria a alteração do Fedgvt para que este controlasse e sincronizasse o processo de estabelecimento de *checkpoint*. Isto inclui a interrupção momentânea da simulação para evitar a dessincronização dos *checkpoints*, o que poderia ser negativo, principalmente para os elementos síncronos.

Os elementos otimistas devem estabelecer o primeiro *checkpoint* no início da simulação, no tempo 0. Nada é salvo, pois ainda não existem mensagens recebidas. É o registro de uma marca que representa o início da simulação. Isto garante que no pior caso, a simulação não precisa ser interrompida caso um elemento necessite retroagir descartando tudo que executou durante a simulação, como exemplificado na Figura 4.4. Portanto, ele apenas volta para o *checkpoint* situado no tempo 0 e a simulação continua. Isso torna a implementação de *checkpoints* e a simulação tolerante a situações que possam gerar esse *rollback* (Elnozahy et al, 1999).

A maior vantagem de um *checkpoint* não-coordenado é que o elemento

tem autonomia de estabelecer *checkpoints* para um comportamento interno. Portanto, quando for identificada uma quantidade de informação considerada razoável, ele pode estabelecer ali um *checkpoint* (Elnozahy et al, 1999). Na literatura não é encontrado o que seria uma quantidade razoável.

4.1.7. Nova estrutura do DCB

Para explicar o funcionamento das novas estruturas do DCB, foi desenvolvido um diagrama de classes, apresentado como Figura 4.6. O diagrama real foi simplificado para mostrar as partes de maior pertinência para este trabalho.

O DCB possui uma classe que centraliza as operações, que é a classe ApplicationDCB. Esta classe é a responsável pela criação de um objeto de todas as outras, o DCB (objeto NewDCB), DCBS (objeto NewDCBS), DCBR (objeto NewDCBR) e Gateway (objeto NewGateway). Os módulos não se comunicam diretamente, mas sim através da classe ApplicationDCB (com o objeto denominado App).

O diagrama representa parte do estudo de caso desenvolvido que será explicado na Seção 4.2. Pode ser observado uma classe chamada Gateway5 que tem acesso diretamente ao chat através de um objeto chamado Fed. Isto ocorre, pois cada elemento necessita do seu próprio Gateway, ou seja, este é o Gateway do elemento número 5, entretanto, todos seus Gateways possuem as mesmas funcionalidades, apenas a executam de maneira particular para cada objeto especificamente.

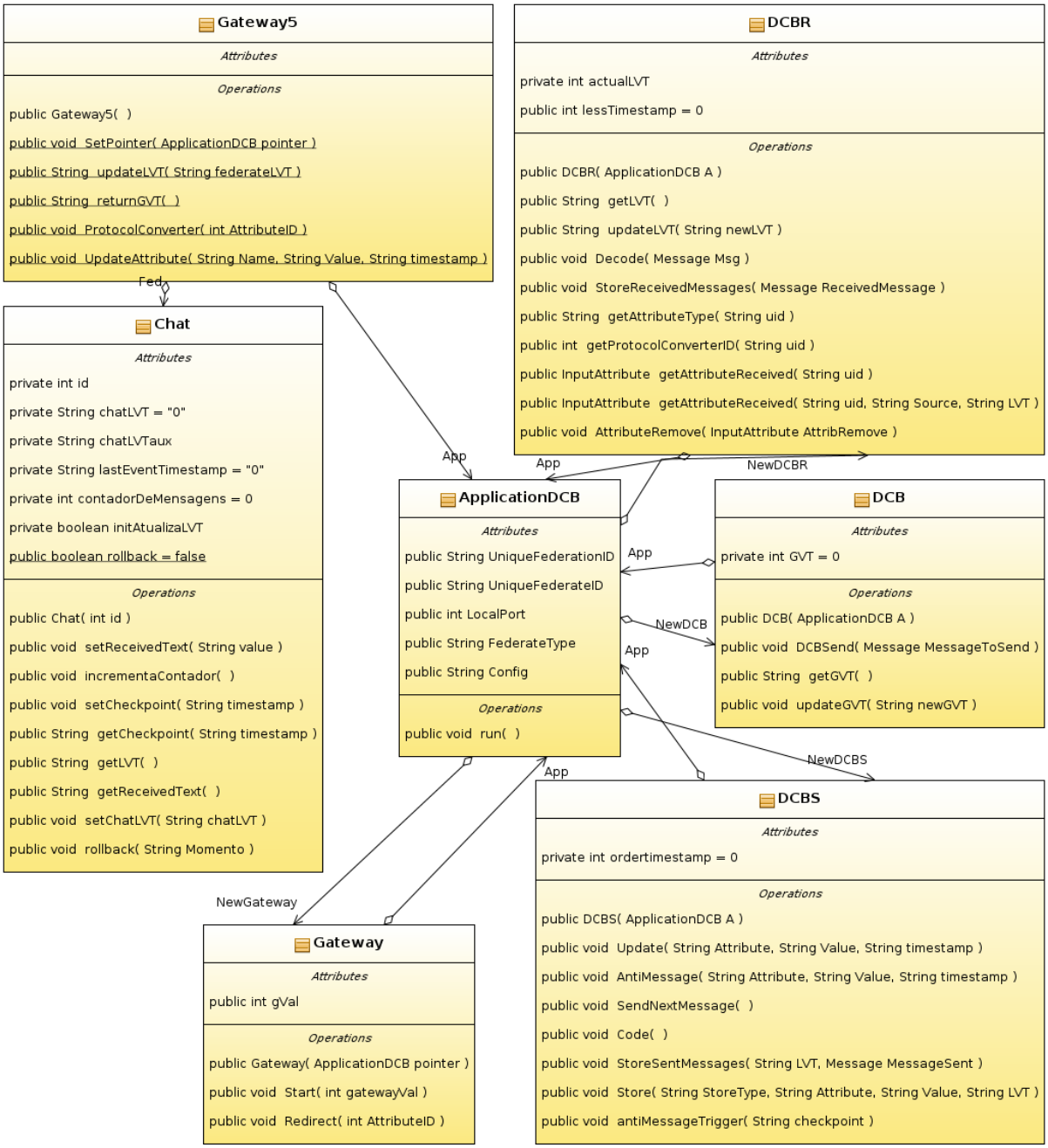

Figura 4.6: Estrutura do DCB.

Fonte: Desenvolvido pelo autor.

O objeto NewGateway (do tipo Gateway) identifica o número do elemento que está sendo carregado (atributo gVal) e cria seu Gateway específico, no caso, Gateway5.

O Gateway5 através do objeto App tem acesso aos mecanismos do DCBS e o DCBR. O DCBS fica envolvido no procedimento de envio de mensagem. A mensagem toma corpo (armazenando informações como federação e elemento destinatário, federação e elemento remetente , conteúdo da mensagem, *timestamp* da mensagem, entre outros) dentro deste módulo e é enviada para o DCB para que este envie para o destinatário, utilizando a rede ou localmente. O elemento recebe a mensagem dentro do seu módulo DCB (para cada elemento, TODOS os módulos essenciais são carregados, somente o GatewayX é apenas um para cada) e posteriormente é entregue ao módulo DCBR. O DCBR vai desmontando a mensagem, retirando as informações não mais necessárias (tais como federação e elemento destinatário, entre outros),e a entrega para o Gateway repassar ao elemento.

4.2. Estudo de caso

Foi desenvolvido um estudo de caso para observar o comportamento de elementos otimistas em uma simulação híbrida.

Neste estudo de caso foram desenvolvidos cinco elementos e incorporados ao DCB: um síncrono, três assíncronos e um *untimed*. Os elementos foram dispostos desta maneira para que pudesse ser construído um quadro de simulação complexo, pois envolve todos os tipos de elementos numa mesma simulação.

Foram escolhidos três elementos assíncronos para representar situações de *rollback*, restauração de *checkpoints* e efeito dominó. Porém, como em uma simulação híbrida não existe apenas elementos otimistas, foram incorporados outros dois elementos, um conservador e um *untimed*. Com isso, problemas de

sincronização e *rollback* foram representados.

Para testes e validação dos mecanismos de *checkpoint* e *rollback* no DCB foi implementado um estudo de caso chamado ChatDCB.

Foram construidos elementos com diferentes noções temporais para comunicarem entre si e dessa forma vieram à tona problemas inesperados, como apresentados na Seção 4.1.3.

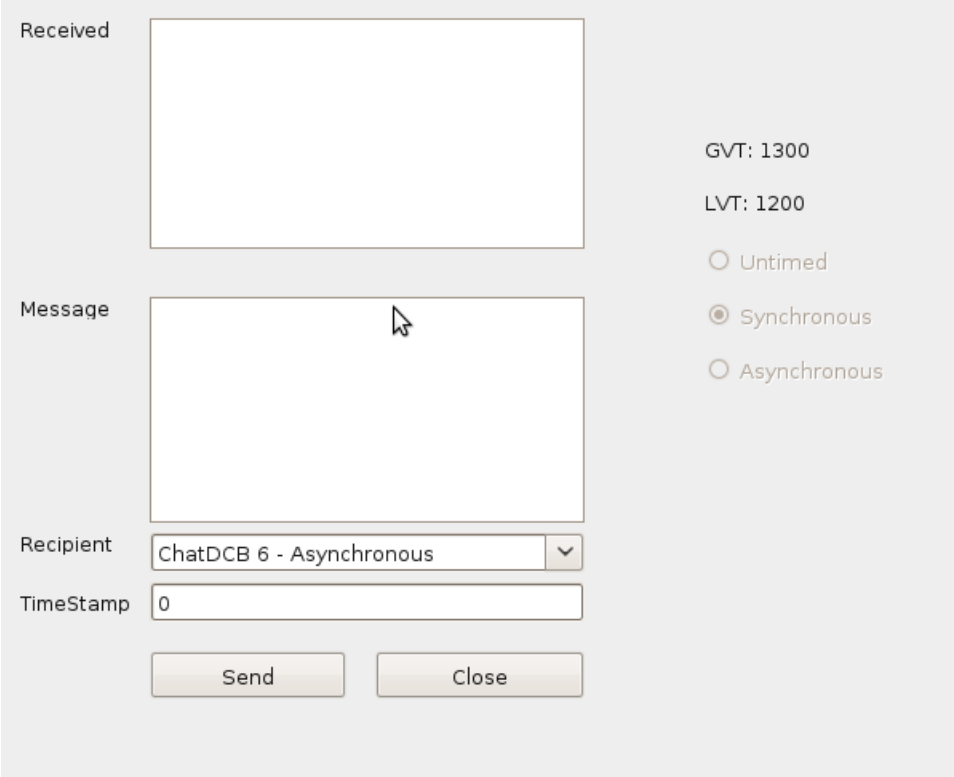

Figura 4.7: Exemplo de elemento síncrono.

Fonte: Desenvolvido pelo autor.

A Figura 4.7 mostra o elemento conservador elaborado no estudo de caso. Ao lado da parte denominada *Received* estão as mensagens recebidas pelo elemento. Ao lado de *Message* a mensagem para o usuário digitar. No *combo*

box denominado *Recipient* (destinatário) existem os elementos que podem receber mensagens. O campo denominado *TimeStamp* deve ser preenchido com o tempo o qual o outro elemento deve chegar (LVT) para receber a mensagem enviada.

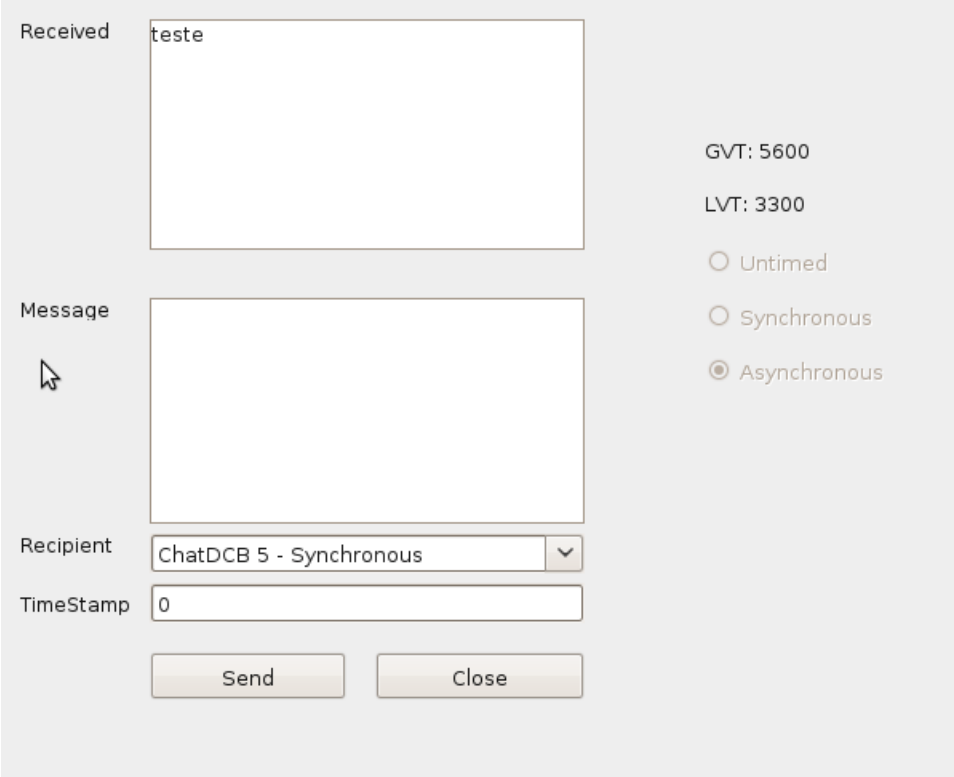

Figura 4.8: Exemplo de elemento assíncrono.

Fonte: Desenvolvido pelo autor.

O botão *Send* envia a mensagem caso o elemento receptor não seja um elemento síncrono e o *timestamp* não seja menor que o GVT, isso é uma violação de tempo. O elemento não pode ter acesso direto ao valor do GVT, porém como pode ser observado na figura, ele se encontra lá. Isto se deu pois era necessário saber o comportamento do GVT durante *rollbacks*. Abaixo do valor

do GVT encontra-se o valor do tempo do elemento, o seu LVT.

Por último tem-se a identificação automática da noção temporal do elemento, *Synchronous* (síncrono), *Asynchronous* (assíncrono) e *Untimed*. Esta identificação se dá no tempo em que o elemento está sendo carregado, e carregada sua configuração, que se encontra em um arquivo de configuração na linguagem XML.

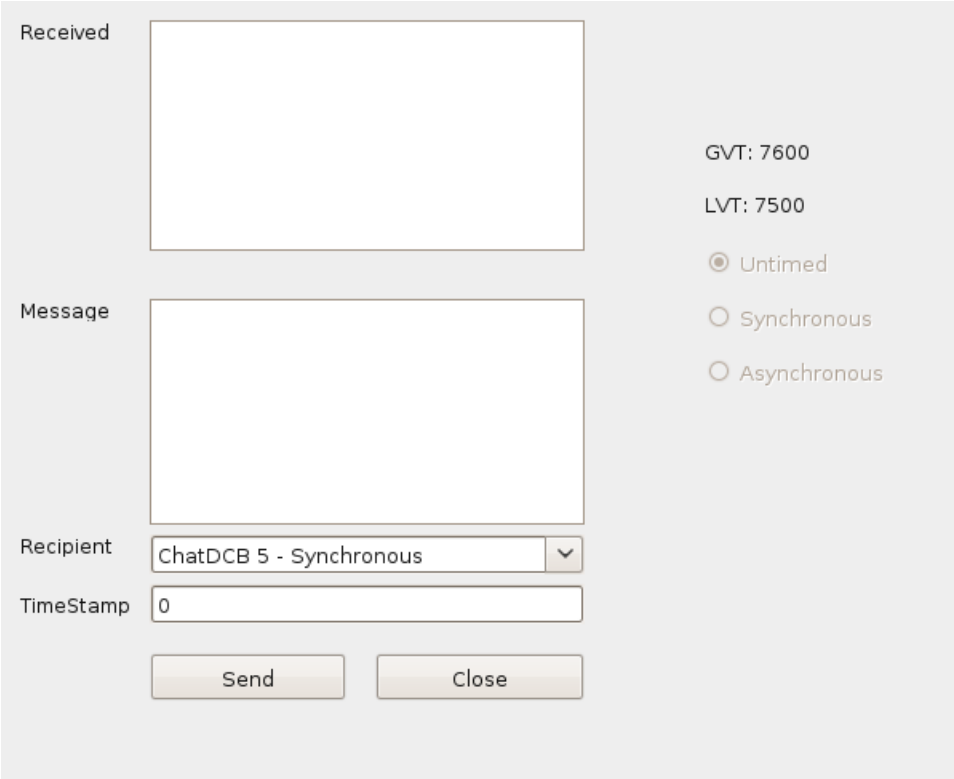

Figura 4.9: Exemplo de elemento untimed.

Fonte: Desenvolvido pelo autor.

Os elementos contidos no estudo de caso são: ChatDCB 5 (síncrono), ChatDCB 6 (assíncrono), ChatDCB 7 (assíncrono), ChatDCB 8 (assíncrono) e por último ChatDCB 9 (*untimed*).

Na Figura 4.8 temos o exemplo de um dos três elementos assíncronos. Visualmente é igual a um elemento síncrono, porém sua estrutura interna é diferente. Este *screenshot* foi tirado após o elemento realizar um *rollback* para receber a mensagem “teste”. Por isso, podemos observar a distância do GVT, maior do que 100 unidades de tempo, portanto o problema do cálculo do GVT mostrado na Seção 4.1.3 foi resolvido.

Por último podemos observar na Figura 4.9 o elemento ChatDCB 9, o elemento *untimed*.

4.2.1. Mecanismos de atualização do LVT

No estudo de caso, foi implementado um mecanismo que trabalha de forma quase totalmente independente do elemento. Não é totalmente, pois se o elemento se encontra em *rollback*, este mecanismo fica parado durante este tempo.

```
1.      while (rollback == false){
2.          aux = chatLVT + 100;
3.          chatLVT = GatewayX.updateLVT(aux);
4.      }
```

Figura 4.10: Atualização do LVT.

Fonte: Desenvolvido pelo autor.

Na Figura 4.10 podemos observar um pequeno trecho que mostra o mecanismo de atualização interna dos elementos do ChatDCB. Na linha 3 após *Gateway* o X é substituido pelo número de cada elemento (que varia de 5 a 9). O método “updateLVT” retorna o LVT que o GVT devolve para o elemento, não necessariamente será o valor que o elemento deseja (o valor atual + 100). Esta tentativa de evolução do LVT ocorre aproximadamente de 1 em 1 segundo.

4.2.2. Checkpoints no estudo de caso

No estudo de caso foi feito da seguinte forma: é verificado de tempos em tempos o acúmulo de eventos desde o estabelecimento do último *checkpoint* do elemento. Um evento é contado quando o elemento envia ou recebe mensagens. E para fins de estudo, foi estabelecido que uma quantidade razoável de eventos é igual a 10, portanto acima dessa quantidade de eventos é estabelecido um *checkpoint* após o último evento. Porém com a identificação de estados seguros, fica mais fácil definir uma política de periodicidade de estabelecimento de *checkpoints*.

Um último caso, porém não menos importante, em que ocorre o estabelecimento de um *checkpoint* é quando foi passado um tempo significativo desde o último evento realizado pelo elemento. Este tempo foi estabelecido como 5000 unidades de tempo da simulação. Então é estabelecido um *checkpoint* na marca de tempo igual ao último evento adicionado de 10 unidades de tempo.

4.2.3. Exemplo de execução

Um exemplo de execução do estudo de caso para testar *checkpoints*, *rollback*, avanço do GVT, entre outros, ocorreu da seguinte forma:

- O elemento 6 (assíncrono, com LVT igual 1000) disparou uma mensagem m_0 para o elemento 7 com *timestamp* igual a 2000;
- O elemento 7 (assíncrono) ao alcançar o LVT igual 2000 recebe a mensagem m_0 do elemento 6;
- O elemento 8 (assíncrono, com LVT = 400) envia uma mensagem m_1 para o elemento 6 com *timestamp* igual a 500;
- O DCB detecta que isto gerará um *rollback* e dispara uma anti-mensagem para o elemento 6 com *timestamp* igual a 500;
- O Gateway6 (*Gateway* do elemento 6) inicia o processo de

rollback. Primeiramente é parado o avanço no tempo do elemento. Posteriormente, através do DCBS, é iniciado o disparo de anti-mensagens para outros elementos. No caso citado, apenas será disparada uma anti-mensagem a_0 para o elemento 7 com *timestamp* igual a 1999, para que ele descarte a mensagem enviada pelo elemento 6. Isto ocorre pois o elemento 6 neste momento possui apenas um *checkpoint* com LVT igual a 0 e a mensagem enviada ao elemento 7 saiu do elemento 6 quando seu LVT era igual a 1000. O elemento volta para o início da simulação, evolui então para o tempo 500 e recebe a mensagem enviada por 8.

No exemplo citado não ouve recuo do GVT nem dos elementos 5 (síncrono), 8(assíncrono) e 9(*untimed*). A Figura 4.11 mostra com um gráfico o que aconteceu no exemplo citado anteriormente.

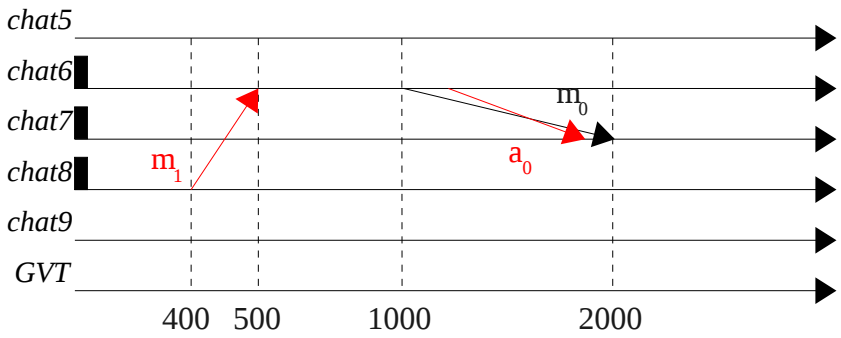

Figura 4.11: Exemplo de execução.

Fonte: Desenvolvido pelo autor.

5. CONCLUSÕES

Este trabalho apresentou a implementação de *checkpoints* não-coordenados no DCB. Os checkpoints permitem que durante a execução de elementos otimistas, estes podem realizar o *rollback* e restaurar as características do elemento salvas no *checkpoint*.

Os elementos otimistas necessitam de *checkpoints* para poderem voltar no tempo durante um *rollback*, como explicado anteriormente. A solução dos *checkpoints* não-coordenados foi escolhida, dentre outras, pois se adequou às principais características do DCB e não acarretou em perda de desempenho. Isto ocorre, pois a responsabilidade do estabelecimento e manutenção dos *checkpoints* fica por conta dos elementos, apesar do processo em si ser comandado pelo DCB.

Os elementos otimistas têm autonomia em relação aos módulos do DCB e em relação a outros elementos otimistas para o estabelecimento de seus *checkpoints*, portanto não existe troca de mensagem para seu estabelecimento, é uma decisão interna. Com isso não existe *overhead* de mensagens, consequentemente não há perda de desempenho.

Na realização do estudo de caso foi possível observar situações que exigiram a revisão de alguns aspectos do DCB. Entre eles, o cálculo do GVT durante um *rollback* e o aparecimento de *checkpoints* inúteis.

O problema dos *checkpoints* inúteis só pode ser eliminado com a identificação de estados seguros. Se faz imprescindível a eliminação destes *checkpoints*, pois ao trocar mensagens com elementos síncronos pode ser gerada uma inconsistência no sistema, como apresentado na Seção 4.1.3.

Com a implementação deste trabalho, o DCB se torna mais completo em se tratando de simulação otimista. O DCB é capaz de realizar as seguintes

operações:

- Disparar anti-mensagens: o módulo DCBS fica responsável pela análise de quais elementos devem receber anti-mensagens, isto ocorre de acordo com o *checkpoint* que será utilizado pelo elemento;
- Estabelecer *checkpoints:* como foi escolhida a solução de *checkpoints* não-coordenados, os elementos ficam responsáveis por estabelecer e gerenciar seus *checkpoints*.
- Realizar *rollback:* o *Gateway* é o principal responsável pela realização do *rollback*, que é uma ação conjunta dos módulos DCBS e DCBR juntamente com o elemento, coordenado pelo *Gateway;*

Após testes com o estudo de caso, foi observada a eficiência da implementação, pois não foi observado perda de desempenho no estabelecimento de *checkpoints* nem na execução do *rollback*. Estes fatos tornam-se imperceptíveis durante a execução.

Para a conclusão da simulação otimista, fica apenas faltando a identificação e análise de estados consistentes no DCB. Pois com isso, os *checkpoints* inúteis poderão ser removido sem prejuízo para a simulação.

5.1. Trabalhos Futuros

Como trabalho futuro podem ser feitos estudos sobre os elementos *untimed*. Definir melhor seus protocolos, criar uma identidade própria e ver o reflexo deles juntamente com elementos otimistas e conservadores no DCB.

Outro trabalho futuro poderia ser a implementação de *log* no DCB. Deixar a escolha do elemento, entre trabalhar com *logs* ou *checkpoints*, ser feita dinamicamente ou no momento de carregar os atributos do elemento. Continuando esse trabalho, pode ser feita uma comparação de desempenho entre execuções com apenas *logs*, apenas *checkpoints* ou ambos (alguns elementos utilizando *checkpoints* e outros usando *logs*).

6. REFERENCIAL BIBLIOGRÁFICO

AMORY, A.; MORAES, F.; OLIVEIRA, L.; HESSEL, F.; CALAZANS, N. **Desenvolvimento de um ambiente de Co-Simulação Distribuído e Heterogêneo.** Faculdade de Informática - Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, 2000.

BRUSCHI, S. M.; SANTANA, R. H. C. **ASDA – Um Ambiente de Simulação Distribuída Automático.** ICMC- Instituto de Ciências Matemáticas e de Computação, Universidade Federal de São Paulo, São Carlos, 2002.

CALVIN, J. O. e WEATHERLY, R. **An Introduction to the High Level Architecture (HLA) Runtime Infrasctructure (RTI).** 1996.

CHUNG C.A.. **Simulation Modeling Handbook: A Practical Approach.** CRC Press, 2004.

ELNOZAHY, M. et al. **A Survey of Rollback-Recovery Protocols in Message-Passing Systems.** [S.l.]:Carnegie Mellon University, Department of Computer Science, 1999. 45 f. (Technical Report CMUCS99148).

FERSCHA, A. **Parallel and Distributed Simulation of Discrete Event Systems.** [S.l.]:University of Vienna, Áustria, 1995. 65 f. (Technical Report on Parallel and Distributed Computing).

FUJIMOTO, R. M. **Time Management in the High Level Achitecture.** College of Computing Georgia Institute of Technology, 2000.

FUJIMOTO, R. M. **Parallel and Distributed Simulation Systems.** In: WINTER SIMULATION CONFERENCE, 33., 2001, Arlington, Virginia. Proceedings... [S.l.:s.n.], 2001. p.147-157.

HIGAKI, H., SHIMA K., TACHIKAWA T. e TAKIZAWA M. **Checkpoint and Rollback in Asynchronous Distributed Systems.** Department of Computers and Systems Engineering, Tokio Denki University, 1997.

KUHL, F.; WEATHERLY, R.; DAHMANN, J. Creating Computer **Simulation Systems: An Introcuction to the High Level Architecture.** [S.l.]: Prentice Hall PTR: The MITRE Corporation, 2000. 212 p.

MELLO, B. A. e WAGNER, F. R. **A Standard Co-Simulation Backbone**. I Forum de Estudantes de Microeletronica, Pirenopolis, Setembro, 2001.

MELLO, B. A. **Co-Simulação Distribuída de Sistemas Heterogêneos.** Tese de doutorado – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2005.

MELLO, B. A. **Distributed management of elements for modeling and simulation of heterogeneous models .** 2009.

MERTENS, D. R. 1976. **Principles of modeling and simulation in teaching and research.** Animal and Dairy Science Department - University of Georgia , 1976.

NASCIMENTO, J. M. A. **Simulador Computacional para Poços de Petróleo com Método de Elevação Artificial por Bombeio Mecânico .** Dissertação de mestrado – Centro de Tecnologia, Universidade Federal do Rio Grande do Norte, Natal, 2005.

REYNOLDS, P. F. **Heterogeneous Distributed Simulation.** Department of Computer Science and Institute for Parallel Computation – The University of Virginia , Charlottesville, Virginia, 1988.

SPERB, J. K. **Geração de Modelos de Co-Simulação Distribuída para a Arquitetura DCB.** Dissertação de Mestrado – Instituto de Informática – Universidade Federal do Rio Grande do Sul, Porto Alegre, 2003.

SOUZA, U. R. F.; SPERB, J. K.; MELLO, B. A. e WAGNER, F. R. **Tangram – Virtual Integration of Heterogeneous IP Components in a Distributed Co-simulation Environment .** IEEE – Design and Test of Computers, v.22, n.5, p.462-471, 2005.