UNIVERSIDADE ESTADUAL PAULISTA
"JULIO DE MESQUITA FILHO"
Pós-Graduação em Ciência da Computação

Bruno Simioni

# Plataforma de Simulação Computacional Paralela com base nos Conceitos de Relógios Lógicos e Tempo Virtual

UNESP
2012

Bruno Simioni

# Plataforma de Simulação Computacional Paralela com base nos Conceitos de Relógios Lógicos e Tempo Virtual

Orientadora: Profa. Dra. Renata Spolon Lobato

Dissertação apresentada para obtenção do título de Mestre em Ciência da Computação, área de Sistemas de Computação junto ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Campus de São José do Rio Preto.

UNESP
2012

Simioni, Bruno.
Plataforma de simulação computacional paralela com base nos conceitos de relógios lógicos e tempo virtual / Bruno Simioni. - São José do Rio Preto : [s.n.], 2012.
133 f. : il. ; 30 cm.

Orientador: Renata Spolon Lobato
Dissertação (mestrado) - Universidade Estadual Paulista, Instituto de Biociências, Letras e Ciências Exatas

1. Computação. 2. Simulação (Computador). 3. Relógio lógico. I. Lobato, Renata Spolon. II. Universidade Estadual Paulista, Instituto de Biociências, Letras e Ciências Exatas. III. Título.

CDU – 004.383.4

Ficha catalográfica elaborada pela Biblioteca do IBILCE
Campus de São José do Rio Preto - UNESP

Bruno Simioni

# Plataforma de Simulação Computacional Paralela com base nos Conceitos de Relógios Lógicos e Tempo Virtual

Dissertação apresentada para obtenção do título de Mestre em Ciência da Computação, área de Sistemas de Computação junto ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Campus de São José do Rio Preto.

## BANCA EXAMINADORA

Profa. Dra. Renata Spolon Lobato
Universidade Estadual Paulista "Júlio de Mesquita Filho"
Orientadora

Prof. Dr. Marcos Antonio Cavenaghi
Universidade Estadual Paulista "Júlio de Mesquita Filho"

Prof. Dr. Ronaldo Augusto Lara Gonçalves
Universidade Estadual de Maringá

São José do Rio Preto, 2 de março de 2012.

*Aos meus pais Celso e Antonia,*
*À minha irmã Beatriz,*
*À Camila,*

*Pela compreensão, amor e perseverança, que tornaram possível mais esse passo.*

# Agradecimentos

*À Profa. Dra. Renata Spolon Lobato*

Pelo grande e valioso esforço, mesmo com a distância, para tornar o estudo viável. Pela confiança, orientação e excelência, sem os quais esse trabalho não teria sido realizado.

*Aos meus pais Antonia e Celso,*

Por serem meu grande exemplo de esforço, dignidade, persistência, perseverança, e acima de tudo, auto-motivação. Pelo esforço sem medidas investido em minha formação e por compreenderem minha ausência durante os períodos críticos dos estudos.

*À minha irmã Beatriz,*

Pelo incentivo, compreensão e suporte nos momentos em que mais precisei.

*À Camila,*

Por ser a mulher da minha vida, e por compreender todas as fases desse estudo, abrindo mão de diversos momentos. Obrigado por me ajudar a ser uma pessoa melhor.

*À toda família Oliveira,*

Que me acolheu incondicionalmente como um filho/irmão e me incentivou tanto para a conclusão desse trabalho.

*Ao César,*

Meu sócio-irmão em quem pude confiar minha ausência, viabilizando esse trabalho.

*Aos meus amigos e colegas de trabalho,*

Que puderam compreender a minha ausência para mais esse passo em minha vida.

E também àqueles que direta ou indiretamente contribuíram para a realização deste trabalho.

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Resumo

Este trabalho apresenta a plataforma de simulação computacional de eventos *Darfia*, arquitetada através do emprego de memória distribuída e compartilhada (DSM) utilizando o *framework Terracotta DSO*, com o objetivo de facilitar a construção, manutenção e análise dessa abordagem de espaço de endereçamento local e distribuído. A plataforma de simulação foi desenvolvida utilizando-se de conceitos de tempo virtual e relógios lógicos propostos por Lamport, e foi implementada na linguagem de programação comercial, de quarta geração, Java, sendo configurável através de documentos portáveis. Este trabalho também apresenta uma introdução de estudos para simulações baseadas na *web*, oferecendo uma interface *web* para a plataforma de simulação, construída com tecnologias oferecidas pelo HTML5, proporcionando a utilização da plataforma de simulação também pela *web*.

# Abstract

This document presents the work related to a simulation platform event driven, *Darfia*, engineered through the use of distributed and shared memory (DSM) using the framework Terracotta DSO, in order to facilitate the construction, maintenance and analysis of this kind of approach to the local and distributed address space. The simulation platform was developed using the concepts of virtual time and logical clocks proposed by Lamport, and was implemented in the programming business, fourth generation, Java, and is configurable via portable documents. This work also provides an introduction to simulation studies of web-based, offering a web interface for the simulation platform, built with technologies offered by HTML5, providing the use of simulation platform also for the web.

# Capítulo 1

# Introdução

## 1.1 Introdução

Neste capítulo é apresentado o contexto no qual este trabalho foi inserido, destacando as principais motivações da computação distribuída e os recentes avanços na simulação paralela, bem como a problemática a ser trabalhada, e introduzindo estudos de uma plataforma de simulação paralela, através de sincronização e tempo virtual.

A simulação computacional de eventos é uma área da computação que congrega um grande volume de pesquisas, pela importância em facilitar o estudo de novas soluções nas mais diferentes áreas da ciência e tecnologia, sem a necessidade da construção de grandes e custosos protótipos. É utilizada amplamente como elemento crucial para tomada de decisões, através dos detalhes de implementações, e em ambientes diversos, incluindo situações que envolvem elevados riscos.

Nesse sentido, novas aplicações computacionais para representação de modelos tendem a simular sistemas cada vez mais complexos, e ajudar no trabalho de monitoramento para um melhor

aproveitamento de recursos. O ponto crucial da modelagem é o processo de constante melhora que, nesse caso, necessita das melhores tecnologias e das melhores abordagens. Além disso, a constante da melhora não cobre somente arquiteturas computacionais, como *clusters*, *grids*, ou os cenários mais recentes de *cloud computin*g, mas também um conjunto de aplicações utilizadas para o funcionamento e construção do modelo em si.

Para possibilitar uma menor distância entre o aumento da complexidade computacional necessária e os recursos existentes, tem-se algumas alternativas no sentido de conseguir um bom balanceamento entre a modelagem da aplicação e os recursos disponíveis [CHEN 2011], como por exemplo, análise e detecção de gargalos, pouca eficiência em gerenciamento de recursos, entre outros. Somente a área de modelagem de simulações, é um complexo campo da computação que envolve diversas ferramentas que facilitam o trabalho e observam detalhes como os mencionados. Um exemplo de ferramenta para esse uso é o SIMCAN, descrito em [FER 2011].

Entretanto, mesmo com a tendência atual do aumento do poder computacional através de dispositivos *multi-core* [CHEN 2011], é fato que o poder computacional pode não acompanhar o aumento da complexidade de modelos físico-matemáticos [HWA 87], fazendo com que os tempos de execução de tais aplicações possam crescer desproporcionalmente com relação a complexidade dos respectivos modelos.

Como alternativas a esse cenário, tem-se a possibilidade da melhora ou reorganização de algoritmos de soluções; porém, há casos que algum tipo de paralelismo na computação é necessário, utilizando-se de vários elementos computacionais.

Nesses casos, a carga de computação é divida entre diversos elementos processadores, através da reformulação do algoritmo da simulação. Tal carga computacional pode ser distribuída através de diversos processadores locais ao contexto em questão, ou processadores distribuídos, conectados através de alguma rede de intercomunicação [HWA 87].

Deve-se anotar também que a computação distribuída através de diversos processadores remotos ilustra outros benefícios, tais como distribuição geográfica, tolerância a falhas, e o baixo acoplamento, permitindo para o software a fácil manutenção [FUJ 2000].

No contexto de distribuição de carga entre processadores, existe a tratativa do tipo de organização de memória a ser utilizada na operação. Nesse sentido, existem dois grandes grupos: memória compartilhada e memória distribuída.

Há também uma junção entre as duas definições anteriores, caracterizada por memória compartilhada distribuída, onde tem-se os benefícios da memória compartilhada, aliado a uma camada transparente, responsável por administrar a construção de tal memória de forma distribuída através da utilização de diversos montantes de mémória, em outros computadores, conectados por uma rede de interconexão [PRO 96].

No contexto de memória distribuída e processadores desacoplados, modelos mais flexíveis tornaram-se uma opção eficiente para tais sistemas, sendo que com a distribuição do poder computacional entre componentes remotos foi possível, de tal forma, que programas em máquinas locais tivessem acesso a serviços remotos ou distribuídos. Esse benefício trouxe novos desafios, tanto no aspecto distribuído de desenvolver e gerenciar softwares, quanto nas dificuldades inerentes ao novo ambiente: um conjunto de sistemas autônomos sem acesso a estados globais.

Dificuldades como a impossibilidade de resolver certos problemas básicos, como concordar em ações comuns na presença de falhas, ou mesmo a falta de ferramentas (de desenvolvimento e gerenciamento distribuído) passaram a ser uma barreira para o novo paradigma da computação distribuída [MAC 2006].

Devido a problemas dessa ordem, diversos protocolos e ferramentas de programação e simulação paralela foram desenvolvidos, e podem ser divididos em dois grandes grupos: conservativos e otimistas, dependendo da necessidade de preservar a ordem correta de execução de eventos inerentes ao processo. Há também protocolos híbridos, os quais não se encaixam em somente uma definição de tal escopo.

Protocolos conservativos mantêm a ordem exata de acontecimento de eventos no ambiente distribuído, enquanto protocolos otimistas não, criando mecanismos de retrocesso, quando a ordem de dependências em um dado momento pode não ser correta.

Esse é o contexto onde o presente trabalho está inserido, propondo uma plataforma de tratamento de processos e eventos baseado em abordagens de implementação de protocolos otimistas, em um ambiente de memória compartilhada distribuída, para disponibilizar uma forma acessível de cálculo de relógios globais, assim minimizando os problemas da computação distribuída.

## 1.2 Motivações e Inovações

Com o contexto da proposta, as principais inovações e motivações do presente trabalho se dão pelo o emprego de mecanismos computacionais modernos e simples para abordagens de simulação paralela já bem estabelecidas e eficazes.

Para tal, na questão da implementação do protocolo de simulação paralela, foi escolhida a linguagem de programação Java, de quarta geração, e que provê entre outras características [GOSLING 2000]:

- Orientação a objetos;
- Portabilidade e independência de plataforma;
- Recursos avançados de rede (TCP/IP, *Sockets,* etc);
- Mecanismos básicos de segurança e acesso à memória;
- Vasto conjunto de APIs distribuídas conjuntamente;
- Desalocação e coleta de lixo automática;
- Carga dinâmica de código.

Além da utilização de uma linguagem de programação comercial para implementação da plataforma de sincronização e simulação paralela, também propõe-se a utilização do *framework* Terracotta, que será melhor discutido ao longo do trabalho, para facilitar o trabalho de criação, distribuição e gerência de um ambiente de memória compartilhada e distribuída (*Distributed Shared Memory* - DSM), onde o protocolo será utilizado.

Para facilitar o acesso e utilização da plataforma, empregou-se tecnologias *web* para construção de uma interface de dados acessível através de navegadores modernos, para obtenção de dados em tempo real, indepentende de sistema operacional.

Com tais inovações, propõe-se a adaptação e implementação de protocolos otimistas para ambientes de memória compartilhada e distribuída, para facilitar que tais protocolos escalem em ambientes atuais de simulação paralela.

## 1.3 Problemática

A problemática discutida nesse trabalho é sobre a remodelagem de protocolos (TimeWarp) já existentes de sincronização e simulação paralela em ambiente distribuído, utilizando novas tecnologias (linguagem de programação comercial, programação em alto nível, etc) para implementação do ambiente de simulação.

O ambiente deve utilizar-se de uma arquitetura de memória compartilhada e distribuída com a finalidade de analisar troca de mensagens, procedimentos de recuperação de estados, erros de causalidade e análise de testes e desempenho, além do emprego e execução de operações de cálculos de tempo virtual, utilizando-se de mecanismos lógicos fundamentados em relógios lógicos, apresentados por Lamport [LAM 78].

O ambiente de memória compartilhada e distribuída (DSM) será provido por um *framework* baseado na API (*Application Programming Interface*) TerraCotta [TER 2008a], fundamentado na tecnologia *JavaSpaces* para a criação de espaços de memória reservados e passível de serem conectados via interfaces de comunicação, compartilhando assim, dados e instâncias de objetos.

## 1.4 Organização do texto

A organização do presente documento consiste em uma vasta fundamentação teórica que dará base para a proposta do autor, e a apresentação da proposta do trabalho, além da apresentação de resultados obtidos através de modelos de estudo.

No capítulo 2 é feito o embasamento teórico, analisando tópicos sobre computação e simulação paralela, além de uma breve descrição sobre modelos de filas e ordenação de eventos em sistemas distribuídos. Serão apresentados algumas das abordagens existentes sobre protocolos de simulação paralela, bem como considerações sobre cada tipo, além da descrição do funcionamento da API do Terracotta.

No capítulo 3 é apresentada a descrição da plataforma de simulação desenvolvida, junto com as testes e validações.

No capítulo 4 são apresentados estudos relacionados com a análise de redes de fila e um modelo sintético.

E por fim, no capítulo 5 são apresentadas as conclusões, as contribuições e as propostas para continuidade do trabalho.

# Capítulo 2

# Revisão Bibliográfica

## 2.1 Considerações Iniciais

Neste capítulo é apresentada uma vasta revisão bibliográfica em torno dos temas de programação paralela, simulação paralela, introdução a teoria de filas, ordenação de eventos em sistemas distribuídos e o funcionamento do *framework* Terracotta - responsável por prover uma ambiente DSM - com o objetivo de dar sustentação aos temas necessários à abordagem desse trabalho.

A organização do capítulo é dada da seguinte maneira: na seção 2.2 tem-se um apanhado geral sobre programação paralela e suas abordagens de arquitetura: troca de mensagens, memória compartilhar e DSM, além de modelos de concorrência. Na seção 2.3, fala-se sobre simulação paralela e seus protocolos, conservativos e otimistas, além de algumas variações desses. Na seção 2.4 é abordado o tema de modelagem de redes de filas, elencando suas propriedades e modelos. Na seção 2.5, o tema de ordenação de eventos em sistemas distribuídos será abordado, mostrando ao leitor a necessidade de se utilizar um tempo não físico, virtual, para comunicação entre componentes de uma simulação paralela, e finalmente na seção 2.6, o tema abordado será sobre o

*framework* de memória compartilhada distribuída Terracotta, responsável por abstrair a aglomeração de memórias virtuais em máquinas distribuídas.

## 2.2 Programação Paralela

A presente seção tem como objetivo detalhar as abordagens de organização de memória e arquitetura de software e hardware, utilizadas em ambientes de processamento paralelo. Serão discutidas diversas formas de arquitetura e programação, juntamente com os modelos de consistência de dados para tais abordagens, finalizando com as considerações para cada modelo.

A crescente demanda por alto poder computacional sempre motivou a evolução das pesquisas na área da computação, viabilizando implementações de aplicações que envolvem uma elevada taxa de computação e grandes volumes de dados. Como tentativa para diminuir o tempo de resposta, têm-se as alternativas de aumentar o desempenho do processador ou de utilizar diversos processadores [BRU 03], dividindo a tarefa a ser executada.

Para esse fim, diversas arquiteturas surgiram ao longo do tempo, demandando classificações e modelos de taxonomia. A mais famosa entre essas é a classificação de Michael Flynn, proposta em 1972, que agrupa as arquiteturas que surgiram com o tempo, em quatro grandes áreas: SISD (*Single Instruction Flow, Single Data Flow*), MISD (*Multiple Instruction Flow, Single Data Flow*), SIMD (*Single Instruction Flow, Multiple Data Flow*), e MIMD (*Multiple Instruction Flow, Multiple Data Flow*) [FLY 72]

Com todas essas possibilidades de organizações arquiteturais, a programação paralela tornou-se facilmente uma tarefa pouco trivial. A escolha da abordagem a ser utilizada é fundamental, pois o paradigma escolhido determina diretamente como as tarefas serão realizadas. Com isso, a atenção do desenvolvedor voltou-se para explorar o paralelismo da aplicação e estabelecer a integração entre os processos

Dentre os diversos paradigmas de programação concebidos e disponíveis, pode-se dividir os mais comuns como baseados em trocas de mensagens entre nós de uma rede, e baseados em memória compartilhada entre os diversos elementos processadores num mesmo ambiente computacional local.

No paradigma de programação por troca de mensagens, a maneira com que os processos interagem é feita de forma explícita através de primitivas para envio e recebimento de mensagens.

Já o paradigma de memória compartilhada assume a existência de um espaço único para endereçamento global, fazendo com que a interação entre processos seja feita através de leituras e escritas em dados compartilhados [SEI 98].

Além disso, o paralelismo de uma aplicação pode ser definido de forma implícita ou explícita. Para a forma implícita, foram criados compiladores que paralelizam automaticamente os *softwares*, minimizando as dificuldades da programação paralela, porém, ignorando algumas regras de negócio da simulação, que possibilitariam uma maior taxa de paralelistmo, reaproveitando programas já sequencialmente construídos. Entretanto, a programação paralela explícita proporciona soluções de paralelismo que vão além da detecção automática, permitindo uma maior exploração de questões inerentes à solução [POL 88], através de uma análise mais profunda sob a ótica da simulação em si, e de como paralelizá-la.

### 2.2.1 Programação por troca de mensagens

Os sistemas de computação paralela que não possuem memória compartilhada local encaixam-se no escopo da programação por troca de mensagens, necessitando implementar a troca de mensagens e rotinas de sua sincronização para a comunicação entre os processadores [TIB 96].

Nesse modelo, os processadores estão ligados a uma interface de conexão que permite o seu envio/recebimento, onde cada processador possui sua própria memória local, que só pode ser acessada diretamente por ele [KUM 94]. Além disso, esta abordagem também pode ser implementada e utilizada em multiprocessadores de memória compartilhada.

Uma troca de mensagem envolve, pelo menos, dois processos: o transmissor, que envia a mensagem, e o receptor, que a recebe. Isso é feito através de primitivas do tipo *Send* e *Receive* em duas principais modalidades: a comunicação síncrona e a comunicação assíncrona.

Na comunicação com troca de mensagens síncrona, o transmissor envia a mensagem para o receptor e aguarda até que este sinalize o seu recebimento. Se o receptor não estiver pronto para receber a mensagem, o transmissor é bloqueado temporariamente; caso o receptor esteja pronto para receber a mensagem antes que ela seja enviada, também será bloqueado temporariamente. Após receber a mensagem, o receptor envia de volta um sinal confirmando seu recebimento com o que, imediatamente, os dois são desbloqueados para seguir seu fluxo de execução. O sinal de recebimento (a confirmação) é transparente ao usuário, sendo enviado sem que o programador

necessite explicitamente fazê-lo, ou seja, ele acontece a cargo do sistema.

Na comunicação assíncrona, o transmissor envia a mensagem e prossegue em seu fluxo normal de execução sem sofrer nenhum tipo de bloqueio. Caso o receptor ainda não esteja pronto para receber a mensagem, esta deve permanecer armazenada temporariamente em um *buffer*; se estiver pronto para receber uma mensagem que ainda não foi enviada, deverá permanecer bloqueado temporariamente. Os sistemas que implementam esta modalidade geralmente oferecem primitivas adicionais que verificam se alguma mensagem chegou ou não, sem bloquear o processo receptor ([GEY 98]).

Com o objetivo de aumentar o desempenho de um sistema orientado a troca de mensagens, mais processadores podem ser inseridos, porém com alguns limite à vista. Inicialmente, havia a conclusão do crescimento linear de desempenho, em relação ao aumento do número de elementos processadores, porém a conjectura de Minsky prova que a curva é logarítmica, em relação ao número de nós na rede, devido ao *overhead* gerado pelo acúmulo de mensagens na rede de intercomunicação entre os processadores. Além disso, há o limitante da carga de trabalho a ser distribuída por processadores (paralelismo infinito), onde quanto maior o número de processadores, menor a carga de trabalho a ser distribuída [HWA 87].

### 2.2.2 Programação com memória compartilhada

Na segunda classificação de memória estão os sistemas de computador com memória compartilhada, que é determinado por um grupo de processadores que se comunicam realizando operações sobre variáveis compartilhadas ao invés de enviar e receber mensagens [KUM 94]. Tal modelo é comumente utilizado para programar em máquinas de memória centralizada, o que, necessariamente, não significa que possa ser aplicado somente nesse ambiente. Os processadores comunicam-se da mesma maneira que múltiplas *threads* ou processos o fazem na programação seqüencial, podendo obter ou liberar *locks* e usar semáforos, monitores ou algum outro mecanismo de comunicação e sincronização entre processos.

Caracteriza-se como vantagens inerente a esse modelo o fato do programador não precisar mover explicitamente os dados, facilitando o processo de depuração devido o dado estar imediatamente acessível, e a portabilidade, pois aplicações que utilizam o modelo de memória compartilhada são mais fáceis de serem transportadas para ambientes paralelos. O paradigma de

programação de memória compartilhada é considerado mais simples do que a troca de mensagens, uma vez que o programador deixa de controlar a comunicação entre os processos através de mensagens explícitas [CAL 99]. Alguns autores, porém, a consideram menos segura ou viável que a programação via trocas de mensagens, por questões de prioridade e exclusividade de acesso à zonas críticas de dados compartilhados, e portanto, problemas de coerência de informação, e questões de gargalo em relação ao acesso exclusivo, quando um grande número de processadores disputam pelo recurso.

### 2.2.3 Programação com memória compartilhada e distribuída

O uso do paradigma de programação com memória compartilhada esteve associado a multiprocessadores com memória centralizada durante muito tempo. Porém, tais arquiteturas apresentam contenção no acesso ao barramento comum que dá acesso à memória, ainda que existam variações que não fazem uso de barramento simples, mas sim um conjunto de *switchs*, limitando sua escalabilidade. Sistemas com memória compartilhada distribuída, ou DSM, surgiram com a motivação da facilidade de programação do paradigma de memória compartilhada com a escalabilidade de ambientes distribuídos orientado a troca de mensagens [SEI 98, IOS 2004].

A arquitetura DSM pressupõe a existência de processadores com memórias locais, mas compartilhando um espaço de endereçamento único [SIL 99]. Em sistemas multicomputadores, cópias nas memórias locais dos dados compartilhados permitem que seus acessos sejam efetuados eficientemente. Entretanto, essa abordagem chamada *caching*, cria o problema de consistência da *cache*, que ocorre quando um processador atualiza dados compartilhados [HWA 87]. Como cópias desses dados podem estar presentes em outros nós, estes devem ser mantidos consistentes, não permitindo que um processador obtenha um valor não atualizado. Para resolver esse problema, alguns modelos de consistência foram propostos [KSH 2008].

Além do problema da coerência de *cache*, três abordagens têm sido utilizadas na implementação de sistemas DSM: implementação por *hardware*, estendendo técnicas tradicionais de *caching* para arquiteturas escaláveis, implementação de bibliotecas pelo sistema operacional, onde o compartilhamento e a coerência de dados são obtidos através de mecanismos do gerenciamento de memória virtual, e a implementação pelo compilador e bibliotecas, onde acessos compartilhados são automaticamente convertidos em primitivas de coerência e sincronização [SIL

99, KSH 2008].

### 2.2.4 Modelos de Coerência

Levando em conta que muitos programas paralelos definem seus próprios requisitos de consistência de mais alto nível, requisitos de consistência de memória podem ser relaxados. Porém, para a construção correta de um programa em um sistema com memória compartilhada distribuída, o programador deve conhecer como as atualizações são propagadas no sistema.

Os modelos de consistência de baixo nível, podem ser listados como segue [KSH 2008, REB 2000]:

- consistência estrita: qualquer leitura em uma determinada posição de memória retorna o valor mais recentemente escrito na mesma posição;
- consistência seqüencial: Os acessos a posições de memória devem ser ordenados sequencialmente, considerando os acessos de todos os processadores, ou *threads*;
- consistência causal: escritas que são relacionadas com outras devem ser vistas em todos os processos na mesma ordem; entretanto, escritas concorrentes podem ser vistas com diferentes ordens em processadores distintos;
- consistência de processador: escritas de um único processador são recebidas em ordem nos demais processadores; já escritas de diferentes processadores podem ser vistas em diferente ordem por outros processadores;
- consistência fraca: os acessos aos dados são tratados separadamente dos acessos de sincronização, mas requerem que todos os acessos aos dados anteriores sejam feitos antes que o acesso de uma sincronização seja obtido;
- consistência *release*: é uma consistência fraca com dois tipos de operadores: *acquire* e *release*. Um operador *acquire* é usado no início de uma seção crítica, para adquirir o direito exclusivo à sua execução, e o operador *release* para liberá-la e exportar os dados atualizados;
- consistência *lazy release*: é um tipo de consistência *release* que busca reduzir o número de mensagens e a quantidade de dados exportados por acessos remotos. Neste modelo, as modificações são exportadas apenas quando ocorre um acesso aos dados, através do operador *acquire*;

- consistência *entry*: é utilizada a relação entre variáveis de sincronização específicas que protegem as seções críticas e os acessos aos dados compartilhados nelas efetuados. São permitidos múltiplos acessos a dados compartilhados para leitura, através dos acessos de sincronização, que podem ser especificados como exclusivos ou não-exclusivos.

A coerência de memória mais intuitiva é a apresentada pela consistência estrita (*strict consistency*), na qual uma leitura retorna o valor mais recente do dado. Entretanto "o mais recente" é um conceito ambíguo em um sistema distribuído, fazendo com que alguns sistemas DSM providenciem uma forma reduzida de coerência de memória.

Modelos com consistência forte, porém menos restritivos que a estrita, sofrem de problemas de baixo desempenho. Para melhorá-lo, modelos de coerência fraca podem ser a solução, porém, impossibilitando a exatidão de resultados na solução como um todo. Trabalhos com o objetivo de reduzir o *overhead* das barreiras de sincronização e melhorar mecanismos de coerência podem ser encontrados em [SEI 98].

### 2.2.5 DSM e processadores de diversos núcleos

Com avanços recentes em áreas estratégicas como a divisão e a criação de núcleos de processadores comerciais de propósito geral, questiona-se como adaptar estratégias de memória compartilhada distribuída, ou mesmo substituir esforços em DSM, para a adoção de abordagens multi-núcleos.

Trabalhos recentes demonstram o emprego conjunto de multiprocessadores para implantar estratégias de memória DSM, e assim, obter um ganho maior de desempenho.

Em [CHEN 2010] e [CHEN 2011], comenta-se que atualmente há uma tendência das arquiteturas *single-chip* evoluírem de *single-core* para *multi-core* e até mesmo *many-core*. Com tal evolução, passa-se a exigir mecanismos de comunicação e organização dos núcleos de processamento que sejam eficientes e escaláveis em relação ao tamanho do sistema alvo, característica a qual os mecanismos baseados no uso de vários barramentos convencionais não realizam de maneira satisfatória.

Neste cenário introduz-se o conceito de *Network- on-Chip* ou *NoC*, que aplica teorias e sistemáticas de redes a *SoC*s (System-on-Chip). Outra tendência é o aumento do tamanho da

memória embutida no chip, fazendo com que esta ocupe uma área cada vez maior do *SoC*.

Memórias grandes e centralizadas geram gargalos de desempenho, apresentam problemas de consumo de energia e definem altos custos em sistemas de médio a grande porte, alavancando abordagens de memórias distribuídas como importantes, na solução de tais problemas.

Considerando tais fatores, o artigo tem como proposta a implementação de um suporte à Memória Distribuída Compartilhada (DSM) em *NoC*s por meio da utilização de um módulo de hardware programável, denominado "*Dual Microcoded Controller*" (DMC). A escolha por memória compartilhada e distribuída se deve ao fato desta facilitar a programação e permitir o reuso de código legado já testado e validado. Já o *DMC* é flexível ao permitir que o usuário o configure programando suas funções (e, assim, atende exigências como *time-to-market*, por exemplo) e apresenta bom desempenho.

Já em [RAS 2010], há a apresentação da arquiteura modular *Clupea*, baseada em múltiplos núcleos (*many core*). Os conceitos chaves da arquitetura são módulos de processamento alocáveis e suporte individual configurável para abstrações de programação para cada aplicação que execute no sistema.

O suporte à modelos de programação é baseado em interfaces do processador *NoC* especializadas e programáveis, que integra a interface *NoC* e partes do controlador de cache.

Comparado ao trabalho anteriormente citado ([CHEN 2010]), a arquitetura *Clupe* usa somente um processador de interface de rede simples e proporciona total suporte para coerência de *cache* em sistemas de memória compartilhada.

O modelo DSM, e também seu emprego em arquiteturas *multi-core*, apresenta algumas vantagens sobre o modelo de programação por troca de mensagens e seus adjacentes. Uma dessas vantagens pode ser pontuada como a forma lógica da programação. O protocolo de acesso usado é consistente com o modo como as aplicações sequenciais acessam os dados, permitindo uma transição natural para aplicações distribuídas.

A abordagem DSM esconde o mecanismo de comunicação dos processos e permite que estruturas complexas possam ser enviadas somente por referência, simplificando a programação. Por essa razão, o código de aplicações paralelas escritas para ambientes com memória compartilhada distribuída é menor e mais compreensível que os equivalentes com trocas de mensagens [REB 2000].

Uma grande preocupação dos modelos de programação baseados em memória compartilhada é a sincronização entre processadores no momento de acessos a variáveis compartilhadas, a qual pode ser efetuada usando, por exemplo, *locks* ou barreiras. A programação com troca de mensagens também apresenta a necessidade de sincronização, a qual, contudo, é atingida implicitamente através de primitivas de comunicação.[SEI 98].

Entretanto, os modelos DSM também sofrem de desvantagens em relação aos outros modelos. Um exemplo que caracteriza tais problemas é o desempenho em relação a ambientes puramente de memória compartilhada. Em ambientes distribuídos, com cada processador tendo acesso somente à sua memória local e a DSM sendo implementada através de *software* (ambientes de estações de trabalho ligadas em rede), a comunicação, de uma forma ou outra, ocorre através de trocas de mensagens.

Os *softwares* que proporcionam a memória compartilhada apenas escondem este mecanismo do programador. Dessa forma, a inserção de novas camadas deteriora o desempenho. Estudos realizados por Lu ([LU 95, LU 97]) mostram claramente as diferenças entre DSM e troca de mensagens.

Entretanto, a abordagem DSM proporciona uma transição menos custosa para a utilização de programação paralela em implementações, ao custo do desempenho das novas camadas de instrumentação de código. Tais adições geram novos desafios em relação a coerência de dados e quantidade de dados que devem ser transmitidos, o que é variável em relação ao propósito da implementação e da aplicação

## 2.3 Simulação Paralela e Distribuída

Nessa seção, serão apresentadas algumas das abordagens mais comuns utilizadas em simulação paralela. As técnicas otimistas e conservativas, para a elaboração da estratégia de protocolos de simulação serão explanadas, bem como as derivações e modificações das técnicas otimistas, foco do trabalho.

Simular significa reproduzir o funcionamento de um sistema do mundo real com o auxílio de um modelo, o que permite testar hipóteses sobre o valor de alguns componentes controlados [SIL 98]. A simulação baseada em computadores é uma das ferramentas mais poderosas disponíveis para projetar, planejar, controlar e avaliar novas alternativas e/ou mudanças de estratégia em sistemas do

mundo real [SHA 92].

Tal tipo de simulação é amplamente utilizado nos dias de hoje em sistemas de predição e análise de desempenho. Entretanto, com o aumento da demanda de grandes modelos e cada vez mais complexos, a simulação sequencial vem se tornando menos atrativa em virtude do tempo de simulação consumido. Na tentativa de reduzir o tempo de simulação e aumentar o tamanho dos sistemas simulados, tem-se utilizado vários processadores para acelerar o processo de simulação. Entretanto, a maior desvantagem desse procedimento é a complexidade inerente a esse tipo de simulação, uma vez que a idéia do "tempo global" não é facilmente manipulável na computação paralela [OVE 91]. Para esse fim, diversos protocolos de simulação paralela foram desenvolvidos nas últimas duas décadas [PHA 99a].

A representação da simulação deve ser realizada no decorrer de um tempo (chamando-se de tempo simulado ou tempo de simulação), categorizando-se em tempo discreto ou contínuo. Este pode ou não coincidir com o tempo físico (de relógio) que a aplicação computacional consome durante sua execução. É comum o uso de escalas de tempo em processos de simulação discreta para acelerar o processo [FUJ 2000].

Com a representação do tempo na realização da simulação, podem ser utilizadas três perspectivas para assumir a mudança de estados do sistema real: simulação por escalonamento de eventos, por análise de atividades e por interação entre processos [COP 96]. Em simulações baseadas em escalonamento de eventos, um evento acontece num ponto isolado do tempo, no qual decisões devem ser tomadas de forma a iniciar ou terminar uma atividade [OVE 91].

O funcionamento geral de simulações discretas utilizam-se três entidades: atividades, processo e evento, por unidade de processamento. A relação dessas entidades é ilustrada na Figura 2.1.

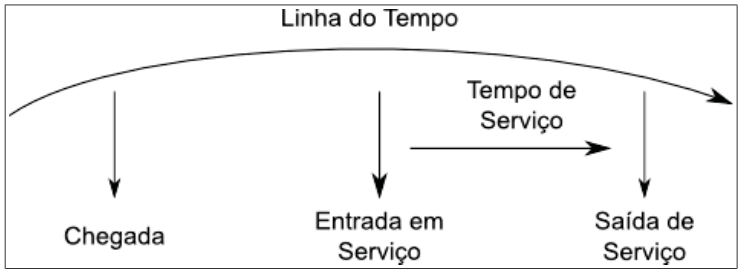


Figura 2.1 - Funcionamento geral de uma simulação

A atividade é definida como a unidade de trabalho, tendo a ela associada um tempo de serviço (ou execução), e é demarcada por eventos de entrada e saída do prestador de serviço. Sua definição depende do nível de abstração adotado na simulação.

A simulação de eventos também possui internamente uma lista ordenada (por tempo de execução) de eventos a serem simulados, conhecida como *Event List* (EVL ou LEF – Lista de Eventos Futuros). O processo de simulação ocorre pela seleção do evento com o menor tempo para execução (*timestamp*) e sua posterior execução, modificando assim, o tempo da simulação. Na execução de um evento, este pode agendar um novo a ser executado em um tempo futuro [FUJ 90a].

Na simulação paralela de eventos, o sistema a ser simulado é dividido em subsistemas a serem executados de forma paralela nos processadores, criando em cada um, um *Logical Process* (LP). Cada LP pode trabalhar com a evolução de seu tempo de processamento independentemente de outros LPs, mantendo, assim, seu próprio *Local Virtual Time* (LVT). O conjunto de LVTs de todos os LPs irá determinar o valor do *Global Virtual Time* (GVT), que representa o avanço geral da simulação. A execução da simulação num LP pode utilizar ou alterar dados pertencentes a outros LPs [FER 96], necessitando de sofisticados algoritmos de comunicações locais e intercâmbio de informações [OVE 91].

Pode-se visualizar a simulação paralela como a cooperação de um conjunto de LPs interagindo entre si, cada um deles simulando uma parte ou região do sistema do mundo real. Geralmente, uma região é representada pelo conjunto de todos os eventos a ela relacionados no mundo real [FER 96].

Um LP é dividido basicamente em duas funções distintas: a execução de eventos em um prestador de serviço, ou centro de serviço, ou máquina de simulação, e a comunicação com outros LPs. A execução de eventos é realizada pelo prestador de serviço, processando somente os eventos locais, podendo em sua execução, agendar um novo evento a ser inserido em sua própria lista ou em outra. O controle do avanço local da simulação também é controlado pelo prestador. A comunicação é realizada pela interface de comunicação, que utiliza um sistema para possibilitar a troca de informações, bem como a sincronização dos LPs. A interface de comunicação, cuida ainda, da propagação de efeitos causais em eventos a serem executados por LP remotos. Um exemplo de

LP é ilustrado na Figura 2.2.


Figura 2.2 – Ambiente de Simulação [MIS 86]

Um dos principais problemas dos modelos de simulação paralela é a forte correlação entre os seus componentes. Para que a execução em paralelo de um modelo de simulação seja correta, um pré-requisito a ser atendido é o da garantia da não-ocorrência de Erros de Causalidade Local (ou *Local Causality Constraint* – LCC) [FUJ 90a, FER 96].

Erros de causalidade local são provocados pelo tratamento de eventos em ordem temporal não crescente durante a simulação. Dessa forma, eventos futuros podem, erroneamente, afetar o comportamento do modelo quando da execução de fatos passados. Para evitar tais erros, é necessária a utilização de protocolos de sincronização, de modo a certificar-se de que o modelo está sendo executado corretamente com relação à ordenação temporal dos eventos [FUJ 2000]. Na figura 2.3 é ilustrado um exemplo do erro.

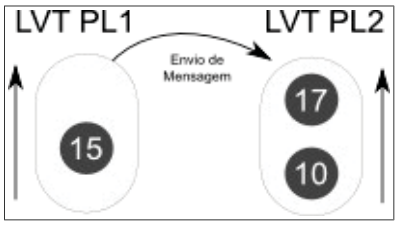


Figura 2.3 – Ocorrência de Erro de Causalidade

Foram propostos diversos protocolos de simulação paralela com o objetivo de tratar LCC. O desempenho de tais protocolos é intrinsecamente dependente do ambiente, da finalidade e do tipo de modelagem utilizado no processo de simulação, não tendo sentido sua classificação. Porém, duas linhas claras de abordagem podem ser determinadas: conservativos e otimistas.

Os protocolos conservativos, propostos inicialmente por Chandy e Misra, processam os eventos seguindo estritamente a sua ordem seqüencial de acontecimento. Por sua vez, a ênfase dos protocolos otimistas, pela primeira vez proposto por Jefferson, está na detecção e recuperação de erros de causalidade local. Mais detalhadamente, são descritos como:

• Protocolos Conservativos: inicialmente propostos por Chandy e Misra (1979) e Bryant (1984) [FER 96, FUJ 2000], são assim denominados por se aproximarem muito dos modelos de execução tradicional de simulação. Neste tipo de protocolo, um evento somente é executado quando todos os eventos que podem afetá-lo já o foram. Dessa forma, sua funcionalidade inibe a ocorrência de LCC;

• Protocolos Otimistas: buscam explorar a violação aos LCC sendo baseados na observação empírica de que a ocorrência de erros de causalidade tende a ser minimizada pelo próprio processo de simulação. Assim, é permitida a execução de eventos mesmo que não se tenha a garantia de que seja temporalmente segura, tratando os erros de causalidade através de processos de *rollback* [JEF 85].

### 2.3.1 Protocolos Conservativos

A classe de protocolos conservativos, também conhecidos como CMB (Chandy, Misra e Byron, seus criadores), não permite a ocorrência de LCC. O princípio de funcionamento desses protocolos consiste em determinar quando é seguro executar um evento. Por exemplo, se determinado LP possui um evento não processado $E_1$, com timestamp $T_1$, não existindo nenhum outro com tempo para execução inferior ao seu, e o LP puder determinar que é impossível o recebimento posterior de evento com tempo menor que o $T_1$ a ser executado, o evento $E_1$ é um evento seguro ([FUJ 90a, THU 2010]).

Tal classe necessita em cada LP a especificação de todos os canais de comunicação, definindo todos nós adjacentes que podem comunicar-se. Para determinar quando um evento pode ser executado, é necessário que todos os outros LPs enviem mensagens aos demais, transmitindo-as através do canal de comunicação em ordem crescente de tempo para execução. Esse procedimento garante que a última mensagem recebida em um canal tenha o maior *timestamp* entre as já recebidas [FUJ 2000].

Tais mensagens que chegam por um determinado canal são armazenadas em ordem FIFO (*First In, First Out*), que é a mesma ordem dos tempos das mensagens a serem executadas. Cada canal possui um relógio com valor igual ao da primeira mensagem da fila, caso a mesma contenha mensagens; ou igual ao da última mensagem recebida, se estiver vazia.

Em seu funcionamento, o LP repetidamente seleciona, dentre os eventos seguros a serem processados, o de menor *timestamp*, executando-o. Os LPs contendo eventos a processar não seguros devem ser bloqueados enquanto permanecerem nessa situação. A situação de bloqueio pode ser originada pela inexistência de eventos com o menor relógio a serem processados na lista, mas existirem eventos em outras listas de outros canais de comunicação; dessa forma, ocasionam-se situações de *deadlock* que devem ser tratadas [FUJ 90a, OVE 91, FER 96, LON 2010].

A situação de *deadlock* ocorre num ciclo de processos lógicos quando esses estão bloqueados. Fujimoto [FUJ 90a] aponta que *deadlocks* podem ser freqüentes em situações em que existe um baixo número de eventos em relação ao número de ligações entre processos lógicos.

Mensagens nulas (que denotam eventos sem efeito) podem ser utilizadas para evitar as situações de *deadlock*. As mensagens nulas não têm relação com o modelo a ser simulado e servem apenas para sincronização entre os LPs. Determinar o *timestamp* das mensagens nulas a serem enviadas é fator decisivo para o funcionamento dessa abordagem. Uma forma para fazer isso é usar o menor valor das filas de chegadas de mensagens, que é o tempo do próximo evento a ser executado. [FUJ 2000]. Um problema que tal abordagem enfrenta é o *overhead* trazido pela grande quantidade de mensagens.

Uma forma simples de recuperação é utilizar o evento com menor *timestamp* nos *buffers* das filas de entrada e executá-lo. Sua localização é uma tarefa relativamente direta, pois a execução está bloqueada (*deadlock*) e novos eventos não estão sendo criados [FUJ 2000]. O mecanismo de detecção baseia-se na falta de comunicação entre os processos para indicar a ocorrência de *deadlock*.

Os protocolos conservativos diferem, ainda, na forma como suas principais variações são implementadas. Abordagens síncronas, como “janela de tempo conservativa” e *lookahead*, podem explorar os eficientes mecanismos de sincronização em *hardware* encontrados em ambientes multiprocessados. Por outro lado, abordagens assíncronas, como “prevenção” e “detecção e

recuperação" de *deadlock*, adaptam-se melhor a ambientes de memória distribuída ([FUJ 2000]).

A variação do protocolo, "janela de tempo conservativa" (Figura 2.4), introduz uma janela lógica no tempo dos eventos, na qual os que são por ela compreendidos possuem livre execução (são considerados seguros), sem necessitar de nenhum tipo de verificação; podem, assim, ser executados paralelamente [AYA 92]. O algoritmo divide-se em duas fases: a primeira determina o tamanho da janela, delimitando os eventos compreendidos como seguros, e a segunda é responsável pelo tratamento dos eventos contidos na janela em ordem cronológica.

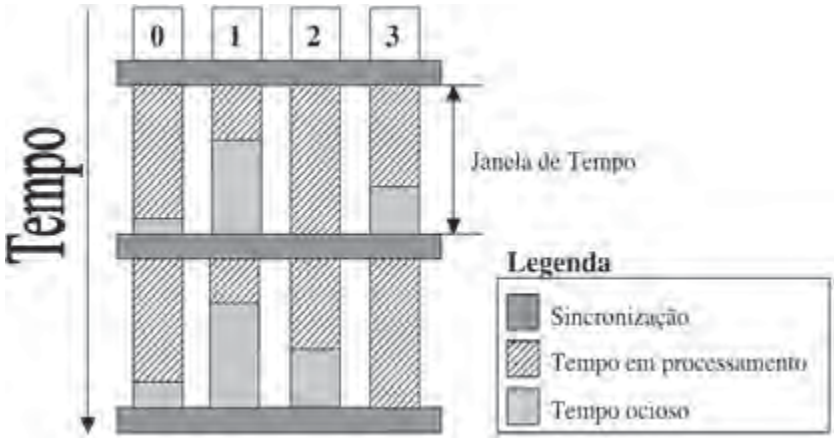


Figura 2.4 - Janela de tempo conservativa (adaptada de [MUK 97])

É importante notar que, ao final de cada fase, é necessária uma sincronização de todos os processos lógicos. Como as sincronizações podem ser frequentes, o desempenho do algoritmo está diretamente relacionado aos recursos de sincronização da arquitetura alvo.

Diferentemente da variação de "janela de tempo conservativa", a variação *Lookahead* é a habilidade de um processo em predizer quais serão os eventos gerados no futuro. Se um processo no tempo simulado $T_{atual}$ puder determinar que todos eventos serão gerados acima do tempo $T_{atual} + L$, diz-se que o processo tem um *lookahead* L. Esse é utilizado principalmente nos protocolos de prevenção de *deadlock*, nos quais essa informação é adicionada à mensagem nula, garantindo, assim, que nenhum evento seja enviado com marca de tempo menor que $T_{atual} + L$ [FUJ 2000].

Embora haja diversas variações, um dos grandes problemas das técnicas conservativas é que não exploram de forma completa o paralelismo nas aplicações de simulação, e também sofrem

invariavelmente, a possibilidade de ocorrência do *deadlock*. Algumas de suas variações evitam e outras permitem sua ocorrência, tratando-o após. De qualquer forma, ele é um problema a ser solucionado na construção de uma aplicação de simulação paralela conservativa.

Um outro ponto negativo são as ligações estáticas (canais de comunicação) por necessitar de um conhecimento profundo sobre o comportamento dos processos a serem simulados, para a inclusão de paralelismo e sincronização explícita nos processos a serem simulados. Dessa forma, uma das maiores desvantagens é a preocupação do programador com o mecanismo de sincronização para obter um bom desempenho [FUJ 90a].

### 2.3.2 Protocolos Otimistas

Diferindo dos protocolos conservativos, protocolos otimistas não necessitam determinar quando é seguro processar um evento. Apresentados, inicialmente, por Jefferson e Zowizral em 1985 ([FUJ 2000]), tais protocolos detectam e recuperam erros de causalidade, mas não impedem que esses ocorram. Permitem a ocorrência de erros LCC e, quando da ocorrência de um, chamam um procedimento para recuperar o processamento perdido [BAR 96].

Uma das vantagens desses mecanismos é permitir ao simulador explorar o paralelismo em situações nas quais é possível a ocorrência de erros de causalidade, mas que de fato, não ocorrem. Em observações empíricas da evolução de experimentos de simulação, foi constatado que, em grande parte das situações em que havia risco de erros LCC, eles não ocorriam. Dessa forma, ao tratar a ocorrência de erros de causalidade como uma exceção, e não como uma regra, pode-se explorar naturalmente o paralelismo intrínseco dos modelos e apresentar melhores desempenhos [FUJ 90a].

O principal problema dos protocolos otimistas está no momento da ocorrência de erros LCC, situação em que os LPs devem retornar a um estado seguro, anterior ao tempo atual simulado, para só então, executarem novamente os eventos. O problema pode tornar-se ainda maior quando o retrocesso de eventos inclui comunicações entre LPs. Nesse caso, todos os LPs retornam até encontrarem um estado seguro, podendo sofrer forte efeito dominó. Resta salientar que, para que os processos retornem a um estado seguro, eles devem ser armazenados em memória estável. O mecanismo de *rollback* exige que os LPs registrem os estados da simulação, acumulando

informações de seus eventos internos e externos de forma cronológica. Eventos externos dizem respeito às filas de recebimento e envio de mensagens, ao passo que os eventos internos tratam do conjunto de variáveis pertencentes ao processo. Aos estados salvos dá-se o nome de *checkpoint* ou ponto de recuperação [JAL 94, CAM 96, LUM 97].

Uma outra atribuição de cálculos do LP é o cálculo do avanço global da simulação (*Global Virtual Time* - GVT). Diferente dos protocolos conservativos, nos quais o menor LVT pode ser considerado como GVT, os otimistas empregam complexos cálculos para determinar o GVT, que serve, dentre outros motivos, como um dos quesitos para a liberação de memória de estados passados que não mais serão utilizados (*fossil collection*) [FUJ 2000].

O primeiro e mais conhecido protocolo otimista criado foi o *Time Warp* [JEF 85, MER 2011, BAU 2009, CHEN 2011, MER 2010]. Assim como os protocolos CMB utilizam a troca de mensagens para a sincronização, este protocolo determina os eventos com o menor *timestamp* entre os não processados nos LPs. Dessa forma, a execução da simulação pode avançar mais em um LP do que em outro, tornando o processo de *rollback* obrigatório de ser implementado.

No modelo "cancelamento agressivo" (*Aggressive Cancellation*), ao receber uma mensagem contendo um *timestamp* menor que o LVT atual, o processamento é bloqueado, retrocedido, e refeito. Tais mensagens, denominadas *stragglers*, acionam o mecanismo de *rollback*, retrocedendo todas as alterações realizadas prematuramente por processos, voltando com o relógio local a um ponto consistente da simulação[KAL 97, REIH 90].

Gafni, em 1998, ([FER 96, FUJ 2000]), propôs o "cancelamento preguiçoso" (*Lazy Cancellation*), como alternativa ao cancelamento agressivo. A nova estratégia não envia imediatamente as mensagens negativas, na chegada de um *straggler*. Diferentemente, faz uma comparação entre as mensagens enviadas e as novas geradas, enviando antimensagens apenas para as que são diferentes [FUJ 2000]. Tal técnica de retardar o envio das antimensagens, pode melhorar ou piorar o desempenho, gerando um *overhead* adicional, quando do processamento de uma mensagem, para verificar se existe a antimensagem correspondente. Em contrapartida, possibilita que os estados do sistema sejam salvos com menor freqüência, necessitando de menos memória que o cancelamento agressivo [OVE 91].

Outra alternativa ao *Time Warp* é uma técnica baseada numa forma muito semelhante ao

atraso no envio das mensagens, chamada de “reavaliação preguiçosa” (*Lazy Reevaluation*) proposta por West em 1988, ([FER 96, FUJ 2000]). Essa técnica difere da *Lazy cancellation* por operar com vetores de estado no lugar de mensagens, consistindo em avaliar se o estado do sistema é igual antes e depois da chegada e processamento de um *straggler*. Caso não tenham chegado novas mensagens, outra execução irá produzir os mesmos valores da original. Pode-se, após essa avaliação, “saltar” (*Jump Forward*) sobre os eventos, dependendo do exame do vetor de estados ([FER 96, FUJ 2000]). Um exemplo de que a *Lazy Reevaluation* pode representar uma grande melhoria no desempenho ocorre quando um evento denota uma tarefa apenas de leitura nos processos. Neste caso, não são necessárias novas reexecuções resultantes de consultas ([FER 96]).

Tais protocolos descritos acima, exploram a invariância de certos eventos em relação à ocorrência de *stragglers*, mesma propriedade em que se baseia o processo de *lookahead* dos protocolos conservativos. A vantagem dos otimistas é o fato de serem intrinsecamente capazes de explorar o *lookahead* de um problema, desobrigando o analista de se preocupar com tais aspectos no momento da implementação do modelo. Isso ocorre em virtude da realização de *rollbacks* somente quando os efeitos de um *straggler* provocarem alterações de estados relativamente significativas [FUJ 90a].

Da mesma forma que nos protocolos conservativos, são propostas nos protocolos otimistas as janelas de tempo. Neste caso, as janelas de tempo são utilizadas para permitir a execução somente dos eventos compreendidos nelas, evitando que algum LP avance em demasia e tenha de refazer muitos eventos. Este protocolo, originalmente proposto por Sokol, Briscoe e Wieland em 1988, [FER 96, FUJ 2000]), carece de processos com eficiência cientificamente comprovada para a geração do tamanho da janela.

Uma revisão do protocolo, denominada *Adaptative Time Warp concurrency control algorithm* – ATW, desenvolvida por Ball e Hyot em 1990, ([FER 96, FUJ 2000]), traz melhorias, permitindo que o seu tamanho seja adaptado dinamicamente através da suspensão temporária do processamento de eventos nos LPs que excederem um número de *rollbacks* predeterminado [FUJ 2000].

O mecanismo *Wolf Calls*, proposto por Madisetti, Walrand & Messerschmitt em 1988, ([FER 96, FUJ 2000]), representa uma das primeiras tentativas de redução do efeito dominó

causado pelos *stragglers*, onde mensagens *stragglers* provocam o envio de mensagens de controle especiais de alta prioridade (*wolf calls*) que param os processos afetados.

A literatura aponta desvantagens desta abordagem pois alguns processos podem ser afetados desnecessariamente. A utilização dessa técnica também apresenta casos onde o conjunto de processos afetados com sucesso é significativamente menor do que os que poderiam ser afetados, representando baixa eficiência. Além disso, o protocolo requer que se saiba qual é o tempo real de envio de mensagens de modo a calibrar a velocidade das *wolf calls*, tarefa que pode não ser trivial, dependendo do meio físico disponível e da taxa efetiva de uso [FER 96].

Outra alternativa é o *Direct cancellation*, proposto por Fujimoto em 1990 [FUJ 90a]. Este protocolo tem por objetivo evitar a geração de grandes cascatas de *rollbacks* em razão da velocidade de transmissão de mensagens contaminadas por informações inconsistentes. Ao invés de propor mensagens de alta prioridade, como no caso das *Wolf Calls*, Fujimoto propôs a manutenção de ponteiros para cada mensagem gerada. Assim, no caso de ocorrência de *stragglers*, o algoritmo somente tem de eliminar as mensagens através dos ponteiros e restabelecer os estados consistentes.

As vantagens apresentadas por este protocolo são a redução de *overheads* para geração e envio de antimensagens e boa eficiência no restabelecimento de estados consistentes para o sistema. Como desvantagem, há a necessidade da utilização de uma linguagem com suporte a ponteiros para entidades temporárias [FUJ 90a].

Embora os protocolos otimistas definem formas de evitar um comportamento de obstrução ao processamento, há também o compromisso de manter a quantidade de computações incorretas em um número bem menor do que as corretas. Caso a aplicação contenha um limitado paralelismo sobre a quantidade de processadores disponíveis, um elevado número de *rollbacks* é aceitável. Tal situação é oriunda do avanço indevido dos processadores na execução em aplicações nas quais a execução simultânea não poderia ser realizada.

Entretanto, em relação ao uso de memória, os protocolos otimistas sofrem restrições, e um dos principais problemas é a obrigatoriedade de realizarem o armazenamento do estado dos processos periodicamente [FUJ 92]. Além disso, outro aspecto a ser levado em consideração é a preocupação com possíveis erros arbitrários, os quais podem ser causadores de novas execuções. Tais incorreções computacionais podem entrar em laços infinitos, necessitando de mecanismos de

controle para uma intervenção a fim de interromper o sistema.

Os proponentes de métodos conservativos asseguram que os protocolos otimistas são mais complexos de implementar do que os conservativos, particularmente ao tentarem encontrar erros arbitrários [FUJ 90a].

### 2.3.3 Protocolos Hibridos

A maior motivação da existência de protocolos híbridos é a tentativa de unir os melhores aspectos de ambas as abordagens de protocolos. Essa seção apresentará três deles, sendo *Probabilist Optimism*, *Bouded Lag Algorithm with Filtered Rollbacks,* e o SRADS.

O *Probabilist optimism* foi proposto por Fersha [FER 96] e tem por objetivo explorar o comportamento dinâmico da simulação. Em cada passo, um grau justificável de otimismo é atribuído através de funções de verossimilhança calculadas sobre a probabilidade de que um evento tenha relação com outros. O protocolo, então, decide se um evento deve ser executado de forma otimista (imediatamente) ou conservativa (sendo retardado).

O *Bounded Lag Algorithm with Filtered Rollbacks*, proposto por Lubachevsky, Shwartz e Weiss em 1989, ([FER 96, FUJ 2000]), usa a noção de distância entre processos para determinar se um processo é seguro. Ocasionalmente, pode ser quebrada a restrição de uma distância mínima de segurança, levando a possíveis erros de causalidade. Nessas situações, o protocolo faz uso de um processo de rollback para restaurar os estados consistentes [FER 96].

Dickens e Reynolds, [FER 96, FUJ 2000], propuseram, em 1988 e 1990, um protocolo híbrido baseado em um algoritmo conservativo denominado SRADS (*Shared Resource Algorithm for Distributed Simulation*). Este protocolo sugere a utilização de um protocolo conservativo para processar eventos seguros e de outro otimista para processar os demais, sem a capacidade de propagação de subeventos. A justificativa para essa abordagem é o confinamento de possíveis *stragglers* a processos locais, evitando-se a necessidade de geração de antimensagens. O principal problema neste protocolo é o fato de eventos processados de maneira otimista não poderem ser utilizados pelo protocolo de sincronização para determinar novos eventos seguros [FUJ 2000].

Apesar das abordagens híbridas, questões relativas à qualidade dos protocolos otimistas e conservativos são freqüentemente levantadas. Regras gerais de superioridade de um ou de outro

mecanismo não podem ser formuladas por causa da grande diversidade dos processos a serem simulados; assim, devem-se evitar comparações diretas entre eles. Reforçando o argumento, o desempenho de comparação pode sofrer influências do *hardware* da implementação (a razão da velocidade das computações), do modelo de comunicação (FIFO, topologia da rede de comunicação, possibilidades operacionais), e finalmente, dos modelos de sincronização (unidade de controle global, variáveis compartilhadas).

### 2.3.4 Variações recentes de protocolos otimistas

Com a tendência e adoção de dispositivos de múltiplos núcleos, também conhecidos como *multi-core*, há diversas variações do Time Warp no sentido de adaptar o protocolo a essa nova realidade. Diversos trabalhos [CHEN 2011, MER 2010, BAU 2008, BAU 2009 , VAL 2008a, VAL 2008b] demonstram essa adaptação seguindo diversas estratégias e prioridades, além de realizarem trabalhos sobre partes do *Time Warp* original. Alguns deles particionam a simulação em *threads* e outros em processos do sistema operacional. Há casos da adaptação de partes de sincronização do protocolo para evitar *overhead*, de adaptações no mecanismo de cálculo de tempo global, adaptações para balanceamento de carga dinâmico, e até mesmo para distribuição de cálculos dinamicamente, em um ambiente onde o número de processadores é variável. Há também casos de adaptações para circunstâncias bem mais específicas, diretamente em *hardware*.

## 2.4 Redes de Filas

Tão importante quanto o modelo de simulação e sincronização aplicado, é a modelagem do sistema em si, ao qual se deseja prever resultados. Para tal, a seleção de um modelo apropriado de encadeamento de filas, reflete diretamente na qualidade dos dados obtidos no processo de simulação.

A característica de divisão de recursos limitados dentro de um sistema é extremamente importante na sua operação. Se não há disputa, a análise de desempenho torna-se mais fácil. Porém, casos reais de operação refletem a realidade da divisão constante de recursos e necessidade contínua de aumento de disponibilidade/velocidade. Para ilustrar esse compartilhamento, tem-se a situação em que tarefas em um sistema computacional, competem pelo prestador de serviço (UCP), unidades de entrada e saída, canais, entre outros; mensagens em redes de comunicação competindo por

enlaces, *buffers*, permissão, janelas; e tarefas em uma linha de montagem competindo por ferramentas, áreas de estocagem, robôs, mecanismos de transportes, entre outros [SOA 92].

Basicamente, uma rede de filas existe quando há um conjunto de prestadores de serviços e clientes, os quais recebem serviços nos centros. Um centro de serviço é constituído de um ou mais servidores, representando os recursos do sistema, que prestam serviço aos usuários, e uma linha de espera, denominada fila, para os clientes que estão esperando pelo serviço. Como exemplo de centro de serviço pode-se considerar a UCP de um sistema computacional. Na Figura 2.5 são apresentados os tipos de sistemas de redes de filas: na Figura 2.5(a) um modelo com apenas uma fila e um servidor, na Figura 2.5(b) uma fila e vários servidores, na Figura 2.5(c) várias filas e um servidor e na Figura 2.5(d) um sistema constituído por múltiplas e filas e servidores.

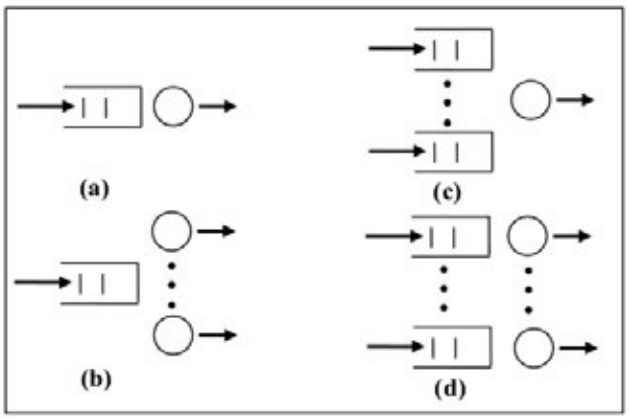


Figura 2.5 – Tipos de Redes de Filas

Pelo fato de serem redes de filas, deve haver uma política de escalonamento para escolha do usuário que será o próximo a ser atendido pelo centro de serviço. Exemplos são FIFO (*First In, First Out*) e LIFO (*Last In, First Out)*, caracterizadas pelos seguintes componentes [SAN 94]: padrão de chegada, distribuição do serviço, capacidade de serviço, disciplina da fila, número de servidores e tamanho da área de espera [SOA 92].

Alguns sistemas de fila são compostos por um número de subsistemas interligados em rede. Em tais sistemas os clientes recebem serviço em mais de um componente do sistema, antes de terem seus requisitos de serviço totalmente atendidos. Tais sistemas são denominados redes de fila. A notação padrão para sistemas baseados em redes de filas é: *A/S/c/k/m*, onde *A* representa a distribuição do tempo de chegada, *S r*epresenta a distribuição de tempo de serviço, *c* é o número de servidores, *k* é o número máximo de clientes no sistema e *m* é o número de clientes disponíveis na

fonte [MAC 87].

O padrão de chegada é a freqüência com que os clientes chegam ao sistema, podendo ser regular ou aleatória, sendo essa última a mais adequada a simulações reas. Na freqüência aleatória, os dados podem ser obtidos através de tabelas ou pela utilização de uma distribuição de probabilidade. Algumas distribuições importantes são [SOA 92]:

- Distribuição Uniforme;
- Distribuição Triangular;
- Distribuição Exponencial;
- Distribuição de Poisson;
- Distribuição Normal;
- Distribuição de Erlang.

O montante de serviço solicitado por um cliente é denominado demanda de serviço ou simplesmente carga de trabalho. A unidade de serviço varia de acordo com a natureza do servidor, dos clientes, e da própria concepção da simulação.

A demanda de serviço por clientes segue uma distribuição chamada de distribuição de serviço. Além disso, deve-se especificar também a capacidade do servidor, a qual representa o quão rápido o servidor realiza o serviço.

A ordem com que os clientes são retirados da fila é denominada disciplina da fila, e para tal, existem vários algoritmos de escalonamento usados para decidir qual cliente deve entrar em serviço quando um servidor se encontra disponível. Entre esses, cabe elencar:

- FCFS (f*irst-come first-served*): consiste em uma fila padrão, a ordem de chegada é a ordem de atendimento no centro de serviço;
- LCFS (*last-come first-served*): o último usuário a chegar ao centro é o que entrará em serviço assim que houver disponibilidade do servidor;
- RR (r*ound-robin*): cada usuário é atendido por um pequeno intervalo de tempo denominado *quantum*. Caso não tenha sido suficiente, o usuário volta para o final da fila e o usuário seguinte entra para receber seu quantum de serviço. O processo se repete até que o serviço requisitado pelo usuário se complete;

PS (p*rocessor sharing*): todos os usuários dividem a capacidade do centro de serviço, como se executassem em paralelo, sob a perspectiva de uma fila infinita.

Para a tarefa de prestação de serviço, pode-se ter um ou mais servidores, alocando-se mais que um cliente ao mesmo tempo. Para o caso quando pelo menos um servidor está disponível, deve-se escolher qual deles irá atender o cliente, e essa disciplina pode ser definida de diversas formas, como por exemplo, selecionando o que está desocupado a mais tempo, ou pode ser escolhido aleatoriamente ou, ainda, segundo algum tipo de prioridade.

Por fim, nem todos os sistemas têm uma área de espera infinita para as filas, ainda mais quando se trata da alocação de memória para armazenar o histórico de filas. Quando clientes estão enfileirados em número excessivo para um espaço finito de área de espera, alguns clientes podem ser perdidos ou rejeitados, dependendo do ambiente em que se quer simular.

## 2.5 Ordenação de eventos em sistemas distribuídos

Essa seção trata sobre a ordenação de eventos em sistemas distribuídos, abordando assuntos necessários para o tema, como técnicas de ordenação parciais de eventos, conceitos de relógios lógicos, e algoritmos para ordenação total de eventos

Para a implementação do sincronismo em DSM, é necessário observar os conceitos propostos por Lamport [LAM 78], sobre mecanismos de sincronização em ambientes distribuídos, baseados no conceito de tempo.

Tal conceito é fundamental na nossa maneira de pensar, sendo derivado do conceito mais básico em relação à ordem de acontecimento de eventos. A ordenação temporal de eventos é subjetiva no modelo de sistemas simulados. Entretanto, considerando um sistema distribuído, este conceito deve ser examinado novamente, no intuito de definir um mecanismo lógico para ordenação eventos.

Um sistema distribuído consiste da coleção de processos distintos que são espacialmente separados e que se comunicam um com os outros através da troca de mensagens, como por exemplo, uma rede de computadores interconectados. Um computador único também pode ser visto como um sistema distribuído, no qual uma unidade central de controle, as unidades de memória e os canais de entrada e saída, são processos separados. Mais especificamente, um sistema é distribuído se o atraso de transmissão de mensagens entre eventos é perceptível, em comparação entre eventos em um processo único [LAM 78]. Um dos problemas inerentes a sistemas distribuídos, é a dificuldade, ou muitas vezes a impossibilidade de dizer quais dos eventos ocorreram primeiro.

## 2.5.1 Ordenação parcial de eventos

Embasado em teorias físicas do tempo, é comum afirmar que, dados dois eventos *A* e *B*, se um evento *A* aconteceu antes de um evento *B*, *A* aconteceu num tempo anterior ao tempo de *B*. Entretanto, para um sistema ser especificado corretamente, este deve ser descrito através de eventos observáveis dentro do próprio sistema. Uma vez que a especificação é baseada no tempo físico, e o sistema contém relógios reais, surge o problema da falta de certeza e precisão do tempo real, criado através dos mecanismos internos de funcionamento dos relógios físicos. Portanto, com tais problemas, a definição de *"happens before"*, que defende a ordem de acontecimento de eventos em um sistema distribuído, defendida por Lamport [LAM 78], baseia-se numa relação lógica de acontecimentos, sem o emprego de relógios físicos.

Dado um sistema composto por uma coleção de processos onde cada um deles consiste de uma sequência de eventos (evento, dependendo do nível da aplicação, e do nível da abstração, pode ser caracterizado como a execução de um subprograma ou mesmo a execução de uma simples instrução de máquina), define-se por *"happened before"*, um evento *A* que tenha ocorrido antes de um evento *B*, numa sequencia de eventos despachados ou recebidos por um processo. Esta relação é definida por $A \rightarrow B$ .

Formalmente, a relação $\rightarrow$ é definida por três condições:

- Se *A* e *B* são eventos no mesmo processo, e *A* precede *B*, então $A \rightarrow B$
- Se *A* é o remetente de uma mensagem em um processo, e *B* é o receptor da mesma mensagem, em outro processo, então $A \rightarrow B$
- Se $A \rightarrow B$ e $B \rightarrow C$ , então $A \rightarrow C$ . Dois eventos distintos são concorrentes se A $\nrightarrow$ B e B $\nrightarrow$ A.

Outra maneira de apresentar a definição é dizer que $A \rightarrow B$ significa que é impossível para um evento *A* causar efeito em um evento *B*. Dois eventos são concorrentes se nenhum pode causar efeito no outro.

## 2.5.2 Relógios Lógicos

Para superar os obstáculos impostos por relógios físicos, são apresentados relógios lógicos, e para a sua definição dentro do sistema a ser observado, será necessário definir um ponto abstrato de visão, onde um relógio é simplesmente uma maneira de associar um número a um evento, e onde tal

número é assimilado como uma noção do tempo em que o evento ocorreu. Mais precisamente, definindo um relógio $C_i$ para cada processo $P_i$, $C_i(A)$ é a função que representa um evento *A* no processo, em um dado tempo [TAN 2002].

Com essa função relógio, e dadas as considerações sobre o que tal função tem por significado correto no sistema, a definição tem por base a ordem em que os eventos ocorrem. A condição é que se um evento *A* ocorre antes do evento *B*, então *A* deve ocorrer em um tempo anterior que o tempo de *B*, como a formalização a seguir:

**Condição de relógio**: Para quaisquer eventos *A*, *B*: Se $A \rightarrow B$ , então $C(A) < C(B)$.

As conseqüências dessa condição são observadas por duas proposições:

- Se *A* e *B* são eventos no processo $P_i$, e *A* é anterior que *B*, então $C_i(A) < C_i(B)$.
- Se *A* é o remetente da mensagem pelo processo $P_i$, e *B* é o receptor desta mensagem, pelo processo $P_j$, então $C_i(A) < C_j(B)$.

Para garantir que todos os relógios de um sistema satisfaçam a condição acima, será necessário certificar que estes também assumem as seguintes condições [LAM 78]:

- Cada processo $P_i$ incrementa $C_i$ entre dois eventos sucessivos.
- Se um evento *A* enviar uma mensagem *M* para um processo $P_i$, então a mensagem *M* contém o timestamp $T_m = C_i(A)$. Ao receber a mensagem *M*, o processo $P_j$ define $C_j$ maior ou igual ao seu valor atual e maior que $T_m$.

Com tais afirmações acima, é garantido um sistema funcionalmente correto de relógios lógicos.

## 2.5.3 Ordenação total de eventos

Pode-se utilizar um sistema de relógios baseados na condição de relógio para estabelecer a ordem total no conjunto de todos os eventos do sistema, através da ordenação do tempo. Para isso, a ordenação total de eventos é definida através da relação $\Rightarrow$: Se *A* é um evento no processo $P_i$, e *B* é um evento no processo $P_j$, então $A \Rightarrow B$ , se e somente se $C_i(A) < C_j(B)$ ou $C_i(A) = C_j(B)$ e $P_i \prec P_j$. Tal afirmação define a ordenação total e a Condição de Relógio implica que se $A \rightarrow B$ , então, $A \Rightarrow B$ . A relação $\Rightarrow$ é uma maneira de complementar a relação *"happened before"*, parcialmente ordenada, em ordenação total [LAM 78].

Ordenar totalmente os eventos pode ser muito útil na implementação de sistemas distribuídos. De fato, a razão para implementar um sistema correto de relógios lógicos é obter tal ordenação total.

A ordenação total de eventos pode ser conseguida, em um sistema distribuído, de diversas formas. São utilizados algoritmos com nós centralizados para sincronização, ou algoritmos de sincronização distribuída, onde oferecem recursos para que todos os nós do sistema estejam sincronizados através de marcas de *timestamps*, a fim de prover a ordenação.

No caso de algoritmos centralizados, há um nó central que garante a distribuição de recursos e ordenação total de eventos, porém pode apresentar problemas de gargalo de rede e intolerância a falhas, além de outras características que prejudicam sua abordagem.

No caso de algoritmos distribuídos, dadas algumas regras, há a participação ativa de todos os nós e também comunicação de todos para todos através de mensagem com *timestamps* anexados, a fim de garantir o uso de recursos, e que tal uso seja feito de maneira correta [LAM 78].

## 2.6 Framework Terracotta

Para prover um ambiente paralelo, distribuído e compartilhado sobre a rede, o *framework* Terracotta será utilizado. Tal *framework* é baseado na linguagem de programação Java, de quarta geração e multiplataforma.

O desenvolvimento de aplicações Java é simples quando o ambiente é baseado em uma única *Java Virtual Machine* (JVM). Entretanto, com o aumento de requisitos de software e simulações mais complexas, tem-se motivado a distribuição de aplicações em mais do que uma JVM, tornando-as disponíveis e escaláveis. Tal motivação tem causado uma proliferação de soluções que tem sobrecarregado as arquiteturas de muitas aplicações Java.

Como uma solução para tal problema, o *framework* Terracotta propõe memórias anexadas via rede (*Network-Attached Memory* – NAM) onde a memória *heap* de uma máquina virtual Java (arbitrariamente grande), é anexada via rede, permitindo diversas *threads* em diversas JVM, em diversas máquinas, interagirem umas com as outras, como se estivessem na mesma JVM. Com a utilização dessa abordagem, a adição de servidores para aumento de disponibilidade do *cluster*, torna-se simples, e rápida, além de prover escalabilidade e alta confiabilidade sem modificações na arquitetura da aplicação, ou sobrecarga de banco de dados, ou desafios para o desenvolvedor [TER

2008a].

Na figura 2.6 é ilustrado o posicionamento do *framework* numa arquitetura integrada de servidores.

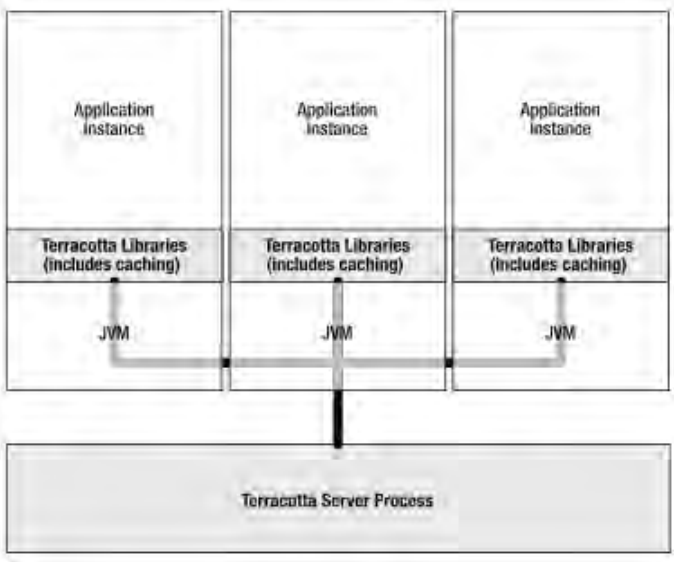


Figura 2.6 – Integração e instâncias da aplicação no *framework* Terracotta [TER 2008b].

No caso da utilização de JVM única, as *threads* interagem umas com as outras através de modificações feitas em objetos na memória *heap* e através de primitivas de concorrência em linguagens de programação, como blocos do tipo sincronizados ou métodos primitivos de objetos, como *wait()*, *notify()*, e *notifyAll()*. O *framework* Terracotta permite *threads* em um *cluster* de JVMs interagirem umas com as outras, através de mecanismos transparentes entre JVMs utilizando as mesmas facilidades *built-in*, estendidas em um contexto de *cluster*. Tais capacidades de *clustering* são injetadas atravéss de manipulação de *bytecodes* das classes da aplicação, em tempo de execução, portanto, não é necessário utilizar código de uma API externa ou estruturas explícitas de paralelismo [TER 2008a].

## 2.6.1 Arquitetura Terracotta

Os conteúdos e acessos das *heaps* anexadas via rede são controlados por uma matriz de servidores que empregam o *framework* conectados por uma rede de alta velocidade. Na figura 2.7 é apresentada uma imagem ilustrativa sobre a composição básica da arquitetura do *framework* Terracotta.

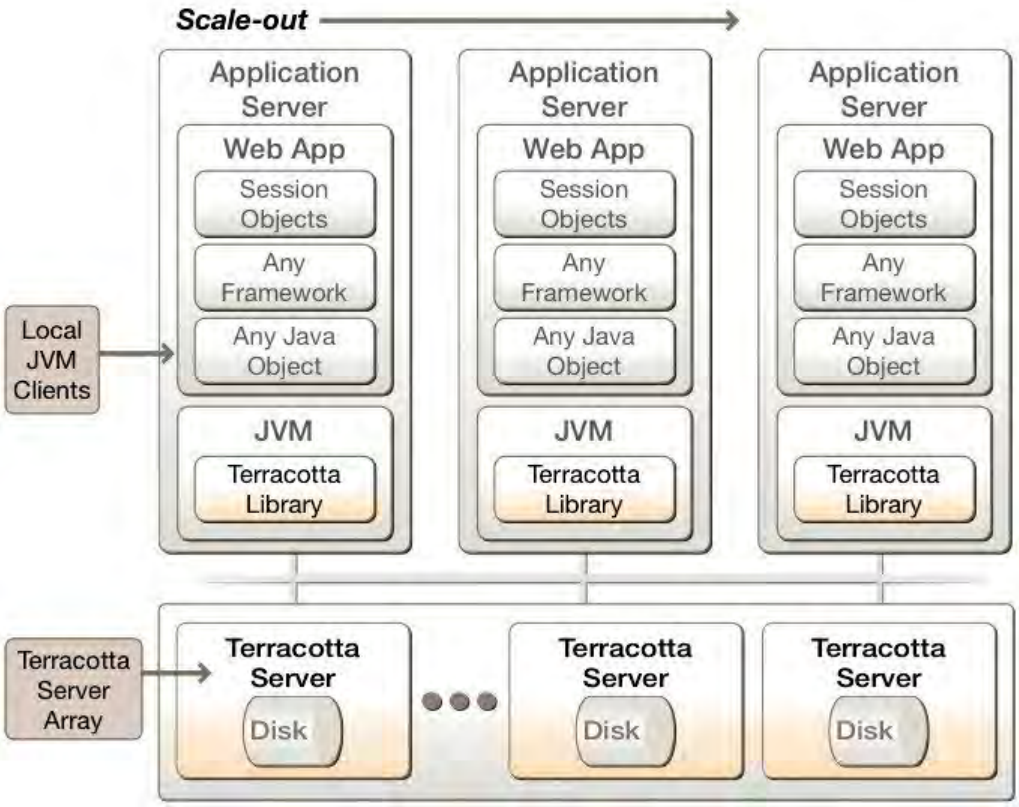


Figura 2.7 – Arquitetura Básica do *framework* Terracotta [TER 2008a].

Uma segunda preocupação da arquitetura proposta é a característica *"hot standby"* para a conexão ou remoção de nós da matriz de servidores, facilitando o tratamento em relação a tolerância a falhas ou disponibilidade. O servidor em *standby* se torna ativo imediatamente quando uma falha de uma instância de servidor ocorre, e todas as JVMs ativas são automaticamente reconectadas a um novo servidor, fazendo com que o trabalho continue de forma transparente. Devido a capacidade de persistir todos os estados de todas as *heaps* anexadas via rede, o sistema todo pode ser completamente desligado e reiniciado a partir de um estado anterior ao desligamento.

## 2.6.2 Instrumentação de *Bytecode*

O gerenciamento da NAM acontece com a injeção de código no código pré-compilado da aplicação, utilizando técnicas padrões de manipulação de *bytecodes*, conforme as classes são carregadas na JVM. Os *bytecodes* de uma classe são interceptados no tempo de carga e examinados pela API, de forma transparente. Em seguida, os *bytecodes* são modificados de acordo com a configuração de funcionamento passada para a JVM.

A fim de manter as modificações de objetos, as instruções *bytecode* PUTFIELD e

GETFIELD sofrem sobrecarga. Instruções PUTFIELD são inseridas para recuperar dados do servidor via campo, conforme necessário (se ainda não houver recuperado a referência do objeto do servidor e instanciado na *heap* física local).

Para gerenciar a coordenação de *threads*, as instruções *bytecode* MONITORENTER e MONITOREXIT também sofrem sobrecarga, bem como as instruções INVOKEVIRTUAL para métodos *Object.wait()*, e *Object.notify()*. MONITORENTER significa a requisição de uma *thread* para um monitor. Uma *thread* bloqueará nessa instrução até que seja garantido o uso exclusivo do objeto. Uma vez que a trava é garantida, tal *thread* possuirá uma trava exclusiva em tal objeto até que a instrução MONITOREXIT seja executada. Se o objeto em questão for um objeto compartilhado, o *framework* certifica-se que, além da requisição da trava local neste objeto, a *thread* também bloqueará até receber permissão de uso exclusivo, através de todas as JVMs conectadas em tal objeto. Quando a *thread* libertar o trava local, o trava compartilhada também será dissolvida.

Os locais de chamada dos métodos *wait()* e *notify()* também são instrumentados. Quando um método *wait()* é chamado em um objeto compartilhado, uma instância do servidor Terracotta adiciona a *thread* de chamada a um conjunto de *thread* em todas as JVMs conectadas, aguardando tal objeto. Quando um dos métodos *notify()* é chamado, a instância do servidor certifica que o número apropriado de *threads* através do *cluster* seja notificado. Quando o método *notifyAll()* é executado, a instância do servidor notifica todas as *threads* que estão aguardando no objeto, em todas as JVMs a serem notificadas, dentro do *cluster*.

### 2.6.3 Raíz e grafos de objetos compartilhados

O *framework* Terracotta possui um grafo de objetos compartilhados, para a manipulação de todos os objetos, variáveis e campos que devem ser controlados e replicados com consistência ao longo do *cluster*.

Objetos compartilhados iniciam-se na raiz do grafos de objetos compartilhados do *framework*. A raiz é identificada por um conjunto particular de um ou mais campos e é declarado na configuração da API com um nome único.

Em relação a consistência dos dados, quando uma raiz é inicialmente instanciada, o objeto raiz e todos os seus objetos alcançáveis por referência, do nível mais alto para o mais baixo,

tornam-se também objetos compartilhados. Seus campos de dados são enviados para uma instância folha qualquer, e armazenados. Uma vez que um nó raiz é replicado em qualquer JVM, todas as outras associações para tal campo são ignoradas e o valor do objeto raiz compartilhado é associado para o campo. As associações para os campos raiz feitas no construtor de outras instâncias são ignorados devido àquela raiz já ter sido criado pela primeira instância da aplicação, a fim de manter a consistência do sistema como um todo. Ao invés de associar aos campos raiz os valores dos argumentos passados pelo construtor, a API do *framework* recupera o servidor raiz, e associa a referencia para este, para a variável raiz em questão. Isto representa a única maior mudança da semântica da aplicação, efetuada transparentemente pelo *framework*.

Quando um objeto não compartilhado torna-se referenciável de um objeto compartilhado, o novo objeto e o grafo completo de objetos alcançáveis por referência pelo novo objeto tornam-se compartilhados na *heap* anexada pela rede. Uma vez que um objeto torna-se compartilhado, este é associado com um único objeto de identificação pelo *framework*, e é compartilhado enquanto permanece em seu ciclo de vida. Quando um objeto torna-se não alancável por qualquer grafo de raiz e não há instâncias de nenhuma JVM anexada, então este é elegido para remoção pelo coletor de lixo distribuído, na matriz de servidores.

### 2.6.4 Sincronização, Travas, e modificações em objetos

Métodos sincronizados, blocos sincronizados, e métodos declarados para serem bloqueados na configuração inicial do *framework* apresentam ligações nas transações do sistema. A noção de uma transação é um pouco diferente de uma transação JTA – *Java Transaction API*. É muito mais parecida com as transações utilizada no modelo de memória Java.

Como discutido anteriormente, uma instrução MONITORENTER em um objeto compartilhado é estendida para também ser uma requisição para uma trava compartilhada, tal que a *thread* de chamada irá bloquear até que seja garantida em ambos: na trava local, e na trava compartilhada. Todas as modificações feitas em objetos entre um MONITORENTER e o correspondente MONITOREXIT são coletadas pelo *framework* em um registro de transação local. O *framework* garante que todas as modificações feitas em todas as transações associadas com uma trava de um objeto particular em todas as JVMs dentro do *cluster* serão aplicadas localmente, antes que a *thread* tenha permissão para proceder para uma próxima instrução MONITORENTER [TER

2008a].

### 2.6.5 Replicações de mudanças

Em relação às replicações de alterações ao longo do *cluster*, o *framework* gerencia no nível de campos de objetos e enviadas através da rede somente quando for necessário. As transações que contém modificações nos objetos possuem somente os dados dos campos que foram alterados. Tais transações são enviadas para uma instância de servidor e para as JVMs anexadas para manter o *cluster* consistente. A instância do servidor somente envia a transação para outra JVM que tem objetos instanciados na *heap* que são representados na transação. Igualmente, somente é enviada a porção de transação para as JVMs nas quais a transação deve ser aplicada.

### 2.6.6 Identidade de objetos e serialização

Devido o fato das modificações em objetos serem monitoradas no patamar de campos, e transações conterem fragmentos de objetos - ao invés do grafo do objeto todo – o *framework* não utiliza a serialização Java para replicar modificações nas mudanças de objetos.

O gerenciamento de modificações do *framework* somente transfere através da rede do *cluster* os dados que realmente foram modificados, ao invés da serialização completa dos grafos dos objetos. Com essa medida, também há a preservação da identidade do objeto [TER 2008b].

A preservação da identidade do objeto torna o comportamento de aplicações rodando em diversas JVMs conectadas a *heap* anexada pela rede, normal, igualmente a aplicações de JVM única, permitindo a implantação da arquitetura da aplicação de maneira transversal para o modelo de programação e os objetos de aplicação. Questões de infra estrutura tais como quantas máquinas são implantadas para a aplicação servidora são logicamente diluídas na camada do *framework* Terracotta, no nível JVM, abstraindo a infra estrutura que pertencem.

### 2.6.7 Heap Virtual

Além da habilidade de compartilhar objetos e sinalizar *threads* através de JVMs, o *framework* também permite o uso da *heap* JVM local, para grafos de diversos tamanhos. Conforme o crescimento de grafos de objetos compartilhados, este pode ter um tamanho maior do que o de uma *heap* de uma única JVM. O *framework* abstrai tais dificuldades, manipulando esse ajuste através da poda da instância local para um objeto de grafo compartilhado de acordo com os padrões

de uso na instância. Há também uma janela configurável no grafo do objeto compartilhado, tal que as partes que não encaixam em uma certa porcentagem da *heap* serão espalhadas de acordo com a política de cache. Conforme tais partes são requisitadas, elas são automaticamente recuperadas na JVM, através da matriz de servidores Terracotta.

Esta característica permite a alocação de grafos de objetos arbitrariamente grandes, em tamanhos padrões de *heap*. Também permite particionamento flexível de dados, em tempo de execução.

### 2.6.8 Casos de Uso

Como já descrito anteriormente, o *framework* possibilita o cenário em que diversos servidores físicos trabalhem conjuntamente representando somente um servidor lógico para a aplicação em questão. Para esse cenário, há diversos casos de usos que podem ser listados, como:

- *Cache* distribuído;
- Armazenamento temporário de dados de SGBS;
- Replicação de sessão para aplicações Web;
- Particionamento de carga de trabalho.

Para o caso da aplicação e emprego no trabalho proposto, o cenário de particionamento de carga de trabalho é o ideal, e pode ser comparado a diversas outras abordagens como a tecnologia *MapReduce* [DEAN 2004].

Conforme explanado em [TER 2008b], o particionamento de carga de trabalho proposto pelo *framework* Terracotta tem como objetivo a estratégia dividir para conquistar. A ideia parte da utilização de estruturas nativas do Java onde, através de troca de mensagens se consiga o coordenação de threads, ou do uso de JMS, ou mesmo em alguns casos a utilização de *sockets* conjuntamente com estruturas de dados compartilhados.

Essencialmente, uma aplicação pode espalhar pela rede de servidores, cargas de trabalho como pesquisas ou atualizações de grandes quantidades de dados, sem a necessidade de filas de mensagens ou estrutura de serviços. Somente a utilização de *multithreading* em uma aplicação simples é suficiente.

A figura 2.8 ilustra o caso da distribuição de cargas através de uma rede de servidores, utilizando estruturas básicas do Java.

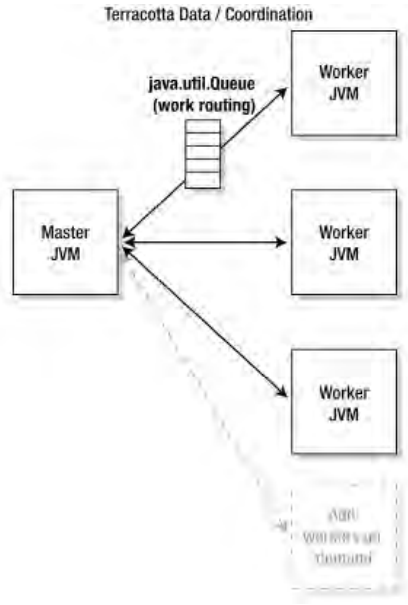


Figura 2.8 – Cenário de distribuição de carga utilizando primitivas do Java [TER 2008b]

Portanto, embora a camada adicional de manipulação de *bytecodes* imposta pelo funcionamento básico do *framework* possa ocasionar perdas de desempenho e falta de controle do funcionamento e mecanismo de distribuição de cargas, em nível baixo, os recursos obtidos através da instrumentação interna e memória anexada via rede além de sua utilização na linguagem de programação Java, tornam simples a manipulação da escalabilidade e a alta disponibilidade, favorecendo a utilização do *framework* como ambiente de simulação paralela, fomentando o ambiente de memória compartilhada e distribuída.

Com o objetivo de facilitar a criação do ambiente DSM, o *framework* será empregado no trabalho, para direcionar o foco exclusivamente na concepção arquitetural e lógica do protocolo de simulação.

## 2.7 Considerações Finais

O perfil da simulação depende muito de suas características próprias e específicas, e também da decisão dos recursos para modela-lá. A escolha dos recursos que caracterizam o encadeamento de clientes, além dele próprio, podem alterar drasticamente o perfil dos resultados obtidos.

Como ferramentas para facilitar essa modelagem, que depende muito do problema a ser

explorado, tem-se como opções os protocolos conservativos, que são menos complexos de implementar, pois possuem consumo moderado de memória, alguns mecanismos como de *lookahead*, e adaptam-se melhor a grandes modelos de simulação, entretanto possuem um fraco paralelismo de execução e protocolos otimistas, que levam vantagem ao proporcionarem uma melhor exploração do paralelismo dos modelos, pois particularidades como a granularidade do ambiente computacional, topologia e localidade temporal de eventos não necessitam ser conhecidas a fundo para uma eficiente comunicação, entretanto há uma enorme complexidade na implementação, além do consumo agressivo de memória e difícil obtenção do GVT.

Portanto, os problemas físicos reais impostos pelos mecanismos de contagem de tempo, implicam na imprecisão da contagem quando num ambiente distribuído. Para que haja a ordenação, coordenação, e informação do tempo global de simulação, é vital a criação e gerência de mecanismos lógicos para tal fim.

Dessa forma, com a criação do mecanismo de relógios lógicos, aliando um número a um evento, e utilizando-o como parâmetro de contagem sucessiva de acontecimentos, tem-se um mecanismo que, uma vez gerenciado corretamente, é válido para o andamento geral e correto da simulação computacional.

E é dessa forma que a plataforma de simulação, objetivo desse trabalho, foi estruturada, utilizando os conceitos mencionados nesse capítulo, bem como a utilização de tempo virtual.

# Capítulo 3

# Descrição e Validação de uma Plataforma baseada em Protocolo Otimista

## 3.1 Considerações Iniciais

O projeto de implementação e adaptação da plataforma de simulação paralela otimista, utilizando a linguagem de programação comercial Java, tem como base toda a revisão bibliográfica descrita nesse documento, além do emprego dos conceitos de relógios lógicos e tempo virtual, propostos por Lamport.

A arquitetura de software do projeto foi concebida em um ambiente computacional distribuído, abstraído através da memória compartilhada e distribuída, utilizando o *framework* Terracotta, baseado na linguagem de programação Java. Com essa arquitetura, foi possível efetuar diversas análises sobre uma simulação distribuída, como por exemplo estudo detalhado sobre controle de simulação, eficiência do algoritmo, consistência e tratamento de erros de causalidade na simulação paralela, tempo útil de trabalho, tempo médio de fila de clientes, cálculo do tempo global virtual da simulação através de tempos locais de simulação.

A figura 3.1 apresenta um diagrama de organização geral da plataforma de simulação.

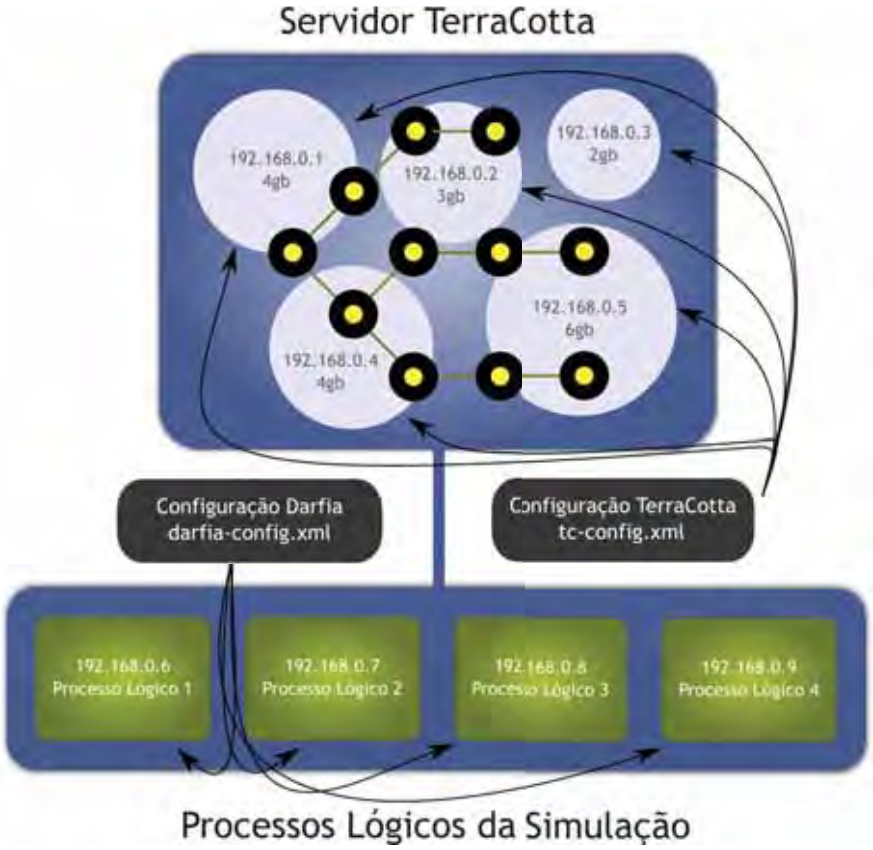


Figura 3.1 – Organização geral da simulação

A análise, construção e validação da plataforma foi efetuada da seguinte forma:

- Análise de seu funcionamento e comportamento. Validação da análise com as premissas definidas por Jefferson [JEF 85], elencadas em [SPO 01], em relação à implementação do algoritmo de protocolo de simulação otimista paralela, no que diz respeito às situações que o protocolo deve estar apto a responder, quando elas acontecerem durante sua execução;
- Validação do algorítimo sequencial de cada unidade de trabalho do protocolo, realizando testes sequenciais simples, ou seja, sem encadeamento de filas. Os resultados obtidos do protocolo foram comparados linearmente com os resultados obtidos através da extensão funcional SMPL [MAC 87].

Para descrever a estrutura e validação do sistema, o texto foi organizado com a apresentação de alguns padrões de projetos empregados no desenvolvimento do sofware, bem como a aplicação desses, na seção 3.2. A seção 3.3 faz-se um levantamento de configuração da simulação. A seção 3.4 é apresentado o módulo de gerenciamento. A seção 3.5 são elencadas as classes componentes da simulação, responsáveis por desmembrar o funcionamento de cada processo lógico. A seção 3.6

apresenta-se o gerenciamento de filas. A seção 3.7 são apresentadas as possibilidades de geração de números aleatórios, disponíveis através de configuração da simulação, e na seção 3.8 é descrito o módulo de utilitários, disponíveis para facilitar a construção do projeto. A seção 3.9 apresenta exemplos da utilização da plataforma. A seção 3.10 apresenta as validações da implementação e finalmente a seção 3.11 apresenta algumas considerações finais.

## 3.2 Padrões empregados no desenvolvimento

Alguns padrões de projeto foram empregados no desenvolvimento do trabalho, tanto na implementação da plataforma, bem como no desenho arquitetural da mesma. Estes podem ser divididos nas seguintes áreas descritas a seguir, baseado no modelo de padrões de projeto **GoF** [GAMMA 95].

- ***Singleton***

  Como padrão de criação, o *Singleton* propõe a solução para o seguinte cenário, definido na tabela 3.1.

**Tabela 3.1: Definição do padrão de projeto *Singleton***

| | |
|---|---|
| *Intenção:* | Certificar-se que uma determinada classe possui somente uma instância no escopo da aplicação como um todo e provê somente um ponto global de acesso a esta. |
| *Motivação:* | É importante que classes possuam exatamente uma instância durante toda a vida de existência de uma aplicação. Esta classe é responsável por se rastrear e certificar-se que não há nenhuma outra instância de si mesma. |
| *Aplicabilidade:* | Quando somente uma instância de uma classe é permitida e deve ser acessível para seus clientes a partir de um ponto de acesso conhecido e formalizado. |
| *Participantes:* | Define uma operação de instância e permite que seus clientes acessem somente essa instância. Pode ser responsável por criar sua única instância ao invés de depender de algum outro mecanismo. |
| *Consequências:* | 1. Acesso controlado a sua instância. |

| | |
|---|---|
| | 2. *Namespace* reduzido.<br>3. Refinamento de operações e representação.<br>4. Mais flexível do que operações de classes, no sentido da utilização de métodos estáticos. |
| *Utilização no protocolo:* | Será utilizado na criação da classe *DarfiaSimulation*, responsável pela gestão da plataforma, bem como na biblioteca de evidências *(logs).* |

- ***Observer***

Como padrão comportamental, o *Observer* propõe a solução para o seguinte cenário, definido na tabela 3.2.

**Tabela 3.2: Definição do padrão de projeto *Observer***

| | |
|---|---|
| *Intenção:* | Definir uma dependência do tipo um-para-muitos entre objetos, portanto quando um objeto muda de estado, todos seus dependentes são notificados e atualizados automaticamente. |
| *Motivação:* | Uma consequência comum de particionar um sistema em coleções de classes de cooperação é a necessidade de manter a consistência entre os objetos relacionados. A utilização da estratégia de observações permite a implementação de baixo acoplamento, facilitando o reuso das classes de cooperação. |
| *Aplicabilidade:* | Entre outros, quando uma mudança em um objeto reflete na mudança de outros objetos e não há o conhecimento de quanto e quais objetos sofrerão os efeitos da mudança. |
| *Participantes:* | O assunto da mudança que é observado e uma interface de atualização para os objetos que devam ser notificados. |
| *Consequências:* | 1. Acoplamento abstrato entre assunto e observador.<br>2. Suporte para comunicação coletiva.<br>3. Atualizações não esperadas. |
| *Utilização no protocolo:* | Será utilizado na troca de eventos entre elementos processadores, de forma a garantir atualização de estado da mensagem através de sua rota. |

- ***Factory Method***

Como padrão de criação, o *Factory Method* propõe a solução para o seguinte cenário, definido na tabela 3.3.

**Tabela 3.3: Definição do padrão de projeto *Factory Method***

| | |
|---|---|
| *Inteção:* | Encapsular a escolha da classe concreta a ser utilizada na criação de objetos de um determinado tipo . |
| *Motivação:* | Permitir um modelo plugável de componentes específicos, para certas situações, a fim de generalizar a utilização desses componentes plugáveis. |
| *Aplicabilidade:* | Entre outros, quando não se conhece o domínio, nem a especificidade do domínio do componente que irá conectar. |
| *Participantes:* | Define uma maneira centralizada de acesso do componente principal ao componente plugável. |
| *Consequências:* | 1. Acoplamento abstrato entre componentes principal e plugável.<br>2. Baixo acoplamento e facilidade de conexão a novos módulos. |
| *Utilização no protocolo:* | É utilizado na geração de números aleatórios, onde, uma ve configurado qual o módulo de geração de aleatoriedade, a classe que implementa o método desejado é acoplada ao componente principal de geração de números aleatórios, onde esse conhece somente o contrato de identidade do módulo conectado. |

- ***Object Pool (não GoF)***

Como padrão de criação, o *Object Pool* (não GoF), propõe a solução para o seguinte cenário, definido na tabela 3.4.

**Tabela 3.4: Definição do padrão de projeto *Object Pool***

| | |
|---|---|
| *Inteção:* | Possibilitar o reaproveitamento de objetos. |
| *Motivação:* | Permitir adquirir recursos a medida em que esses são solicitados, até o limite da disponibilidade, e liberá-los quando o término de sua utilização acontecer. |

| | |
|---|---|
| *Aplicabilidade:* | Entre outros, para manter a uniformidade do acesso a recursos limitados. |
| *Participantes:* | Ao ser solicitado um evento, o gerenciador de recursos manipula a solicitação e é notificado em sua finalização. |
| *Consequências:* | 1. Reaproveitamento de recursos limitados.<br>2. Rastreabilidade de utilização de recursos. |
| *Utilização no protocolo:* | É utilizado nos processos lógicos, no momento da solicitação do centro de serviço por um evento de chegada. O processo lógico possui um número limitado de centros de serviço, e sua utilização deve ser rastreada de forma eficaz. |

- ***Mediator***

Como padrão comportamental, o *Mediator* propõe a solução para o seguinte cenário, definido na tabela 3.5.

**Tabela 3.5: Definição do padrão de projeto *Mediator***

| | |
|---|---|
| *Intenção:* | Diminuir a quantidade de ligações independentes entre objetos, centralizando e uniformizando a comunicação dentro de um domínio controlado. |
| *Motivação:* | Permite rastrear e controlar a comunicação dentro de um sistema controlado. |
| *Aplicabilidade:* | Entre outros, para manter a uniformidade na comunicação entre recursos independentes dentro de um sistema. |
| *Participantes:* | Ao realizar a comunicação/envio de mensagem de um objeto a outro objeto, a solicitação deverá obrigatoriamente atravessar um controlador/mediador, que consiga auditar a transação. |
| *Consequências:* | 1. Controle de comunicação em um domínio.<br>2. Centralização de acessos relacionados a comunicação. |
| *Utilização no protocolo:* | É utilizado nos processos lógicos, no momento do encadeamento de mensagens entre eles. Para evitar que os processos lógicos comuniquem-se diretamente entre si, há um centralizador e atravessador em *DarfiaSimulation* capaz de endereçar os fluxos de |

| | |
|---|---|
| | mensagens entre processos lógicos, centralizando assim, a comunicação. |

Além do emprego dos padrões descritos, também foi utilizado na construção do sistema o padrão de dados *eXtensible Markup Language* (XML), para criação da configuração do Darfia. É objetivo também do trabalho oferecer ao usuário final uma grande flexibilidade para configuração da simulação. Para tal, escolheu-se um formato de dados que pudesser ser legível tanto por máquina quanto por pessoas, caracterizando a configuração do sistema. A sua utilização dá-se no momento do ingresso de um processo lógico no ambiente de simulação, informando à plataforma o caminho do documento.

Maiores detalhes sobre o emprego do padrão XML, bem como a construção da configuração da simulação, estão detalhadas na seção 3.3.

## 3.3 Configuração da simulação

A configuração da simulação reflete os objetos que serão instanciados em memória e, portanto, inseridos na simulação. Para tal, através do documento de configuração, especificado em XML, é possível configurar as seguintes características da simulação:

- Nome, tempo de finalização da simulação e nível de *trace* da execução, facilitando a depuração da execução.
- Número de processos lógicos que comporão o ambiente de simulação e, portanto, compartilharão memória.
- Vínculo entre processos lógico e processos físicos, para definir a responsabilidade de qual desses objetos será processado.
- Configuração de característica de produção artificial de eventos, além da configuração do perfil de probabilidade referente ao tempo de chegada, e ao tempo de serviço, a partir dos perfis disponíveis (Erlang, Exponencial, Fixo, Normal, Uniforme e Aleatório) no sistema. Permite especificar também o número de eventos iniciais no sistema, bem como um limite na produção, para que não sejam produzidos indefinidamente. Além disso, também é possível especificar o número de operações de ponto flutuante executados em eventos de chegada e partida, do centro de serviço. A adição da característica de cálculos de números

de ponto flutuante é devido a característica fictícia de carga útil de processamento de eventos, permitindo a configuração de um simulação onde cada evento processado consome tempo de processamento do processo lógico.

- Configuração da característica de encadeamento de eventos entre processos lógicos, além da configuração das probabilidade de continuidade no sistema ou saída do sistema, bem como o direcionamento do fluxo de encadeamento. Também é possível configurar o perfil de distribuição de probabilidade que definirá o tempo de serviço ocupado no processo lógico destino, após o evento ser encadeado. Essa característica permite simular situações onde, no encadeamento de eventos, um evento possui tempos de serviço diferentes em relação ao processo lógico.

Na figura 3.1 é apresentado um exemplo de configuração de simulação de somente um processo lógico, capaz de produzir chegada de clientes. Nesse exemplo, a simulação chamada de "aSimpleSimulation" tem tempo de finalização em "200000", e não exibe mensagens de depuração. Essa simulação é composta por somente um processo lógico, "CPU1", executada pelo processo físico "0", do sistema operacional. Esse processo lógico possui a característica de produtor de chegadas, com perfil de distribuição de probabilidade "Exponencial" com média em "3" no tempo de chegada de eventos, e desvio "0", e média do tempo que esse evento ocupará no centro de serviço desse processo lógico em perfil de distribuição de probabilidade de "Erlang", em média em "2" e desvio em "0". Além disso, produzirá somente 1 evento em fase de inicialização da simulação, e ao longo da computação do sistema, irá popular o sistema com até 5 eventos de chegada/saída.

```
<?xml version="1.0" encoding="UTF-8"?>
<simulation name="aSimpleSimulation" endtime="200000"
    tracelevel="0">

    <logicalprocesses>

        <logicalprocess id="CPU1" phisicalprocessid="0">

            <arrivalproducer initialnumber="1"
                limitnumber="5" meanarrivaltime="exponential:3,0"
                meanservicetime="erlang:2,0" />

        </logicalprocess>

    </logicalprocesses>

</simulation>
```

Figura 3.2 – Exemplo de configuração da simulação

Ainda no mesmo exemplo, o nó <*simulation*> representa a simulação em si, onde os parâmetros de “name” e “endtime” são obrigatórios, especificando respectivamente o nome da simulação e o tempo virtual de término. O atributo “tracelevel” indica o nível de depuração da simuação. Caso esse atributo não seja especificado, assume-se o valor 0.

Os nós <*logicalprocess*> obrigatoriamente deverão ser filhos do nó <*logicalprocesses*>, e representam os processos lógicos que serão instanciados no sistema de simulação. O atributo “*id*” e “phisicalprocessid” são obrigatórios, representando, respectivamente, o identificador do processo lógico, e o identificador do processo físico do sistema operacional que irá assumir o processamento desse processo lógico.

O nó <*arrivalproducer*> não é obrigatório, mas uma vez definido dentro do nó <*logicalprocess*>, adiciona a esse, a característica de produção de eventos de chegada e saída de clientes. Para isso, os atributos obrigatórios “*meanservicetime*” e “*meanarrivaltime*” orientam o protocolo a configurar o perfil de probabilidade (bem como seus parâmetros) para geração de números aleatórios. O valor desses atributos definem o perfil de probabilidade e os parâmetros desse. No caso do exemplo da figura, o atributo “*meanservicetime*”, que representa a média de tempo de serviço ocupado pelo cliente (ou seja, o quanto de serviço que o cliente irá gastar do prestador de serviço, entre o seu momento de entrada em serviço até sua saída deste) é definido por “erlang:1.5,0.1”, onde “erlang” define o perfil de probabilidade Erlang adotado para a geração de números aleatórios dessa medidade, onde 1.5 é a média e 0.1 o desvio. Os atributos opcionais “*initialnumber*” e “*limitnumber*” representam, respectivamente, o total de eventos que será inserido na fila de eventos futuros em tempo de inicialização do sistema e o número total de chegadas artificiais que serão criadas no processo lógico.

Continuando no exemplo dado, o atributo “*meanarrivaltime*”, indica a média de tempo entre eventos de chegada de clientes, e representa o perfil de distribuição de probabilidade utilizado para a geração de eventos de chegadas de cliente no processo lógico. No caso do exemplo, foi especificado o mesmo perfil, com parâmetros diferentes, 2 para a média e 0.1 para o desvio.

Já na figura 3.2 apresenta-se um exemplo de configuração onde há o encadeamento de eventos através dos processos lógicos do sistema de simulação.

```
<?xml version="1.0" encoding="UTF-8"?>
<simulation name="aSimpleSimulation" endtime="200000"
    tracelevel="0">

    <logicalprocesses>

        <logicalprocess id="CPU1" phisicalprocessid="0">

            <arrivalproducer initialnumber="1"
                limitnumber="5" meanarrivaltime="exponential:3,0"
                meanservicetime="fixed:2" />

            <chaining quitprob="0.2" contprob="0.8" targets="DISK1,DISK2"
                meanremoteservicetime="fixed:9" />

        </logicalprocess>

        <logicalprocess id="DISK1" phisicalprocessid="1" />

        <logicalprocess id="DISK2" phisicalprocessid="1" />

    </logicalprocesses>

</simulation>
```

Figura 3.3 – Exemplo de configuração da simulação com encadeamento

No caso ilustrado pela figura 3.2, o nó <*chaining*>, não obrigatório, representa a adição da característica de encadeamento de mensagens emitidas pelo processo lógico que o nó está. Através da existência desse nó, o processo lógico adquire a caraterística da emissão de mensagens para todos os processos lógicos especificados no atributo "*targets*". O valor desse campo, separado por vírgula, representa o identificador dos processos lógicos que receberão as mensagens enviadas. O atributo "*quitprob*" representa a probabilidade linear da mensagem sair do sistema, o atributo "*contprob*" representa a probabilidade do evento ser encadeado para os alvos e o atributo "*meanremoteservicetime*" representa o perfil da distribuição de probabilidade do tempo de serviço que o evento encadeado terá no processo lógico remoto.

A inicialização e execução da plataforma, obriga o usuário a informar o documento de configuração utilizado para a simulação. Uma vez que simulação é iniciada, o documento é mapeado em memória utilizando as classes especificadas em módulos da implementação.

## 3.4 Módulo de Gerenciamento

O pacote *Managers* é responsável por armazenar as principais classes do sistema, responsáveis por gerenciar todo o ciclo de execução da ferramenta, bem como servir como um gestor de troca de mensagens, além de possuir controles sobre o término da simulação. Na figura 3.3 é detalhada, em diagrama de classes, a estrutura do pacote.

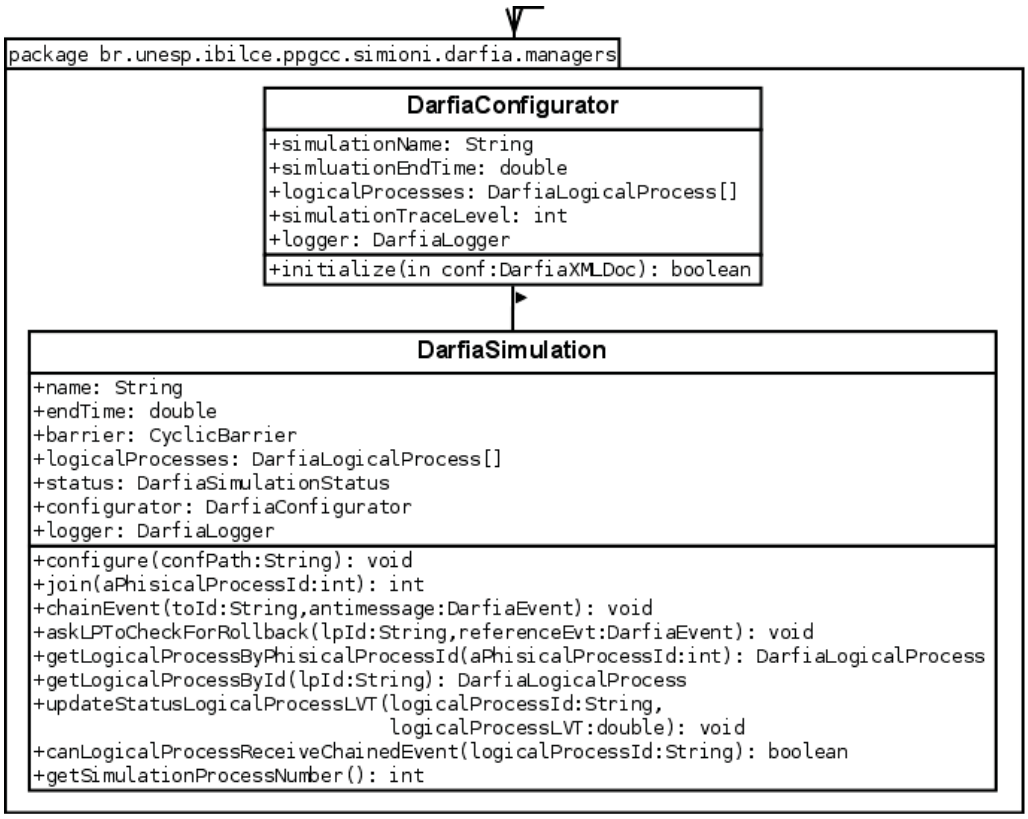


Figura 3.4 – Diagrama de classes – Pacote *Managers*

## DarfiaConfigurator

Classe responsável por interpretar um documento formal XML, que deve ser passado como parâmetro para a simulação, configurando-a, antes de iniciar sua execução. Essa classe recupera e valida todas as informações adicionadas no arquivo de configuração, sendo elas (anotadas no padrão XPath):

- Nó /simulation, possuindo os atributos "*name*", "*endtime*" e "*tracelevel*", representando respectivamente, o nome da simulação e o tempo de término e o nível de depuração de execução desta. A depuração é facilitada através de mensagens de notificações, enriquecendo a marcação de evidências da execução.
- Nós /simulation/logicalprocesses/logicalprocess, possuindo os atributos "*id*" e "*phisicalprocessid*", representando respectivamente, o identificador do processo lógico, bem como a identificação que irá atrelar o processo físico do sistema computacional, com o processo lógico da simulação. O parâmetro de identificação do processo físico é passado por parâmetro na linha de comando que ingressa o membro da simulação no sistema.

- Nó /simulation/logicalprocesses/logicalprocess[N]/arrivalproducer, devendo possuir os atributos "*meanservicetime*" e "*meanarrivaltime*". Esse nó não é obrigatório, entretanto, caso ele exista, indicará que o processo lógico N possuirá característica de produtor de eventos de chegada e saída de clients nele mesmo. Ou seja, poderá além de receber eventos de outros processos lógico, produzir os seus próprios para ele mesmo. O atributo "*meanservicetime*" indica a média de tempo que o cliente gerado por esse processo lógico terá de tempo de serviço, ou seja, ocupará o prestador de serviço. O atributo "*meanarrivaltime*" indica a média de tempo que um novo evento chegará a esse processo lógico (produzido por ele mesmo). Para especificar a média de tempos, é utilizada uma notação própria, que indica o perfil da distribuição aleatória, bem como os parâmetros para essa distribuição. Por exemplo, o valor de um desses atributos poderá ser "*erlang:1.5,0.5*", indiando que a distribuição aplicada será de Erlang, com média em 1.5 e desvio de 0.5. Maiores detalhes sobre como escolher o perfil da distribuição estarão no pacote **Random**. Os atributos não obrigatórios "*operationsnumberinarrival* " e "*operationsnumberindeparture* ", representam respectivamente, o número de operações de ponto flutuante executadas no processamento de evento de chegada e nos eventos de saída do cliente do centro de serviço do processo lógico. Esses atributos podem ser utilizados para conferir a capacidade de carga útil no processamento de eventos. Os atributos não obrigatórios "*limitnumber*" e "*initialnumber*" definem, respectivamente, um número limite de chegads de eventos, bem como um número inicial de eventos a ser processados pelo processo lógico, na fase de inicialização da simulação.
- Nó /simulation/logicalprocesses/logicalprocess[N]/chaining, devendo possuir os atributos "*quitprob*", "*contprob*", *"targets".* Esse nó não é obrigatório no processo lógico N, porém se existir, adicionará ao processo lógico a capacidade de encadear eventos, fazendo com que os eventos que sairem do processo lógico N, passem por uma avaliação sobre o que devem fazer, a partir dos parâmetros mencionados acima, sendo respectivamente: "*quitprob*": probabilidade [0..1] do evento sair do sistema; "*contprob*": probabilidade *1-quitprob* do evento continuar no sistema e ser enviado para outro processo; "*targets*": indica quais os *id* dos processos lógicos que receberão os eventos encadeados, inclusive podendo ser ele mesmo. Lembrando que "*quitprob*" somado a "*contprob*" deve obrigatoriamente ser 1. Ao

final do processamento de um evento no processo lógico, um número aleatório (distribuição normal) é escohido para definir se o evento sai do sistema ou é encadeado.

### DarfiaSimulation

Classe responsável por organizar a simulação e representar o nó raíz compartilhado da arquitetura Terracotta. A arquitetura do *framework* utilizado define que uma vez que um objeto é compartilhado na raiz, todos os seus objetos descendentes também serão alcançáveis e farão parte da árvore compartilhada em memória, com suas instâncias instrumentadas via *bytecode*, para ser distribuída nos servidores da rede.

Basicamente possui três principais propriedades:

- A barreira capaz de alinhar e sincronizar os processos do ambiente de simulação. Essa propriedade é do tipo *java.util.concurrent.CyclicBarrier*, e possui em seu construtor o número total de processos que ingressarão no ambiente. Quando é necessário sincronizar, todos os processos deverão chamar a função da barreira **wait()**, ficando bloqueados até que o Nesimo processo atinja a linha de código.
- Uma estrutura de vetor para ter acesso aos apontamentos de todos os processos lógicos da simulação. É através desse vetor que será possível solicitar a simulação a troca de mensagens, a sincronização e o procedimento de *rollback*.
- O apontamento para o objeto de configuração, comentado acima, para ter acesso ao final da simulação, bem como os processos lógicos e suas características.
- O apontamento para o objeto de *status* do sistema, do tipo *DarfiaSimulationStatus*. Esse objeto será responsável por guardar informações dos processos já ingressantes no sistema, bem como os dados da simulação.

Além de suas principais propriedades também é responsável por iniciar a configuração da simulação, realizar a inicialização de eventos nos processos lógicos, bem como coordenar a ação distribuída de execução dos processos, agindo como um mediador de chamadas entre processos.

Também é responsável por detectar o fim da simulação nos processos lógicos, bem como solicitar ao processos lógicos a computação das estatísticas de cada um, para então agrupar tais dados.

## 3.5 Módulo de Componentes

O pacote *Components* é o grande responsável pela composição da simulação em si, contendo divesas classes necessárias para o funcionamento dos processos lógicos, bem como definições de troca de eventos, encadeamentos, produção de clientes, anotações de estatísticas individuais, e de fato, notificações de *rollbacks*. A figura 3.4a e 3.4b ilustram o diagrama de classes do módulo.

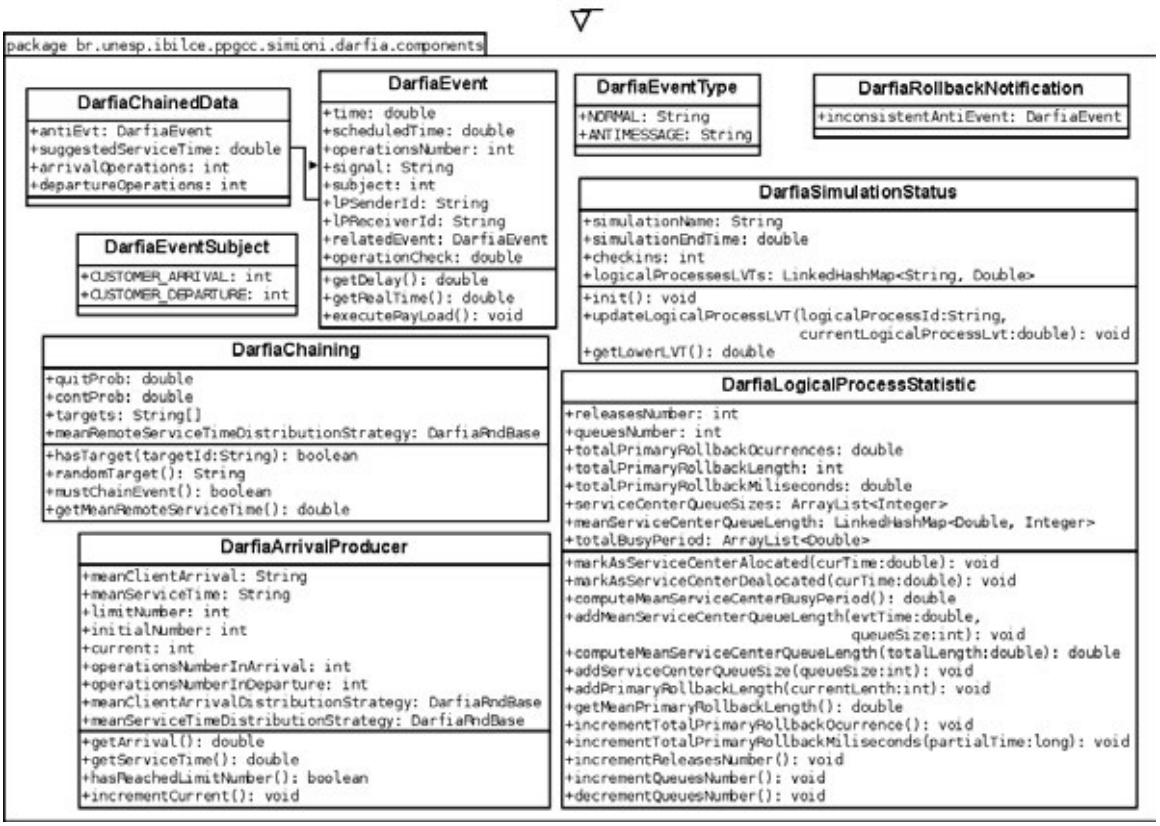


Figura 3.5a – Diagrama de classes – Pacote *Components*


Figura 3.5b – Diagrama de classes – Pacote *Components*

As classes que compõem o pacote estão listadas a seguir.

## DarfiaArrivalProducer

Representa a capacidade de produção (chegadas artificiais) de eventos em um processo lógico. Possui as características da produção e duração e carga útil de cada evento, em relação ao prestador de serviço. É nessa entidade que serão processadas as informações de distribuição de probabilidade de chegadas e tempo de serviço dos eventos do sistema, configuradas em

*DarfiaConfiguration*. O processo de *parsing* dos atributos especificados na configuração varia de distribuição para distribuição, adaptando-se à entrada de parâmetros como média, intervalos, e desvios.

### DarfiaChainedData

Representa a notificação de encadeamento de eventos entre um processo lógico e outro. Devido a situações de concorrência, a troca de mensagens via encademento entre processos lógicos, é implementada através de filas, caracterizando *buffers* de entradas. O conteúdo da fila de eventos encadeados é composta por essa classe. Nela é possível armazenar a antimensagem, bem como o tempo de consumo do centro de serviço do processo lógico remoto.

### DarfiaChaining

Representa a capacidade de encadeamento de mensagens em processos lógicos, configurados também em *DarfiaConfiguration*. Responsável por armazenar as informações de probabilidade de saída e encadeamento dentro do sistema, além da geração da aleatoriedade na decisão de processos lógicos destino, no momento de um encadeamento.

### DarfiaEvent

Representa um evento (ou também chamada de mensagem) dentro do sistema. Possui atributos básicos característico de um evento dentro do processo de simulação: seu tempo de inserção no sistema (por se tratar de troca de mensagens em memória compartilhada, não há atrasos da camada de transporte), seu tempo de processamento (utilizado para medir o atraso do prestado de serviço), sua identificação enquanto propósito (entrada, saída, normal, antimensagem), identificações de processo lógico emissor e receptor, e um apontamento para outro evento relacionado e o número de operações de ponto flutuante que realizará no processo lógico, garantindo a existência de carga útil.

### DarfiaEventSubject

Classe de enumeração de todos os possíveis assuntos que um evento pode ter. Os implementados são somente eventos de chegada e partida.

### DarfiaEventType

Classe de enumeração de todos os possíveis tipos que um evento pode ter. Os implementados são somente eventos normais e antimensagens.

### DarfiaLogicalProcess

Responsável por representar um processo lógico dentro da simulação. É obrigatoriamente executado por um processo físico do sistema operacional que opera do sistema de simulação. Possui diversas características da simulação atual, como a representação do prestador de serviço, o LVT, informações sobre enfileiramento de *rollbacks*, enfileiramento de encadeamentos de eventos de outros processos lógicos, informações sobre geração de eventos, filas de entrada (LEF) e de saída (antimensagens), fila de eventos processados e eventos aguardando a disponibilidade do prestador de serviço. Além disso, também possui informações estatísticas do processamento, para auxiliar nos diversos relatórios, como tamanho médio da fila, tempo de uso do prestador de serviço, etc.

### DarfiaLogicalStatistic

Responsável por armazenar informações históricas sobre a execução do processo lógico, a fim de ser utilizado na coleta de dados estatísticos ao final do processamento, medindo assim, os resultados dos parâmetros informados na configuração da simulação.

### DarfiaRollbackNotification

Responsável por armazenar temporariamente as informações de uma notificação de *rollback* do sistema, como qual o evento causador do *rollback*, e a lista de eventos relacionados, identificados como inválidos (possuindo um LVT maior que o LVT da mensagem causadora).

### DarfiaSimulationStatus

Responsável por guardar as informações da simulação, durante o ciclo de vida desta. Guarda informações sobre o *checkins* de processos físicos, bem como propriedades compartilhadas da simulação, como o relógio global.

## 3.6 Módulo de Filas

O pacote *Queues* é responsável por abrigar as abstrações de filas do sistema, além de abstrair

o funcionamento comum entre elas, permitindo que o protocolo seja adaptado através de sua extensão. Na figura 3.5 é ilustrado o diagrama de classes do módulo.

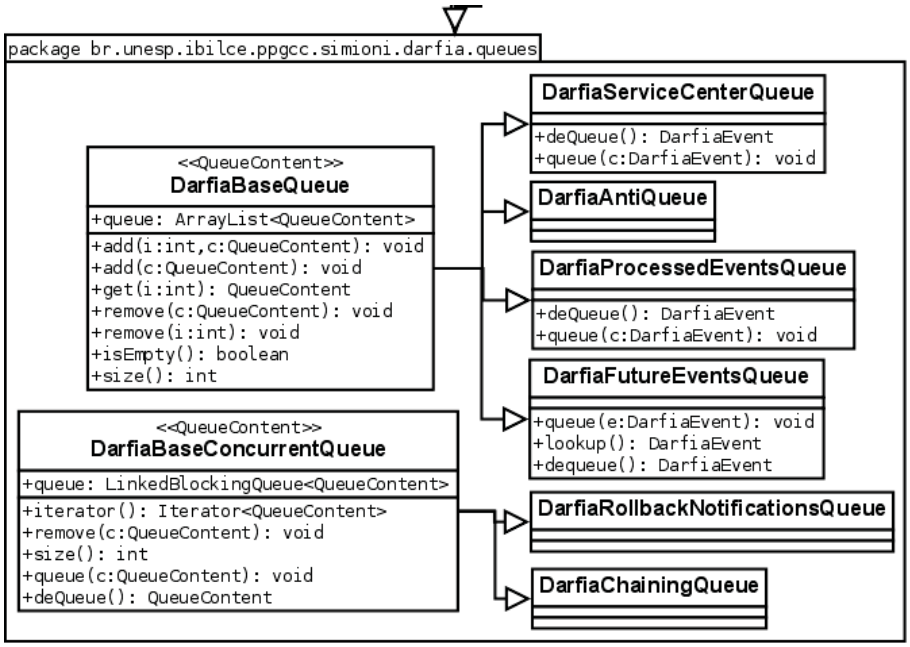


Figura 3.6 – Diagrama de classes – Pacote *Queues*

As classes que o compõem estão listadas a seguir.

### DarfiaBaseQueue

Responsável por formar a base de uma fila comum, implementada utilizando listas dinâmicas e duplamente ligadas, compartilhando acesso a métodos já implementados. As filas do sistema obrigatoriamente estendem essa classe, herdando suas características, facilitando o trabalho da manutenção de código.

### DarfiaAntiQueue

Responsável por armazenar as cópias de mensagens (antimensagens) enviadas para outros processos. É utilizada para validação/verificação de encadeamento inválido de mensagens, tratáveis pelos procedimentos de *rollback*. Herda *DarfiaBaseQueue*.

### DarfiaBaseConcurrentQueue

Responsável por formar a base de uma fila concorrente, utilizada para realizar o

enfileiramento de mensagens trocadas pelos processos lógicos. Sua implementação utiliza o tipo *java.util.concurrent.LinkedBlockingQueue*, deixando para a *JVM* a responsabilidade de gerenciar o acesso exclusivo a ela. As filas do sistema que são utilizadas para enfileiramento de mensagens entre processos lógicos, obrigatoriamente estendem essa classe, herdando suas características, facilitando o trabalho da manutenção de código.

### DarfiaChainingQueue

Responsável por armazenar em formato de *buffer* a chegada de mensagens de encadeamentos nos processos lógicos. Herda *DarfiaBaseConcurrentQueue*.

### DarfiaFutureEventsQueue

Responsável por organizar a chegada de eventos no processo lógico, a fim de obedecer o enfileiramento de eventos em ordem crescente de tempo. É utilizada também como estrutura de controle para detecção de *rollback*. Herda *DarfiaBaseQueue*.

### DafiaProcessedEventsQueue

Representa a fila de eventos processados no sistema. É utilizada para armazenar os evento que foram processados pelo prestador de serviço alocado no processo lógico. Herda *DarfiaBaseQueue*.

### DafiaRollbackNotificationsQueue

Responsável por armazenar em formato de *buffer* a chegada de mensagens de *rollbacks* encadeados nos processos lógicos. Herda *DarfiaBaseConcurrentQueue*.

### DafiaServiceCenterQueue

Representa a fila de eventos do centro de serviço nos processos lógicos. É utilizada para armazenar os eventos que estão em fila de espera para o prestador de serviço do processo lógico. Herda *DarfiaBaseQueue*.

## 3.7 Módulo de geração de números aleatórios

O pacote *Random* é responsável por fornecer diversas classes responsáveis pela geração de

números aleatórios[MAC87]. Oferece uma classe base para geração de números aleatórios uniformemente distribuídos (*DarfiaRndBase*), que é estendida por qualquer distribuição específica de distribuição, que utilize o mesmo método de geração de sementes. Uma vez estendida, obrigatoriamente deverá implementar um método *generate()*, que deverá retornar o valor da distribuição.

Caso o protocolo seja estendido por alguma aplicação cliente, é possível implementar qualquer perfil de probabilidade, sendo possível acoplá-lo no sistema de simulação, através da implementação da interface geradora. Na figura 3.6 é ilustado o diagrama de classes do pacote.

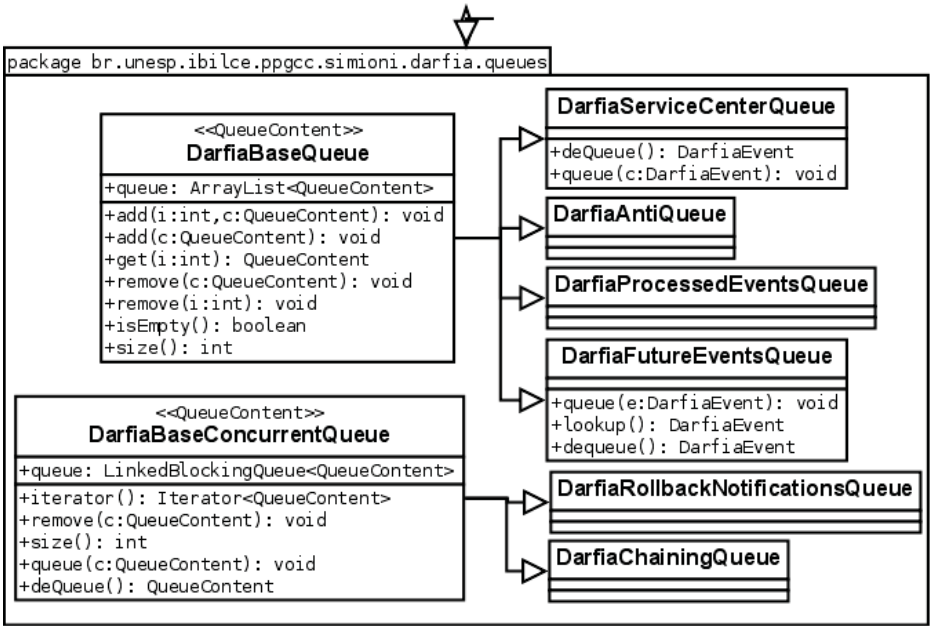


Figura 3.7 – Diagrama de classes – Pacote *Random*

No trabalho presente, as seguintes distribuições foram implementadas, e estão à disposição dos usuários:

### DarfiaRndBase

Possui as sementes de geração de números aleatórios, bem como a geração base de números aleatórios uniformemente distribuídos no intevalo de 0..1.

### DarfiaRndErlang

Retorna uma variável aleatória de distribuição Erlang, com parâmetros de média e desvio. Herda *DarfiaRndBase*.

### DarfiaRndExponential

Retorna uma variável aleatória de distribuição exponencial negativa, com parâmetro de média. Herda *DarfiaRndBase*.

### DarfiaRndFixed

Retorna uma constante, com parâmetro de valor. Herda *DarfiaRndBase*.

### DarfiaRndNormal

Retorna uma variável aleatória de uma distribuição normal, com parâmetros de média e desvio. Herda *DarfiaRndBase*.

### DarfiaRndRandom

Retorna uma variável de uma distribuição aleatória, com parâmetros de intervalo de início e fim. Herda *DarfiaRndBase*.

### DarfiaRndUniform

Retorna uma variável de uma distribuição uniforme, com parâmetros de intervalo de início e fim. Herda *DarfiaRndBase*.

## 3.8 Módulo de Utilitários

O pacote **Utils** tem como responsabilidade guardar os utilitários utilizados pelo sistema em si. Basicamente é formado por um encapsulamento da API DOM Java, com métodos facilitadores para interpretar o documento de configuração da plataforma, além de um pequena biblioteca de erros, para informar o motivo da finalização da execução. Possui também um centralizador de geração de eventos e tratativas de perfis de geração de números aleatórios. Na figura 3.7 é ilustrado o diagrama de classes do pacote.

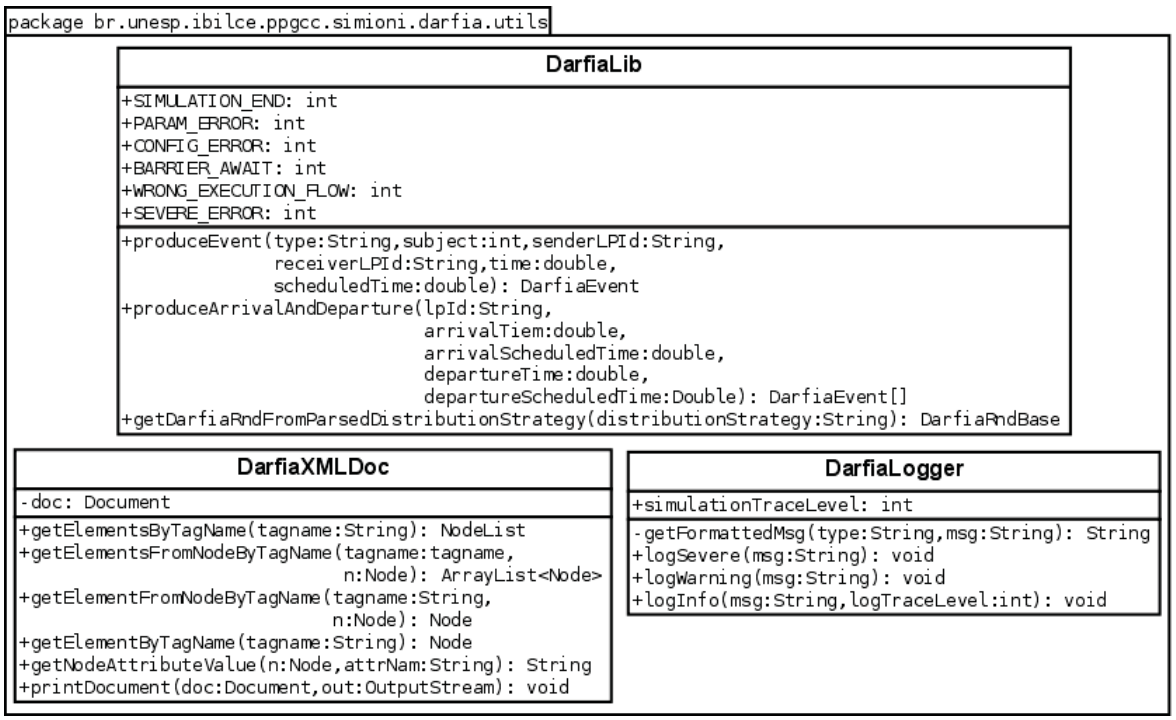


Figura 3.8 – Diagrama de classes – Pacote *Utils*

As classes que compõe o pacote estão listadas a seguir.

### DarfiaLib

Responsável por agir como uma biblioteca de dados utilizados para expor causas e razões de falhas ou ações do sistema. Sua principal função é prover uma biblioteca consistente de erros, para que o usuário/software que solicitar a execução do sistema consiga tratar a saída de erros. Além disso, também exerce o papel de auxiliar na criação de eventos genéricos dentro do processo de simulação, bem como na centralização de *parsing* das distribuições exponenciais especificadas no documento de configuração.

### DarfiaLogger

Responsável por centralizar todas as ações de evidências do sistema, possuindo métodos para evidenciar diversas situações, como notificações, erros ou avisos.

### DarfiaXMLDoc

Responsável por encapsular a API DOM do Java, especificada pela W3C (org.w3c.dom), facilitando o trabalho de *parsing* e recuperação de informações do arquivo de configuração da

simulação.

## 3.9 Implementação e utilização

A plataforma foi projetada para suportar diversos processos físicos comunicando-se entre si, através da utilização das classes e estruturas declaradas acima, onde todos eles executam sobre o mesmo espaço de endereçamento de memória, provido pela API do Terracotta.

Entretanto, para que o processo funcionasse de forma adequada e controlada, algumas restrições e premissas foram definidas:

- Cada processo físico de máquina (estrutura do sistema operacional gerenciada pelo escalonador de processo do núcleo do sistema operacional, capaz de alcançar a área de memória disponível pela API Terracotta) poderá gerenciar somente um processo lógico do sistema (estrutura do sistema proposto pelo trabalho – *DarfiaLogicalProcess*). Tais processos lógicos são objetos em memória que utilizam uma faixa declarada de memória compartilhada e distribuída, gerenciada pelo Terracotta). Essa é uma definição estática, realizada através do documento de configuração da plataforma.
- Cada processo lógico do sistema (*DarfiaLogicalProcess*) será capaz de possuir somente um prestador de serviço. Este, responsável por gerenciar o tempo de serviço ocupado pelos clientes/eventos do sistema, poderá atender somente uma chegada de cliente em um dado momento de tempo. Devido a isso, cada processo lógico do sistema conterá somente uma fila de chegada, capaz de organizar em disciplina FIFO, gerenciando entradas, saídas e atrasos de eventos no sistema.
- Cada processo lógico poderá enviar somente uma mensagem para um outro processo lógico do sistema, por ciclo de consumo de LVT. A cada vez que a LEF for consumida, e o LVT for atualizado, o processo lógico decidirá se, quando ocorrer uma mensagem de saída, esta deve ser direcionada a outro processo lógico, caso o primeiro tenha capacidade de encadear mensagens. Após a transmissão desta, outras mensagem poderão ser enviadas a partir dos próximos eventos processados.
- Cada processo lógico dentro do sistema poderá receber eventos gerados por si mesmo(a partir da configuração especifcada via *DarfiaConfiguration*) ou através de outros processos lógicos, em um processo de encadeamento (também, a partir da configuração). Caso um

processo lógico do sistema não possa receber eventos encadeados e não possa gerar eventos para si mesmo, este não terá carga útil de processamento, e aguardará a finalização dos outros processos lógicos do ambiente de simulação.

- Caso aconteça uma situação de *rollback*, o procedimento padrão será o cancelamento agressivo do processo lógico, retrocedendo imediatamente todos os eventos considerados inválidos a partir da mensagem *straggler*.

O diagrama de estados ilustrado pela figura 3.8 demonstra de maneira geral, o funcionamento básico do sistema, bem como a comunicação entre processos, descrevendo as estruturas anunciadas anteriormente, dentro do ambiente de simulação do *Darfia*.

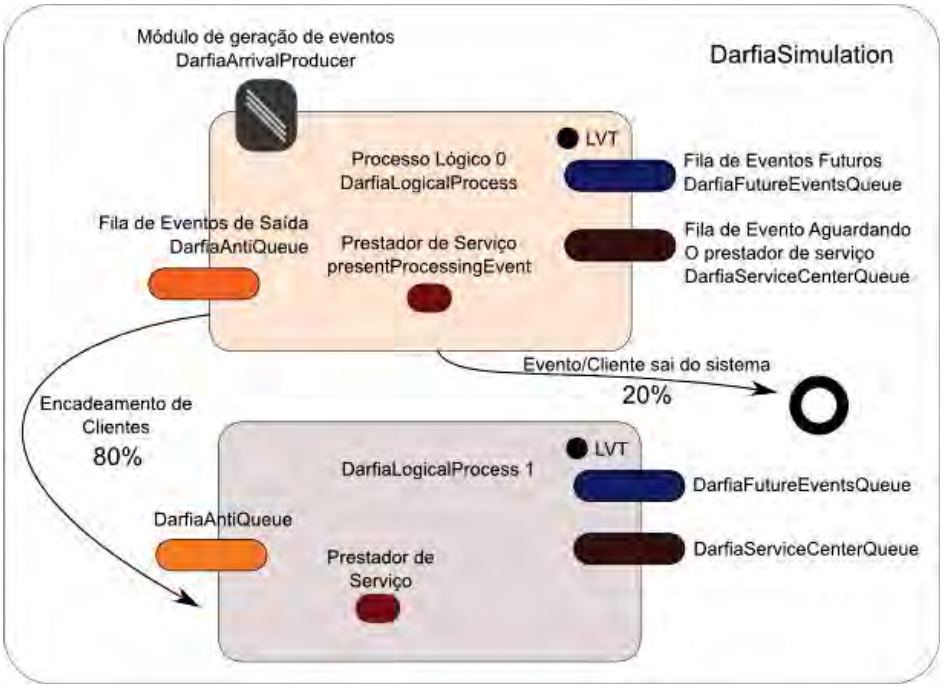


Figura 3.9 – Diagrama de estados da simulação

O diagrama de atividades ilustrado pela figura 3.9, demonstra o funcionamento básico do fluxo de instruções do sistema, baseado nas premissas da simulação computacional paralela otimista.

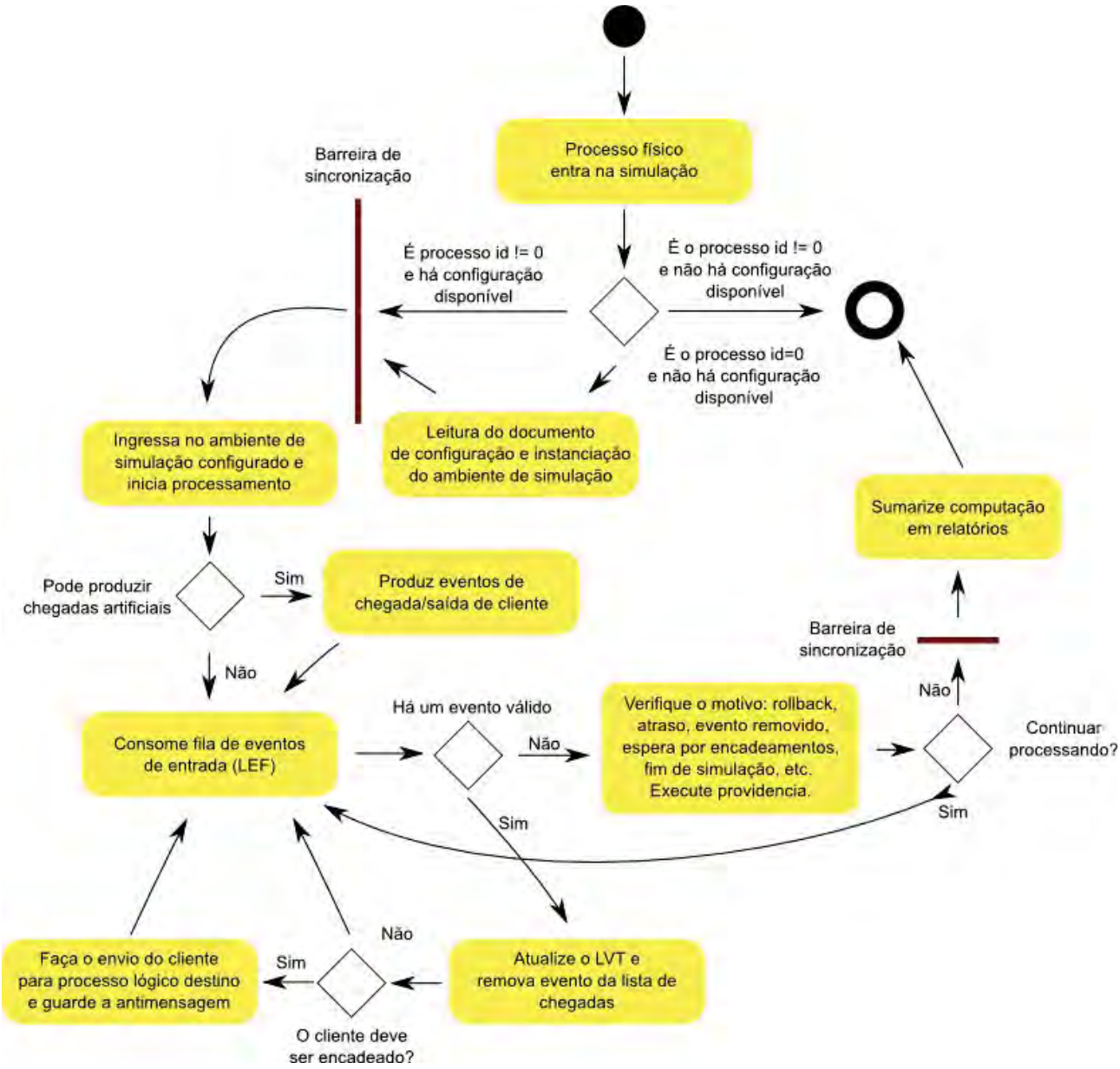


Figura 3.10 – Diagrama de atividades da simulação

Dado o funcionamento geral da plataforma, bem como suas construções internas, nota-se uma grande semelhança em relação ao desenvolvimento de software voltado ao ambiente de memória compartilhada simples. Esse é um grande recurso provido pela API do Terracotta, que livra o desenvolvimento do código, da preocupação da comunicação entre os elementos que consumirão o espaço de endereçamento de memória provido pela API.

A camada de abstração e instrumentação de *bytecodes* provida pela API do Terracotta fica transparente ao desenvolvimento do software deste trabalho. O único trabalho necessário é a configuração incial da API, feita através de um documento XML simples.

Nessa configuração anota-se dados do funcionamento interno da API, como por exemplo, onde ficarão os caminhos para os *logs* da API, quais métodos/objetos serão compartilhados/instrumentados no grafo de objetos compartilhados, qual o caminho do servidor de memória DSM, bem como onde haverá as travas sincronizadas para acesso simultâneo em memória compartilhada.

Em relação a travas, há um ponto importante e crucial para o desenvolvimento do projeto, no que tange o acesso de leitura/escrita a valores dos campos dos objetos compartilhados. Pelo fato do Java prover um modelo forte e conciso de componentes, através dos chamados *Java Beans*, implementados através dos métodos *Getter* e *Setters*, o uso da API do Terracotta só é seguro caso o acesso aos campos dos objetos compartilhados seja feito através de tais métodos, e não diretamente ao campo, em operações de leitura e escrita.

Essa determinação acontece por consequência do funcionamento da API. Como esta é responsável por reimplementar todos os métodos dos objetos definidos como compartilhados (através de reflexão Java), na fase de *bootstrap* da aplicação, viabilizando os *locks* distribuídos , obrigatoriamente para leitura e escrita, os campos devem ser acessados via métodos do tipo *get* e *set*, para que estes também sejam reimplementados, e portanto, seus *bytecodes* resultantes sejam instrumentados pela API de maneira correta, no sentido do endereçamento do dado na memória compartilhada distribuída, mantendo o dado conciso e atualizado.

Caso essa determinação não seja seguida, o servidor de compartilhamento de objetos do Terracotta enviará aviso de acesso indevido a campos compartilhados ao usuário, evitando assim, situações de corrida ou situações de *deadlock* na estratégia produtor/consumidor. Embora haja uma determinação estrita sob essa ótica, através do documento de configuração da API é possível sobrescrever algumas configurações, realizando ajustes de próposito de desempenho.

Na figura 3.10 é ilustado um exemplo de configuração da API Terracotta. Essa configuração deve ser informada ao iniciar o servidor de DSM, bem como ao iniciar um participante da simulação.

```
<?xml version="1.0" encoding="UTF-8"?>
<tc:tc-config xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-6.xsd"
xmlns:tc="http://www.terracotta.org/config" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <tc-properties/>
  <servers>
    <server host="%i" name="localhost">
      <dso-port bind="0.0.0.0">9510</dso-port>
      <jmx-port bind="0.0.0.0">9520</jmx-port>
      <data>terracotta/server-data</data>
      <logs>terracotta/server-logs</logs>
      <statistics>terracotta/cluster-statistics</statistics>
    </server>
  </servers>
  <clients>
    <logs>terracotta/client-logs</logs>
  </clients>
  <application>
    <dso>
      <instrumented-classes>
        <include><class-expression>br.unesp.ibilce.ppgcc.simioni.darfia.componerts.*</class-expression></include>
        <include><class-expression>br.unesp.ibilce.ppgcc.simioni.darfia.managers.*</class-expression></include>
        <include><class-expression>br.unesp.ibilce.ppgcc.simioni.darfia.queues.*</class-expression></include>
        <include><class-expression>br.unesp.ibilce.ppgcc.simioni.darfia.random.*</class-expression></include>
        <include><class-expression>br.unesp.ibilce.ppgcc.simioni.darfia.utils.*</class-expression></include>
      </instrumented-classes>
      <roots>
        <root><field-name>br.unesp.ibilce.ppgcc.simioni.darfia.Darfia.simulation</field-name></root>
      </roots>
      <locks>
        <autolock auto-synchronized="true">
          <method-expression>void *..*(..)</method-expression>
          <lock-level>write</lock-level>
        </autolock>
      </locks>
    </dso>
  </application>
</tc:tc-config>
```

Figura 3.11 – Configuração da API Terracotta

No exemplo ilustrado na figura acima, há configurações tanto para o servidor da API, quanto para os clientes que ingressarão no ambiente de DSM. É nesse documento XML também, que há a especificação de quais classes serão instrumentadas, quais campos serão raiz dos grafos compartilhados de memória, e quais serão os *locks* implementados nos métodos dos objetos compartilhados. Essas informações são compartilhadas tanto pelo servidor da DSM, quanto pelos clientes que ingressam no ambiente de DSM, para direcioná-los no momento da requisição de acesso ao campo da classe instrumentada. Nesse exemplo em específico, todos os métodos do *classpath* irão possuir *lock*, com sincronização de acesso de escrita.

Com isso, no momento da inicialização da API, há uma reimplementação de todos os métodos disponíveis no *classpath* da aplicação, adicionando a informação de sincronicidade e proteção de escrita para distribuição do valor da propriedade.

## 3.10 Validação

O primeiro procedimento utilizado para validação do sistema desenvolvido foi a análise de seu funcionamento e comportamento e a validação a partir das premissas definidas por Jefferson

[JEF 85], elencadas em [SPO 01], em relação à implementação do algoritmo de protocolo de simulação otimista paralela, no que diz respeito às situações que o protocolo deve estar apto a responder, quando elas acontecerem durante sua execução.

Na tabela 3.6 é apresentado o relacionamento entre uma série de situações e o comportamento esperado de resposta do sistema, para conformidade com o protocolo otimista.

**Tabela 3.6 : Validação do modelo de comportamento de um processo lógico**

| Situação | Comportamento Esperado |
|---|---|
| Chegada de uma mensagem e a antimensagem correspondente está armazenada na DarfiaOutputQueue. | Ambas são descartadas |
| Chegada de uma mensagem e a antimensagem correspondente não está armazenada na DarfiaOutputQueue | O processo lógico verifica a marca de tempo da mensagem |
| Marca de tempo da mensagem que chegou é maior ou igual ao LVT | O evento correspondente é inserido na DarfiainputQueue para posterior execução. |
| Marca de tempo da mensagem que chegou é menor que o LVT do processo lógico | O processo lógico imediatamente ativa o procedimento de *rollback*. |
| Chegada de uma mensagem | O processo lógico verifica na DarfiaInputQueue se o evento correspondente já foi executado. |
| O evento correspondente à mensagem que a antimensagem deveria anular já foi executado | O processo lógico imediatamente ativa o procedimento de *rollback*. |
| O evento correspondente à mensagem que a antimensagem deveria anular não foi executado ou não chegou. | A antimensagem é inserida na DarfiaOutputQueue |
| Processo lógico ativa o *rollback* provocado por mensagem atrasada | O evento correspondente à mensagem é inserido na DarfiaInputQueue e o processo envia antimensagens para todas as mensagens enviadas, referentes a eventos executados com |

| | |
|---|---|
| | marca de tempo maior do que a marca de tempo da mensagem atrasada. |
| Processo lógico ativa o *rollback* provocado por antimensagem. | O processo envia antimensagens para todas as mensagens enviadas referentes a eventos executados com marca de tempo maior ou igual do que a marca de tempo da antimensagem. |
| *Rollback* | Todos os eventos executados acima do valor da marca de tempo da mensagem atrasada ou da antimensagem voltam para a DarfiaInputQueue para serem re-executados. |
| DarfiaInputQueueVazia | O processo lógico trata as mensagens e/ou antimensagens contidas na DafiaInputQueue, se existirem. |

Através da realização de testes forçando a ocorrência das situações elencadas acima, o protocolo comportou-se como esperado. Em fase de depuração, pode-se observar exatamente os mesmo comportamentos observados acima.

Em sequência à validação baseada no modelo de comportamento aparente do processo lógico, baseado no par situação-comportamento esperado, é feita a validação dos resultado do algoritmo sequencial de cada unidade de trabalho do protocolo, realizando testes sequenciais, ou seja, sem encadeamento de filas (um sistema M/M/1). Os resultados obtidos do protocolo foram comparados com os resultados obtidos através do algoritmo SMPL.

Os testes foram executados e comparados com os resultados apresentados em [BAL 2005], assumindo a sua comparação com os resultados obtidos através do SMPL, além dos resultados proveniente da compilação e execução do próprio SMPL, como prova real dos resultados. Nesses testes, foram utilizados tempos fixos e exponenciais para a comparação.

Vale observar que nesse passo da validação, embora a mesma semente de geração de números aleatórios tenha sido utilizada, a geração de números aleatórios do algoritmo executado pelo SMPL (implementado em C) envolve computação de baixo nível e detalhes, entre outros, de

arraste de bits.

Pelo fato do algoritmo implementado no Darfa (implementado em Java) ser ligeramente diferente do SMPL, a seqüência de números aleatórios gerados a partir da mesma semente, difere. Entretanto, ao longo do tempo de simulação, observou-se que os resultados tendiam à igualdade, garantindo qualidade nas comparações dos resultados.

Uma segunda observação a ser feita para a implementação do SMPL é que suas estruturas são estáticas em memória, sendo altamente custoso em simulação compridas, envolvendo filas. Observou-se que o Darfia, em um cenário de simulação longa (ordem 1000000), conseguiu 1/5 do tempo de execução do SMPL.

A configuração de máquina utilizada para compilação e execução sequencial do Darfia e do SMPL é composta por um processador *Intel Core2Duo T6670*, com 2.20Ghz de *clock*, com 4GB de memória RAM disponível, em 800Mhz de barramento, utilizando o sistema operacional Linux OpenSuse 12.1, equipado com o Kernel 3.1.9-1.4, e utilizando o GCC 4.6-15.1.3 para compilação do código.

Os resultados são apresentados na tabela 3.6, na qual, por uma questão de organização, foram adotadas algumas siglas. São elas: *Ta:* Intervalo de chegada de um novo cliente; *Ts:* intervalo de utilização do centro de serviço; *Te:* Marcação de tempo de fim de simulação; Util.: taxa de utilização do centro de serviço; *M.B.P.: Mean Busy Period*, ou período médio ocupado do centro de serviço; *M.Q.L.: Mean Queue Length*, ou tamanho médio da fila do centro de serviço; *O.C. Release:* contagem de total de eventos liberados pelo centro de serviço. *O.C. Queues*: contagem do total de eventos atrasados que foram atendidos pelo centro de serviço; *Exec.*: tempo total real de execução da simulação, em segundos; *Events Releases*: número de eventos que foram liberados pelo centro de serviço; *Events Queued:* número total de eventos que foram enfileirados no centro de serviço, independente de seu atendimento ou não. Para essa última medida, a explicação da diferença da contagem do Darfia e do SMPL vem abaixo..

**Tabela 3.7: Validação do algoritmo sequencial do Darfia em relação ao SMPL**

| Algoritmos | SMPL | | | | | | DARFIA | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Configuração/Medidas | Util. | M. B. P. | M. Q. L. | O. C. Release | O. C. Queues | Exec. (s) | Util. | M. B. P. | M. Q. L. | Events Releases | Events Queued | Exec. (s) |
| ***Tempos Fixos*** | | | | | | | | | | | | |
| Ta=2, Ts=3, Te=20000 | 0.99 | 3.0 | 1666.33 | 6666 | 6666 | ~4.7 | 0.99 | 3.0 | 1666.33 | 6666 | 9999 | ~0.4 |
| Ta=3, Ts=2, Te=20000 | 0.66 | 2.0 | 0.00 | 6667 | 0 | ~2.8 | 0.66 | 1.99 | 0.00 | 6667 | 0 | ~0.4 |
| Ta=2, Ts=3, Te=50 | 0.96 | 3.0 | 3.84 | 16 | 16 | ~1.3 | 0.96 | 3.0 | 3.84 | 16 | 24 | ~0.07 |
| Ta=100, Ts=200, Te=200000 | 1.00 | 200.0 | 500.00 | 1000 | 1000 | ~1.7 | 0.99 | 199.8 | 500.0 | 1000 | 1999 | ~0.1 |
| Ta=200, Ts=100, Te=200000 | 0.50 | 100.0 | 0.00 | 1000 | 0 | ~1.5 | 0.50 | 99.9 | 0.0 | 1000 | 0 | ~0.1 |
| Ta=2, Ts=3, Te=200000 | 1.00 | 3.0 | 16666.3 | 66666 | 66666 | ~20.4 | 0.99 | 3.0 | 16666.3 | 66666 | 99999 | ~4.6 |
| Ta=3, Ts=2, Te=200000 | 0.66 | 2.0 | 0.00 | 66667 | 0 | ~1.5 | 0.66 | 1.99 | 0.0 | 66667 | 0 | ~0.5 |
| ***Tempos Aleatórios (Exponenciais em Ta e Ts com Desvio 0)*** | | | | | | | | | | | | |
| Ta=2, Ts=3, Te=20000 | 0.99 | 2.9 | 1731.97 | 6670 | 6670 | ~1.2 | 0.99 | 3.01 | 1743.30 | 6627 | 10114 | ~0.5 |
| Ta=3, Ts=2, Te=20000 | 0.63 | 1.95 | 1.12 | 6542 | 4193 | ~1.4 | 0.67 | 2.00 | 1.53 | 6753 | 4582 | ~0.4 |
| Ta=2, Ts=3, Te=50 | 0.97 | 4.12 | 5.16 | 12 | 12 | <1 | 0.67 | 3.08 | 4.95 | 11 | 21 | ~0.07 |
| Ta=100, Ts=200, Te=200000 | 0.99 | 202.9 | 510.69 | 984 | 984 | ~1.1 | 0.99 | 204.86 | 543.17 | 975 | 2057 | ~0.2 |
| Ta=200, Ts=100, Te=200000 | 0.53 | 103.48 | 0.65 | 1025 | 558 | ~1.3 | 0.54 | 104.30 | 0.63 | 1039 | 563 | ~0.1 |
| Ta=2, Ts=3, Te=200000 | 1.00 | 2.99 | 17101.3 | 66694 | 66694 | ~20.0 | 0.99 | 2.98 | 16857.0 | 66985 | 100629 | ~6.2 |
| Ta=3, Ts=2, Te=200000 | 0.66 | 1.98 | 1.31 | 66794 | 44125 | ~1.1 | 0.66 | 1.99 | 1.35 | 67040 | 44602 | ~2.0 |

No SMPL, ao final da simulação, chama-se a função *report()*, que retorna alguns resultados de contagens de operações, para informar ao usuário do sistema alguns dados estatísticos sobre o tempo simulado. Uma dessas contagens é sobre o número de operações de *QUEUE*. A implementação do SMPL coloca o número de operações de *QUEUE* como sendo o número de eventos que foram enfileirados e foram atendidos pelo centro de serviço do processo lógico dentro do tempo de simulação. Os eventos que foram enfileirados no centro de serviço e não foram processados não são contados nessa estatística.

Já na implementação do Darfia, essa contagem acontece de maneira diferente. A estatística anotada em *Events Queued* diz respeito a eventos que foram enfileirados, independente de sua execução no centro de serviço ou não. Ou seja, como a execução da simulação finaliza antes de esvaziar a fila do centro de serviço, os eventos que não foram executados, também são contados como eventos enfileirados, diferindo os valores em relação ao SMPL.

Para comprovação da validade dos resultados obtidos na implementação do Darfia, o SMPL foi modificado em tempo de coleta de testes para validação total dos resultados e essa medida foi comprovada como coerente, em relação aos dados obtidos na modificação do SMPL.

Após a validação do comportamento sequencial da execução do Darfia em relação ao SMPL, testes foram realizados com processamento paralelo no ambiente de memória compartilhada distribuída, para executar procedimentos de *rollback* nos processos lógicos.

O ambiente DSM foi configurado para os testes utilizando como nó raiz do grafo de objetos compartilhados pela API do Terracotta, o campo *simulation* da classe Darfia. Esse campo armazena um objeto do tipo *DarfiaSimulation,* responsável por representar a simulação em si.

Além disso, todas as classes do módulo de todos os módulos de construção da plataforma Darfia foram instrumentadas, no sentido de garantir o acesso exclusivo às variáveis compartilhadas em memória.

## 3.11 Módulo Web

O propósito do módulo web da plataforma Darfia é para que um pesquisador que se utilize da plataforma, possa criar um modelo de simulação paralela qualquer e executar a simulação inteiramente pelo navegador, independente de plataforma, ao invés de usar características muito particulares, como linhas de comando, acesso nativo a JVM e o sistema operacional Linux.

Nessa abordagem de utilização, a plataforma de simulação continua da mesma forma já descrita no documento original, entretanto a solicitação de início do processo de simulação é feita pelo navegador. Com isso, os resultados apresentados por essa plataforma são os mesmos encontrados no documento original.

Para a construção dessa nova interface, empregou-se a tecnologia para navegadores modernos de internet conhecida como Web Applications 1.0 [HICK 2005], ou HTML5, especificada pelo grupo WHATWG. Utilizou-se recursos como a especificação de *WebSockets* [HICK 2012], que possibilita a criação de conexões persistentes *full-duplex,* permintindo transferência de dados simultâneas entre ambas as partes da conexão, assim como acontece em uma conexão de *sockets* de baixo nível, utilizando protocolos já conhecidos, como o *TCP*. A conexão do tipo *WebSocket* acontece sobre o protocolo já conhecido *HTTP* [IETF 99], como uma forma de solucionar a característica *stateless* desse tipo de solicitação.

Utilizando-se dessas novas tecnologias, o funcionamento básico da ferramenta inicia-se através do documento *HTML* apresentado ao navegador de internet. Após a configuração da simulação, também via documento XML, esse documento conecta-se a uma aplicação *web J2ME* (aqui referenciada como *darfia-web),* hospedada em um servidor do tipo *container web*, responsável por implementar a especificação de *Servlets*[MOR 2009]. Para a construção da ferramenta, utilizou-se o *container web Jetty,* na versão 8.1.1.v20120215, disponibilizando um *servlet* que permita conexões de *WebSockets*.

Quando o navegador cliente solicita o ínicio da simulação, é avaliado o número de processos lógicos que serão utilizados para executar a configuração especificada, e então, para esse número de processos lógicos é criado o mesmo número de conexões do tipo *WebSocket* no cliente, que resultará no mesmo número de processos físicos na máquina hospedeira do servidor. Para a criação desses processos físicos no servidor, utilizou-se em sua implementação, objetos do tipo *java.lang.Process*, onde a JVM permite a criação de processos nativos na máquina hospedeira, através da especificação da linha de comando a ser executada, bem como os parâmetros envolvidos nessa execução.

Após a criação desses processos físicos locais, representando processos lógicos, a saída padrão e saída de erros desses processos lógicos são unificadas e direcionadas para as conexões do tipo *WebSocket* respectivas. Com esse redirecionamento de saídas produzidas pelo processos para as conexões persistentes, os dados produzidos pela simulação são enviados ao cliente (navegador), de forma independente, e em tempo real.

A cada chegada de dados no cliente, esses são renderizados na página, emulando o comportamento de terminais de um sistema operacional fictício. Ao término da simulação, as conexões são fechadas, e o processos de simulação é finalizado também no navegador, gerando um novo estado pronto para uma nova simulação.

A figura 3.12 ilustra um diagrama geral da arquitetura de software criada e empregada para a fabricação do módulo *web.*

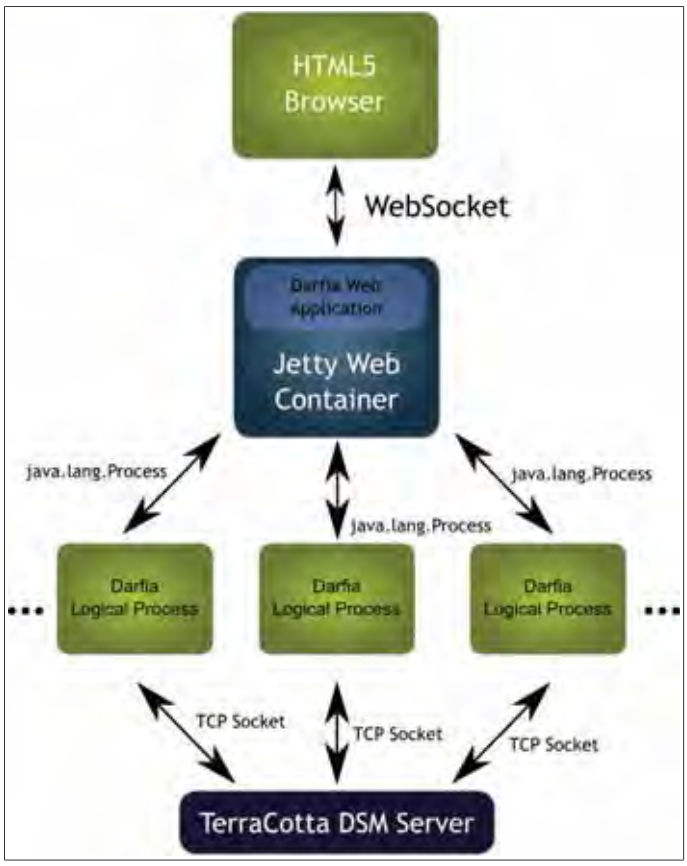


Figura 3.12: Diagrama geral de funcionamento do módulo *web* Darfia

A figura 3.13 ilustra o processo de configuração da plataforma web.


Figura 3.13: Sessão de configuração de simulação.

A figura 3.14 apresenta um terminal fictício dentro do navegador, já com os resultados obtidos através do redirecionamento da saída do processo físico do servidor.


Figura 3.14: Saída do processo físico, ilustrada em um terminal fictício dentro do navegador.

Dessa forma, utilizando-se das tecnologias e da arquitetura ilustrada acima, facilitou-se a utilização do protocolo, enquanto ambiente de testes personalizados de simulação paralela.

## 3.12 Considerações finais

Neste capítulo foi apresentada a plataforma de simulação Darfia. Foi descrito os padrões de projeto empregados no desenvolvimento do trabalho, bem como a definição de seus modulos de construção. Foi descrito as características da configuração da simulação através de documentos XML, e apresentado também a configuração da API do Terracotta, além de diagramas de classe, atividade e estado, exemplificando e demonstrando o funcionamento regular da plataforma. Finalmente, foi validado através de técnicas de comparações com resultados já provados e em comparação com dispositivos sequenciais para garantir a qualidade dos dados. Com a plataforma validada, no capítulo 4 serão discutidos testes de desempenho de simulação de modelos de redes de fila.

# Capítulo 4

# Avaliação de redes de filas utilizando a implementação do protocolo de simulação paralela

## 4.1 Considerações Iniciais

Para demonstrar a eficiência e utilização da plataforma Darfia, diversos testes foram realizados com o intuito de simular situações reais de encadeamento de eventos em filas de diversos sistemas. Tais simulações foram realizadas em cenários comuns encontrados em sistemas computacionais reais, como a organização de componentes eletro-eletrônicos em um sistema computacional, bem como a organização de processamento em um ambiente de produção de alto desempenho.

Um segundo conjunto de testes realizados na plataforma Darfia avaliou o comportamento da simulação através da utilização de um procedimento sintético de análise de simulação computacional orientada a eventos, conhecido como PHOLD [FUJ 90b]. Na implementação desse método, o agendamento de eventos foi realizado localmente, em cada processo lógico. Cada um dos processos lógicos da simulação produziu somente um evento em si mesmo, de chegada e partida de

cliente, sendo alocado em cada centro de serviço próprio. Ao realizar a liberação do centro de serviço do processo lógico, o evento de partida gera um evento encadeado para outro processo lógico da simulação. Através da configuração da simulação, todos os processos lógicos podem enviam a todos os outros, entretanto, somente um processo lógico é escolhido aleatoriamente por ciclo de LVT.

Para melhorar a qualidade dos testes, foi configurada na plataforma a carga de trabalho a cada evento no centro de serviço (tanto eventos de chegada quanto eventos de partida), e essa carga de trabalho é medida em operações de ponto flutuante. Tais operações foram implementadas através da utilização da classe nativa *Double,* que é capaz de armazenar tipos de dados de ponto flutuante com dupla precisão em 64 bits, compatíveis com os padrões definidos IEEE sobre armazenamento e formação de dados flutuantes de precisão dupla [IEEE 85]. Através da configuração da simulação, tais eventos executam um número previamente definido de operações desse tipo.

As métricas utilizadas para esse estudo foram as seguintes:

- Comparação com tempo real sequencial e paralelo de diversos cenário hipotéticos de simulação computacional, utilizando a API do Terracotta.
- Comprimento médio de *rollbacks* nas simulações paralelas.
- Tempo real gasto na realização de *rollbacks*.

Para as comparações de tempo real da execução sequencial e paralela, foram utilizados resultados já conhecidos encontrados em [POR 2005]. A análise sequencial dos cenários foi executada pelo algoritmo SMPL, o mesmo utilizado para validar a simulação sequencial de eventos dos processos lógicos da plataforma.

## 4.2 Descrição dos modelos

Foram utilizados quatro modelos para o estudo de simulação com a plataforma Darfia, configuráveis através de definições como carga de trabalho e tempos de chegada e serviços nos processos lógicos.

As configurações disponíveis e utilizadas para definir as diversas variações dos modelos são as seguintes:

- Número de operações de ponto flutuante em eventos de chegada e partida do centro de serviço.
- Perfil de distribuição de probabilidade para média de tempo de alocação no centro de serviço, média de tempo de chegada de novos eventos nos processos lógicos, e média de tempo gasto em centros de serviços remotos para eventos encadeados.
- Probabilidade de encadeamento ou saída de eventos entre processos lógicos, bem como os endereçamentos de eventos para os componentes da simulação.

Na figura 4.1 é ilustado o primeiro modelo (Modelo Hipotético 1), formado por dois processos lógicos encadeados, onde o primeiro processo lógico (PL1) recebe eventos de chegada e saída e obrigatoriamente os encadeia para o segundo processo lógico (PL2), que irá processar os mesmo.

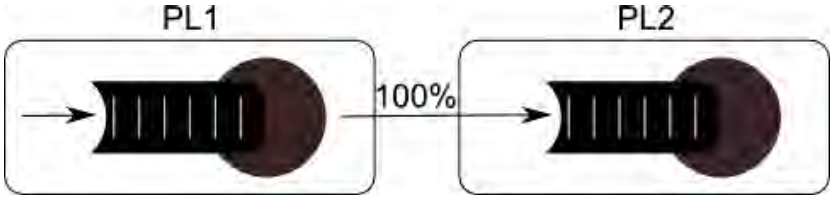


Figura 4.1 – Modelo Hipotético 1

Para esse primeiro modelo hipotético, definiu-se alguns cenários de configuração, semelhantes aos encontrados em [POR 2005], para critérios de comparação de resultados em tempos fixos, definidos na tabela 4.1. Ou seja, no cenário 1 do modelo hipotético 1, temos a situação em que o processo lógico 1 irá receber eventos em ordem de chegada fixa em 3 unidades de tempo, e cada evento gastará um tempo fixo de 5 unidades de tempo no centro de serviço. Após ser liberado do centro de serviço do primeiro processo lógico, esse evento será encadeado para o processo lógico 2, onde sua chegada depende do encadeamento, entretanto, uma vez que sua entrada se der, esse evento gastará um tempo fixo de 9 unidades de tempo no centro de serviço do segundo processo lógico. Esse ciclo se repetirá até que a simulação chegue em 10000 unidades de tempo.

**Tabela 4.1: Configurações para o modelo hipotético 1**

| PL/Configuração | Tempo de Chegada | Tempo de Serviço | Tempo de Simulação |
|---|---|---|---|
| | ***Cenário 1*** | | |
| Processo Lógico 1 | Fixo em 3 | Fixo em 5 | 10000 |
| Processo Lógico 2 | Não Aplicável | Fixo em 9 | 10000 |
| | ***Cenário 2*** | | |
| Processo Lógico 1 | Fixo em 5 | Fixo em 3 | 10000 |
| Processo Lógico 2 | Não Aplicável | Fixo em 12 | 10000 |

Na figura 4.2 representa-se o segundo modelo a ser estudado (Modelo Hipotético 2), formado também por dois processos lógicos. Entretanto, no estudo desse modelo, adotou-se a probabilidade de 50% do evento, ao ser encadeado pelo primeiro processo lógico (PL1), sair do sistema, e 50% do mesmo evento ser encadeado para o segundo processo lógico (PL2). Nesse segundo modelo, assim como primeiro, somente o primeiro processo lógico (PL1) recebe a chegada de eventos e desempenha o encadeamento.

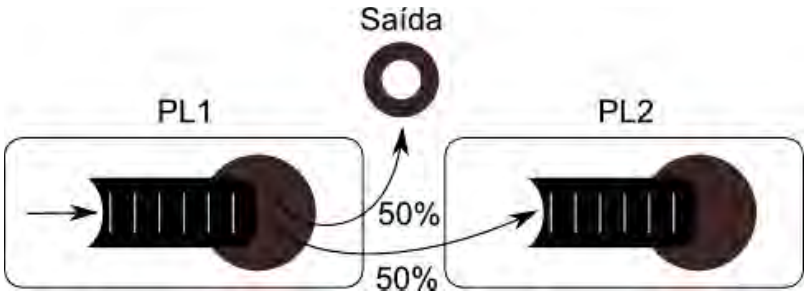


Figura 4.2 – Modelo Hipotético 2

Para o segundo modelo hipotético, definiu-se uma configuração, também semelhante ao encontrado em [POR 2005], também para critérios de comparação de resultados em tempos fixos, definidos na tabela 4.2.

**Tabela 4.2: Configurações para o modelo hipotético 2**

| PL/Configuração | Tempo de Chegada | Tempo de Serviço | Tempo de Simulação |
|---|---|---|---|
| Processo Lógico 1 | Fixo em 2.5 | Fixo em 1.5 | 1000 |
| Processo Lógico 2 | Não Aplicável | Fixo em 4.5 | 1000 |

O terceiro modelo real a ser estudado (Modelo Hipotético 3) é apresentado pela figura 4.3,

composta por três processos lógicos. Nesse modelo, um evento, ao ser encadeado pelo primeiro processo lógico (PL1), tem 50% de probabilidade de ser encadeado para o segundo processo lógico (PL2) e 50% de chances de ser encadeado para o terceiro processo lógico (PL3). Nesse terceiro modelo, assim como os demais, somente o primeiro processo lógico (PL1) recebe a chegada de eventos e realiza o encadeamento de eventos para os demais processos lógicos da simulação.

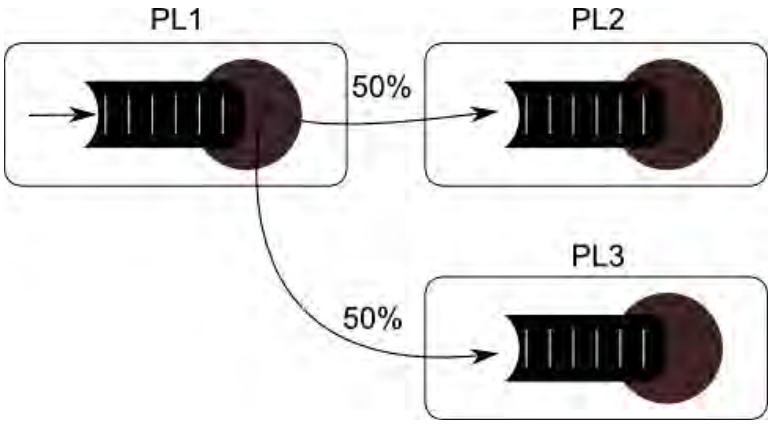


Figura 4.3 – Modelo Hipotético 3

Para esse modelo, assim como nos anterior, definiu-se somente uma configuração, também semelhante ao encontrado em [POR 2005], também para critérios de comparação de resultados em tempos fixos, definidos na tabela 4.3.

**Tabela 4.3: Configurações para o modelo hipotético 3**

| PL/Configuração | Tempo de Chegada | Tempo de Serviço | Tempo de Simulação |
|---|---|---|---|
| Processo Lógico 1 | Fixo em 5 | Fixo em 3 | 10000 |
| Processo Lógico 2 | Não Aplicável | Fixo em 9 | 10000 |
| Processo Lógico 3 | Não Aplicável | Fixo em 9 | 10000 |

E finalmente, o quarto modelo, utilizando a modelagem proposta pelo método PHOLD, define um modelo sintético de estudo.

Aqui cabe uma explicação do que é o PHOLD: é um *benchmark* sintético, que permite explorar um cenário de pior caso, isto é, a execução de um evento implica em pouca computação executada, enquanto se explora a comunicação entre os processos lógicos. Nesse *benchmark*, N processos lógicos são mapeados entre processadores disponíveis. Uma população fixa de eventos é pré-existente em cada PL. Quando um processo recebe um evento, gera um novo evento com

incremento de tempo exponencialmente distribuído e  cujo destino é escolhido aleatoriamente.

Para tornar esse estudo possível, a plataforma Darfia foi configurada inicialmente de forma com que três processos lógicos foram criados, e cada um desses foi assumido por um processo físico do sistema operacional. Para cada processo lógico, definiu-se que esses poderiam enviar mensagens encadeadas para todos os demais, com mesma probabilidade de escolha. Para iniciar o processamento no sistema, definiu-se somente uma mensagem em cada processo lógico (uma para alocação do centro de serviço e uma para liberação), e no momento da liberação do centro de serviço de cada processo lógico, essa mensagem é obrigatoriamente encadeada para algum dos processos lógicos. A figura 4.4 ilustra o modelo simplificado da simulação com o método de análise do PHOLD.

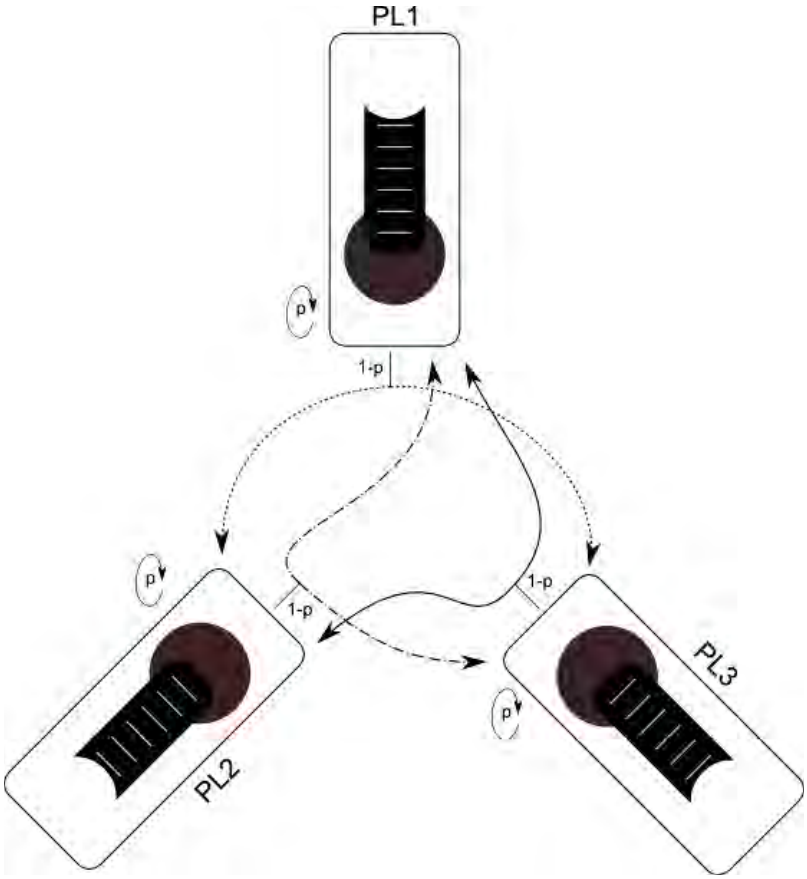


Figura 4.4 – Modelo simplificado do PHOLD.

Para esse modelo, definiu-se a configuração da tabela 4.4.

**Tabela 4.4: Configurações para o modelo PHOLD**

| PL/Configuração | Chegadas Iniciais | Tempo de Chegada | Tempo de Serviço | Tempo de Simulação |
|---|---|---|---|---|
| ***Cenário 1*** | | | | |
| Processo Lógico 1 | 1 | Fixo em 1 | Fixo em 2 | 10000 |
| Processo Lógico 2 | 1 | Fixo em 1 | Fixo em 2 | 10000 |
| Processo Lógico 3 | 1 | Fixo em 1 | Fixo em 2 | 10000 |
| ***Cenário 2*** | | | | |
| Processo Lógico 1 | 1 | Exponencial em 1 | Exponencial em 2 | 10000 |
| Processo Lógico 2 | 1 | Exponencial em 1 | Exponencial em 2 | 10000 |
| Processo Lógico 3 | 1 | Exponencial em 1 | Exponencial em 2 | 10000 |

Além desse modelo base, algumas modificações foram feitas para avaliar o andamento da simulação no método PHOLD, e tais exemplos foram colocados diretamente na avaliação de de desempenho do algoritmo. Para a execução dos testes e análise de desempenho com os modelos definidos acima, utilizou-se o mesmo ambiente citado na processo de validação do algoritmo sequencial e análise de resultados do SMPL, definidos na seção 3.10.

## 4.3 Análise de desempenho

A análise de desempenho foi realizada utilizando a API do Terracotta como provedor de ambiente DSM e de acordo com seu funcionamento, todas as classes que compõe a plataforma de simulação foram instrumentadas, para realizar o correto mapeamento do grafo compartilhado em memória, bem como a distribuição do espaço de endereçamento de memória compartilhada através da rede.

Todos os métodos foram marcados como auto-sincronizáveis para escrita, permitindo com que a informação se mantivesse coerente em situações de corrida ou disputa. Todos os resultados analisados nesse capítulo foram obtidos utilizando esse ambiente.

Não foram avaliados resultados de *speedup* da simulação pois o foco do trabalho foi a produção de um núcleo que fosse robusto e ao mesmo tempo personalizável, de acordo com as necessidades impostas pelos usuários da plataforma de simulação. Em alguns pequenos testes realizados nesse sentido, não houveram bons resultados devido o alto *overhead* colocado pela instrumentação da API do Terracotta. A melhora desse comportamento está anotada como sugestão para trabalhos futuros.

Por questões de organização das informações, siglas foram adotadas para representar os

conteúdos das tabelas dessa são. São elas: *TPar(s)*: tempo real da execução da simulação paralela para a configuração em questão, medidos em segundos; *O.C.R.*: *Operation Count Release,* ou eventos liberados por todos os centros de serviço; *O.C.Q.: Operation Count Queue,* ou eventos enfileirados por todos os centros de serviço; *E.Q.D.: Events Queue Delay*, ou média de atraso na alocação de eventos, de todos os eventos processados; *Util.: Utilization,* ou taxa de utilização do centro de serviço; *M.B.P: Mean Busy Period,* ou média de ocupação do centro de serviço; *M.Q.L.: Mean Queue Length,* ou média do comprimento da fila do centro de serviço; *N.R.: Number of Rollback*, ou contagem de rollbacks por processo lógico; *M.R.L.: Mean Primary Rollback Length,* ou comprimento médio de *rollbacks* primários. *T.R.B.: Total of Seconds in Primary Rollback,* ou tempo gasto em procedimento de *rollback*, em segundos. *O.P.F.:* Operações de Ponto Flutuante em eventos de entrada e saída do centro de serviço.

A primeira análise a ser feita é a referente ao modelo hipotético 1. Para esse teste, utilizou-se as configurações definidas para o modelo em dois cenários distintos. Um deles envolvendo operações de ponto flutuante em eventos de chegada e partida, caracterizando eventos de com gasto computacional e em um cenário sem operações, caracterizando eventos sem carga. A configuração básica da plataforma Darfia para simular ambos modelos estão ilustradas, para o cenário 1, na figura 4.5a.

```
<simulation name="aSimpleSimulation" endtime="10000"
    tracelevel="0">

    <logicalprocesses>

        <logicalprocess id="CPU1" phisicalprocessid="0">

            <arrivalproducer
                operationsnumberinarrival="10000"
                operationsnumberindeparture="10000"
                meanarrivaltime="fixed:3"
                meanservicetime="fixed:5" />

            <chaining quitprob="0" contprob="1" targets="DISK1"
                meanremoteservicetime="fixed:9" />

        </logicalprocess>

        <logicalprocess id="DISK1" phisicalprocessid="1" />

    </logicalprocesses>

</simulation>
```

Figura 4.5a – Configuração do modelo hipotético 1 – Cenário 1.

No primeiro cenário, dois nós <*logicalprocess*> foram definidos, onde hipotéticamente, há o encadeamento de eventos saindo do processo lógico *CPU1* para o processo lógico *DISK1*, com

probabilidade de 100%, indicando que toda liberação feita pelo primeiro processo lógico será encadeada para o segundo, a partir do momento que o evento for liberado do primeiro. Além disso, há a previsão do colapso no sistema, dado que o tempo de chegada é menor que o tempo de consumo do centro de serviço, e a chegada de cliente é aleatóriamente infinita.

E para configurar a plataforma Darfia para o cenário 2, utilizou-se o documento ilustrado na figura 4.5b.

```
<simulation name="aSimpleSimulation" endtime="10000"
    tracelevel="0">

    <logicalprocesses>

        <logicalprocess id="CPU1" phisicalprocessid="0">

            <arrivalproducer
                operationsnumberinarrival="10000"
                operationsnumberindeparture="10000"
                meanarrivaltime="fixed:5"
                meanservicetime="fixed:3" />

            <chaining quitprob="0" contprob="1" targets="DISK1"
                meanremoteservicetime="fixed:12" />

        </logicalprocess>

        <logicalprocess id="DISK1" phisicalprocessid="1" />

    </logicalprocesses>

</simulation>
```

Figura 4.5b – Configuração do modelo hipotético 1 – Cenário 2.

Nesse segundo cenário, também assumiu-se a mesma configuração, somente com alterações de parâmetros de tempos de chegadas e tempos de serviço gastos no centro de serviço.

Os resultados da simulação estão apresentados na tabela 4.5.

**Tabela 4.5: Análise de desempenho do modelo hipotético 1.**

| TPar(s) | PL | O.C.R. | O.C.Q. | E.Q.D. | Util. | M.B.P. | M.Q.L. | N.R. | M.R.L. | T.R.B.(s) | O.P.F. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ***Cenário 1*** | | | | | | | | | | | |
| ~101 | PL0 | 2000 | 3333 | 756.5 | 0.99 | 5 | 666.6 | 0 | 0 | 0 | 0 |
| | PL1 | 24 | 2007 | 38.0 | 0.02 | 9 | 978.6 | 7.0 | 2.28 | 0.162 | 0 |
| ~104 | PL0 | 2000 | 3333 | 756.5 | 0.99 | 5 | 666.6 | 0 | 0 | 0 | 10000 |
| | PL1 | 58 | 2008 | 106.0 | 0.04 | 9 | 945.7 | 7.0 | 2.42 | 0.08 | 10000 |
| ***Cenário 2*** | | | | | | | | | | | |
| ~84 | PL0 | 2000 | 0 | 0 | 0.6 | 3 | 0 | 0 | 0 | 0 | 0 |
| | PL1 | 30 | 2013 | 87.5 | 0.03 | 12 | 973.8 | 10.0 | 2.40 | 0.188 | 0 |
| ~86 | PL0 | 2000 | 0 | 0 | 0.6 | 3 | 0 | 0 | 0 | 0 | 10000 |
| | PL1 | 32 | 2014 | 84.51 | 0.03 | 12.0 | 972.8 | 8.0 | 3.0 | 0.134 | 10000 |

Os resultados obtidos durante a execução do modelo hipotético 1 demonstram claramente o alto tempo real gasto na simulação paralela devido a fatores como a carga de processamento (pequena no processamento paralelo), e a introdução da camada de manipulação de *bytecodes* e sincronização, proveninete da API do Terracotta, além, claro, da simplicidade do modelo estudado, para uma abordagem paralela.

No cenário 1, foi demonstrado que o colocar carga útil no processamento dos eventos, o centro de serviço ficou mais tempo ocupado, causando uma espera maior. Esse resultado é demonstrado pelo aumento do atraso médio da fila de eventos (E.Q.D), que passou de 38 para 106. O atraso é medida através do tempo local que o evento chegou no processo lógico, e o tempo que ele foi processado, ao sair da fila do centro de serviço. Já no cenário 2, devido o maior tempo de serviço no processo lógico 2, não observa-se esse comportamento.

Observa-se que o número de *rollbacks* representa cerca de 13% no primeiro cenário e 25% no segundo cenário, baseado no montante total de liberações de eventos.

A segunda análise, referente ao modelo hipotético 2 está descrita na tabela 4.6. Nessa análise também foi realizado o cálculo de operações de ponto flutuante, na tentativa de incrementar os cálculos feitos por eventos, na tentativa de aumentar o número de *rollbacks,* garantindo tempo de processamento gasto. A configuração utilizada para configurar a plataforma Darfia para o modelo hipotético 2 está ilustrada na figura 4.6.

```
<simulation name="aSimpleSimulation" endtime="1000"
    tracelevel="0">

    <logicalprocesses>

        <logicalprocess id="CPU1" phisicalprocessid="0">

            <arrivalproducer
                operationsnumberinarrival="10000"
                operationsnumberindeparture="10000"
                meanarrivaltime="fixed:2.5"
                meanservicetime="fixed:1.5" />

            <chaining quitprob="0.5" contprob="0.5" targets="DISK1"
                meanremoteservicetime="fixed:4.5" />

        </logicalprocess>

        <logicalprocess id="DISK1" phisicalprocessid="1" />

    </logicalprocesses>

</simulation>
```

Figura 4.6 – Configuração do modelo hipotético 2.

Nesse segundo modelo hipotético, definiu-se o mesmo modelo de dois processos lógicos,

*CPU1* e *DISK1*, onde, além das alterações de configurações de chegadas/tempos de serviço, alterou-se a probabilidade de saída/encadeamento de eventos, garantindo que haja a probabilidade de 50% de saída do sistema, ou 50% de encadeamento de eventos. Além disso, o sistema prevê que não haverá filas no consumo do centro de serviço do primeiro processo lógico, devido os valores de chegada de eventos serem maiores que os valores de consumo de tempo de serviço.

Os resultados da simulação estão apresentados na tabela 4.6.

**Tabela 4.6: Análise de desempenho do modelo hipotético 2.**

| TPar(s) | PL | O.C.R. | O.C.Q. | E.Q.D. | Util. | M.B.P. | M.Q.L. | N.R. | M.R.L. | T.R.B.(s) | O.P.F. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ~31 | PL0 | 400 | 0 | 0 | 0.6 | 1.5 | 0 | 0 | 0 | 0 | 0 |
| | PL1 | 200 | 117 | 4.05 | 0.89 | 4.5 | 0.79 | 39.0 | 1.15 | 0.383 | 0 |
| ~34 | PL0 | 400 | 0 | 0 | 0.6 | 1.5 | 0 | 0 | 0 | 0 | 10000 |
| | PL1 | 206 | 105 | 2.04 | 0.9 | 4.5 | 0.427 | 49.0 | 1.06 | 0.626 | 10000 |

De acordo com a configuração, em média, metade dos eventos liberados pelo primeiro processo lógico foram encadeados para o segundo processo lógico, resultando em torno de 20% de taxa de *rollbacks*, em torno do volume total de liberações.

Nota-se também um aumento no número de *rollbacks* graças a inserção de operações de ponto flutuante (operações simples que envolvam divisões e multiplicações de números de ponto flutuante, informadas na configuração do cenário) realizados no processamento de eventos durante o processo de simulação. Enquanto o primeiro processo lógico (PL0) realiza cálculos de alto custo computacional, o tempo real de execução é atrasado, fazendo com que o segundo processo lógico (com menos processamento de eventos, embora também haja operações de ponto flutuante), prossiga mais rápido na simulação, gerando um aumento no número de mensagens *straglers,* e consequentemente, ativando mais procedimentos de *rollback.* Nesse caso, também não há um atraso significativo no tempo de simulação, através do tempo total de procedimento de *rollbacks.*

A terceira análise, referente ao modelo hipotético 3 está descrita na figura 4.7.

```
<simulation name="aSimpleSimulation" endtime="10000"
    tracelevel="0">

    <logicalprocesses>

        <logicalprocess id="CPU1" phisicalprocessid="0">

            <arrivalproducer
                operationsnumberinarrival="0"
                operationsnumberindeparture="0"
                meanarrivaltime="fixed:5"
                meanservicetime="fixed:3" />

            <chaining quitprob="0" contprob="1" targets="DISK1,DISK2"
                meanremoteservicetime="fixed:9" />

        </logicalprocess>

        <logicalprocess id="DISK1" phisicalprocessid="1" />

        <logicalprocess id="DISK2" phisicalprocessid="2" />

    </logicalprocesses>

</simulation>
```

Figura 4.7 – Configuração do modelo hipotético 3.

Nesse terceiro modelo, observa-se a criação de três processos lógicos, com identificadores *CPU1, DISK1* e *DISK2*, em um sistema computacional onde eventos gerados pela CPU são encadeados para dois discos. Com 100% de probabilidade, todos os eventos que forem liberados do primeiro processo lógico serão encadeados para algum outro processo lógico e não sairão do sistema. Além disso, não haverá colapso no primeiro processo lógico em relação a fila do centro de serviço, devido o valor de consumo do centro ser menor que o valor das chegadas de eventos.

Nesse exemplo não foram utilizadas operações de ponto flutuante, observando somente o desempenho da simulação simples, onde para cada evento processado nos processos lógicos, não haja nenhum cálculo adicional ou consumo de tempo de processador.

Os resultados obtidos estão apresentados na tabela 4.7.

**Tabela 4.7: Análise de desempenho do modelo hipotético 3.**

| TPar(s) | PL | O.C.R. | O.C.Q. | E.Q.D. | Util. | M.B.P. | M.Q.L. | N.R. | M.R.L. | T.R.B.(s) | O.P.F. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 127.1 | PL0 | 2000 | 0 | 0 | 0.6 | 3 | 0 | 0 | 0 | 0 | 0 |
| | PL1 | 143 | 880 | 1.57 | 0.1 | 9 | 368.06 | 40.0 | 1.05 | 0.363 | 0 |
| | PL2 | 637 | 706 | 4.9 | 0.5 | 9 | 71.5 | 129 | 1.25 | 0.783 | 0 |
| 133.9 | PL0 | 2000 | 0 | 0 | 0.6 | 3 | 0 | 0 | 0 | 0 | 10000 |
| | PL1 | 1003 | 557 | 6.05 | 0.8 | 9 | 0.59 | 175 | 1.27 | 0.604 | 10000 |
| | PL2 | 1008 | 506 | 4.63 | 0.9 | 9 | 0.46 | 200 | 1.19 | 0.935 | 10000 |

Nesse cenário, observa-se um grande número de operações de *rollback* curtos nos processos

lógicos 2 e 3, representando no caso onde não há operações de ponto flutuante, a parcela em torno de 20% a 30% do número total de liberações, e no caso onde há operações de ponto flutuante, em torno de 20%. Esse comportamento se dá graças ao alto consumo do centro de serviço para eventos encadeados (9 unidades de tempo, em relação ao tempo de serviço no primeiro processo lógico, 3).

E finalmente, a configuração da plataforma Darfia para realizar uma pequena simulação do método PHOLD, através de alguns cenários de configuração.

No primeiro cenário de simulação, ilustrado pela figura 4.8a, a configuração foi realizada utilizando tempo de simulação de 200.0 unidades de tempo, e três processos lógicos ligados entre si. Todos os processos lógicos enviam mensagens a todos, inclusive para si mesmo, na mesma probabilidade. Além disso, somente um evento é colocado como população inicial em cada processo lógico, e as chegadas/tempos de serviço são fixas em 1.0 unidade de tempo.

```
<simulation name="aSimpleSimulation" endtime="200"
    tracelevel="0">

    <logicalprocesses>

        <logicalprocess id="LP1" phisicalprocessid="0">

            <arrivalproducer operationsnumberinarrival="0" operationsnumberindeparture="0" limitnumber="1"
                initialnumber="1" meanarrivaltime="fixed:1" meanservicetime="fixed:1" />

            <chaining quitprob="0" contprob="1" targets="LP1,LP2,LP3" meanremoteservicetime="fixed:1" />

        </logicalprocess>

        <logicalprocess id="LP2" phisicalprocessid="1">

            <arrivalproducer operationsnumberinarrival="0" operationsnumberindeparture="0" limitnumber="1"
                initialnumber="1" meanarrivaltime="fixed:1" meanservicetime="fixed:1" />

            <chaining quitprob="0" contprob="1" targets="LP1,LP2,LP3" meanremoteservicetime="fixed:1" />

        </logicalprocess>

        <logicalprocess id="LP3" phisicalprocessid="2">

            <arrivalproducer operationsnumberinarrival="0" operationsnumberindeparture="0" limitnumber="1"
                initialnumber="1" meanarrivaltime="fixed:1" meanservicetime="fixed:1" />

            <chaining quitprob="0" contprob="1" targets="LP1,LP2,LP3" meanremoteservicetime="fixed:1" />

        </logicalprocess>

    </logicalprocesses>

</simulation>
```

Figura 4.8a – Configuração do modelo PHOLD, cenário 1.

No segundo cenário de simulação para testes, o tempo foi reduzido para 10.0, e mantido o número inicial de eventos em cada processo lógico. Nesse cenário, foi definido que os tempos de chegadas e serviços seriam exponenciais com média em 1, e desvio em 1, ao invés de tempo fixo. A configuração está ilustrada pela figura 4.8b.

```
<simulation name="aSimpleSimulation" endtime="10"
    tracelevel="0">

    <logicalprocesses>

        <logicalprocess id="LP1" phisicalprocessid="0">

            <arrivalproducer operationsnumberinarrival="0" operationsnumberindeparture="0" limitnumber="1"
                initialnumber="1" meanarrivaltime="exponential:1,1" meanservicetime="exponential:1,1" />

            <chaining quitprob="0" contprob="1" targets="LP1,LP2,LP3" meanremoteservicetime="exponential:1,1" />

        </logicalprocess>

        <logicalprocess id="LP2" phisicalprocessid="1">

            <arrivalproducer operationsnumberinarrival="0" operationsnumberindeparture="0" limitnumber="1"
                initialnumber="1" meanarrivaltime="exponential:1,1" meanservicetime="exponential:1,1" />

            <chaining quitprob="0" contprob="1" targets="LP1,LP2,LP3" meanremoteservicetime="exponential:1,1" />

        </logicalprocess>

        <logicalprocess id="LP3" phisicalprocessid="2">

            <arrivalproducer operationsnumberinarrival="0" operationsnumberindeparture="0" limitnumber="1"
                initialnumber="1" meanarrivaltime="exponential:1,1" meanservicetime="exponential:1,1" />

            <chaining quitprob="0" contprob="1" targets="LP1,LP2,LP3" meanremoteservicetime="exponential:1,1" />

        </logicalprocess>

    </logicalprocesses>

</simulation>
```

Figura 4.8b – Configuração do modelo PHOLD, cenário 2.

No terceiro cenário de simulação para testes, o tempo foi definido em 200.0, e configurado um valor de 5 clientes iniciais no sistema. Nesse cenário, foi definido que os tempos de chegadas e serviços seriam fixos em 1. A configuração está ilustrada pela figura 4.8c.

```
<simulation name="aSimpleSimulation" endtime="200"
    tracelevel="1">

    <logicalprocesses>

        <logicalprocess id="LP1" phisicalprocessid="0">

            <arrivalproducer operationsnumberinarrival="0" operationsnumberindeparture="0" limitnumber="5"
                initialnumber="5" meanarrivaltime="fixed:1" meanservicetime="fixed:1" />

            <chaining quitprob="0" contprob="1" targets="LP1,LP2,LP3" meanremoteservicetime="fixed:1" />

        </logicalprocess>

        <logicalprocess id="LP2" phisicalprocessid="1">

            <arrivalproducer operationsnumberinarrival="0" operationsnumberindeparture="0" limitnumber="5"
                initialnumber="5" meanarrivaltime="fixed:1" meanservicetime="fixed:1" />

            <chaining quitprob="0" contprob="1" targets="LP1,LP2,LP3" meanremoteservicetime="fixed:1" />

        </logicalprocess>

        <logicalprocess id="LP3" phisicalprocessid="2">

            <arrivalproducer operationsnumberinarrival="0" operationsnumberindeparture="0" limitnumber="5"
                initialnumber="5" meanarrivaltime="fixed:1" meanservicetime="fixed:1" />

            <chaining quitprob="0" contprob="1" targets="LP1,LP2,LP3" meanremoteservicetime="fixed:1" />

        </logicalprocess>

    </logicalprocesses>

</simulation>
```

Figura 4.8c – Configuração do modelo PHOLD, cenário 3

No quarto cenário de simulação para testes, o tempo foi definido em 200.0, e configurado

um valor de 15 clientes iniciais no sistema. Nesse cenário, foi definido que os tempos de chegadas e serviços seriam fixos em 1. A configuração está ilustrada pela figura 4.8d.

```
<simulation name="aSimpleSimulation" endtime="200"
    tracelevel="1">

    <logicalprocesses>

        <logicalprocess id="LP1" phisicalprocessid="0">

            <arrivalproducer operationsnumberinarrival="0" operationsnumberindeparture="0" limitnumber="15"
                initialnumber="15" meanarrivaltime="fixed:1" meanservicetime="fixed:1" />

            <chaining quitprob="0" contprob="1" targets="LP1,LP2,LP3" meanremoteservicetime="fixed:1" />

        </logicalprocess>

        <logicalprocess id="LP2" phisicalprocessid="1">

            <arrivalproducer operationsnumberinarrival="0" operationsnumberindeparture="0" limitnumber="15"
                initialnumber="15" meanarrivaltime="fixed:1" meanservicetime="fixed:1" />

            <chaining quitprob="0" contprob="1" targets="LP1,LP2,LP3" meanremoteservicetime="fixed:1" />

        </logicalprocess>

        <logicalprocess id="LP3" phisicalprocessid="2">

            <arrivalproducer operationsnumberinarrival="0" operationsnumberindeparture="0" limitnumber="15"
                initialnumber="15" meanarrivaltime="fixed:1" meanservicetime="fixed:1" />

            <chaining quitprob="0" contprob="1" targets="LP1,LP2,LP3" meanremoteservicetime="fixed:1" />

        </logicalprocess>

    </logicalprocesses>

</simulation>
```

Figura 4.8d – Configuração do modelo PHOLD, cenário 4.

No quinto cenário de simulação para testes, o tempo foi definido em 200.0, e configurado um valor de 40 clientes iniciais no sistema. Nesse cenário, foi definido que os tempos de chegadas e serviços seriam fixos em 1. A configuração está ilustrada pela figura 4.8e.

```
<simulation name="aSimpleSimulation" endtime="200"
    tracelevel="1">

    <logicalprocesses>

        <logicalprocess id="LP1" phisicalprocessid="0">

            <arrivalproducer operationsnumberinarrival="0" operationsnumberindeparture="0" limitnumber="40"
                initialnumber="40" meanarrivaltime="fixed:1" meanservicetime="fixed:1" />

            <chaining quitprob="0" contprob="1" targets="LP1,LP2,LP3" meanremoteservicetime="fixed:1" />

        </logicalprocess>

        <logicalprocess id="LP2" phisicalprocessid="1">

            <arrivalproducer operationsnumberinarrival="0" operationsnumberindeparture="0" limitnumber="40"
                initialnumber="40" meanarrivaltime="fixed:1" meanservicetime="fixed:1" />

            <chaining quitprob="0" contprob="1" targets="LP1,LP2,LP3" meanremoteservicetime="fixed:1" />

        </logicalprocess>

        <logicalprocess id="LP3" phisicalprocessid="2">

            <arrivalproducer operationsnumberinarrival="0" operationsnumberindeparture="0" limitnumber="40"
                initialnumber="40" meanarrivaltime="fixed:1" meanservicetime="fixed:1" />

            <chaining quitprob="0" contprob="1" targets="LP1,LP2,LP3" meanremoteservicetime="fixed:1" />

        </logicalprocess>

    </logicalprocesses>

</simulation>
```

Figura 4.8e – Configuração do modelo PHOLD, cenário 5.

No sexto cenário de simulação para testes, o tempo foi definido em 200.0, e configurado um valor de 1 cliente inicial no sistema. Nesse cenário, foi definido que os tempos de chegadas e serviços seriam fixos em 1. Entretanto, o número de processos lógicos foi aumentado para 5, seguindo o mesmo rigor da simulação. Envio de eventos de todos os processos lógicos para todos os outros.

No sétimo cenário, seguiu-se o mesmo modelo do anterior, entretanto, utilizou-se 10 processos lógicos para realização de testes.

Os dados resultantes da simulação de todo os cenários estão apresentados na tabela 4.8.

**Tabela 4.8: Análise de desempenho do modelo PHOLD.**

| TPar(s) | Duração | PL | O.C.R. | O.C.Q. | E.Q.D. | Util. | M.B.P. | M.Q.L. | N.R. | M.R.L. | T.R.B.(s) | O.P.F. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ***Cenário 1*** | | | | | | | | | | | | |
| 146.71 | 200.0 | PL0 | 638 | 3580 | 19.44 | 0.87 | 28.28 | 229.64 | 69 | 103.79 | 12.13 | 0 |
| | | PL1 | 771 | 4158 | 17.61 | 0.93 | 24.09 | 229.99 | 70 | 121.88 | 14.50 | 0 |
| | | PL2 | 849 | 5066 | 20.38 | 0.88 | 23.52 | 260.88 | 78 | 134.17 | 16.13 | 0 |
| 184.43 | 200.0 | PL0 | 805 | 5455 | 20.66 | 0.77 | 29.23 | 335.54 | 87 | 127.24 | 18.68 | 1000000 |
| | | PL1 | 809 | 5822 | 21.51 | 0.82 | 28.96 | 311.91 | 87 | 134.26 | 21.48 | 1000000 |
| | | PL2 | 1154 | 7960 | 20.27 | 0.87 | 22.287 | 314.27 | 92.0 | 181.09 | 23.51 | 1000000 |
| ***Cenário 2*** | | | | | | | | | | | | |
| 1052.9 | 10.0 | PL0 | 6962 | 33089 | 26.62 | 0.99 | 28.67 | 3281.5 | 861 | 75.99 | 171.25 | 0 |
| | | PL1 | 9315 | 47393 | 26.02 | 0.99 | 24.22 | 3780.3 | 1146 | 85.14 | 225.95 | 0 |
| | | PL2 | 9628 | 49103 | 26.67 | 0.99 | 23.06 | 3745.3 | 1155 | 88.16 | 226.88 | 0 |
| 1152.5 | 10.0 | PL0 | 6889 | 34859 | 25.98 | 0.99 | 28.48 | 3445.2 | 810 | 84.31 | 184.32 | 1000000 |
| | | PL1 | 10200 | 58190 | 27.14 | 0.99 | 22.71 | 4119.7 | 1187 | 101.32 | 267.48 | 1000000 |
| | | PL2 | 10567 | 60326 | 27.92 | 0.99 | 22.25 | 4197.0 | 1180 | 106.35 | 273.53 | 1000000 |
| ***Cenário 3*** | | | | | | | | | | | | |
| 163.61 | 200.0 | PL0 | 520 | 3014 | 20.82 | 0.82 | 31.18 | 251.1 | 66.0 | 87.90 | 16.647 | 0 |
| | | PL1 | 830 | 5026 | 20.81 | 0.79 | 24.23 | 264.9 | 92.0 | 112.94 | 19.068 | 0 |
| | | PL2 | 908 | 6397 | 21.16 | 0.87 | 27.00 | 267.98 | 112 | 118.39 | 20.636 | 0 |
| ***Cenário 4*** | | | | | | | | | | | | |
| 174.66 | 200.0 | PL0 | 584 | 3400 | 21.52 | 0.85 | 29.67 | 276.69 | 72.0 | 91.83 | 16.922 | 0 |
| | | PL1 | 614 | 3802 | 19.88 | 0.87 | 31.97 | 260.91 | 74.0 | 101.60 | 17.246 | 0 |
| | | PL2 | 1111 | 8119 | 25.54 | 0.96 | 25.65 | 308.78 | 111 | 153.74 | 22.88 | 0 |
| ***Cenário 5*** | | | | | | | | | | | | |
| 198.59 | 200.0 | PL0 | 628 | 4141 | 23.72 | 0.9 | 29.48 | 289.01 | 69.0 | 118.78 | 17.406 | 0 |
| | | PL1 | 607 | 4276 | 26.38 | 0.89 | 29.56 | 287.67 | 71.0 | 118.21 | 17.253 | 0 |
| | | PL2 | 998 | 8453 | 25.76 | 0.89 | 25.10 | 339.41 | 91.0 | 191.04 | 22.074 | 0 |
| ***Cenário 6*** | | | | | | | | | | | | |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 210.50 | 200.0 | PL0 | 467 | 1521 | 14.15 | 0.96 | 24.75 | 102.12 | 56.0 | 54.37 | 8.15 | 0 |
| | | PL1 | 540 | 1938 | 15.91 | 0.98 | 25.52 | 103.08 | 64.0 | 63.45 | 8.074 | 0 |
| | | PL2 | 612 | 2112 | 14.35 | 0.99 | 26.62 | 91.09 | 71.0 | 64.76 | 9.857 | 0 |
| | | PL4 | 10 | 1931 | 0.285 | 0.04 | 2.6 | 146.23 | 160 | 19.12 | 1.562 | 0 |
| | | PL5 | 631 | 2366 | 16.64 | 0.91 | 26.25 | 111.62 | 84.0 | 60.47 | 10.194 | 0 |
| ***Cenário 7*** | | | | | | | | | | | | |
| 773.51 | 200.0 | PL0 | 631 | 3167 | 25.03 | 1.0 | 36.55 | 242.78 | 101 | 64.20 | 13.446 | 0 |
| | | PL1 | 702 | 3493 | 21.87 | 0.97 | 34.84 | 230.78 | 117 | 62.72 | 15.801 | 0 |
| | | PL2 | 709 | 4071 | 23.45 | 0.88 | 34.74 | 263.83 | 106 | 79.06 | 16.225 | 0 |
| | | PL4 | 685 | 3734 | 25.75 | 0.85 | 34.97 | 263.32 | 114 | 67.92 | 14.328 | 0 |
| | | PL5 | 754 | 3975 | 19.90 | 0.95 | 33.48 | 226.17 | 117 | 71.51 | 15.96 | 0 |
| | | PL6 | 713 | 3873 | 22.67 | 0.95 | 34.16 | 260.45 | 118 | 67.69 | 16.545 | 0 |
| | | PL7 | 669 | 3778 | 20.35 | 0.89 | 36.50 | 268.28 | 125 | 61.96 | 13.518 | 0 |
| | | PL8 | 646 | 3600 | 23.63 | 0.9 | 37.74 | 262.36 | 103 | 71.62 | 14.177 | 0 |
| | | PL9 | 677 | 3563 | 20.80 | 0.89 | 34.31 | 254.39 | 119 | 62.21 | 16.052 | 0 |
| | | PL10 | 666 | 3315 | 19.78 | 0.98 | 38.64 | 219.45 | 107 | 64.61 | 15.174 | 0 |

A análise dos dados do modelo PHOLD revela um aumento gradativo de comprimento médio da fila do centro de serviço, tempo médio na fila, período médio ocupado, número de procedimentos de *rollbacks* e *rollbacks* mais complexos, dado o tempo gasto total nesse procedimento, de acordo com o aumento de carga útil (operações de ponto flutuante) nos eventos, e consequentemente, nos processos lógicos.

Além disso, nota-se que com o aumento do número de processos lógicos na simulação, aumenta-se a quantidade de *rollbacks,* entretanto, eles são mais simples (número baixo de eventos em cada processo lógico), pois o seu comprimento é menor em relação a simulação com menos processos lógicos. Ainda no aumento de número de processos lógicos, nota-se também que a fila do centro de serviço é menor, além da espera (variação entre o momento que o evento deveria ser alocado no centro de serviço e o momento em que ele realmente foi) também ser menor. Esse comportamento se deve ao fato do aumento do número de processos lógicos caracterizar o aumento de paralelismo, além da maior distribuição de carga.

## 4.4 Considerações finais

Este capítulo apresentou resultados de análise e testes a partir de modelos naturais e sintéticos de redes de filas encadeadas, utilizando a plataforma de simulação paralela Darfia. De maneira geral, as execuções dos testes demonstraram um baixo desempenho na questão da

implementação paralela, por três principais motivos:

- A instrumentação de código realizado pela API do Terracotta, devido a camada de tratamento de chamadas que a API insere no fluxo normal de exeução, para realizar o mapeamento de endereçamento de mémória distribuída pela rede.
- A sincronicidade da distribuição de atualizações pela rede feita pela API do Terracota, e o tempo gasto ao adquirir o *lock* de um método (ou propriedade) em um objeto Java.
- Pela simplicidade dos modelos e a falta de carga útil atrelada aos eventos.
- Pela linguagem de programação utilizada, uma vez que os *bytecodes* são interpretados pela máquina virtual, pelo menos, em primeira instância.

O capítulo 5 apresenta as conclusões gerais desse trabalho, as contribuições, avanços e sugestões para a continuidade da plataforma.

# Capítulo 5

# Conclusões, contribuições e propostas para trabalhos futuros

## 5.1 Contribuições

Neste trabalho apresentou-se o desenvolvimento da plataforma Darfia para simulação distribuída, desenvolvida com base no protocolo otimista de sincronização de processos. Este trabalho contribui com uma vasta revisão bibliográfica, analisando conceitos básicos até elencando trabalhos recentes sobre adaptações específicas em ambientes extremamente controlados.

A principal fonte de contribuição para a comunidade científica da área é o início da construção de uma plataforma para simulação altamente configurável e expansível, e que ao mesmo tempo, consiga fornecer uma interface de comunicação e configuração com outros *softwares* já existentes.

Por ter sido desenvolvida em uma linguagem de programação comercial e altamente integrável, tem-se como recursos a construção de novas funcionalidades através da integração de sua configuração feita em XML, bem como todos os meios de comunicação que o Java oferece.

Além disso, o trabalho desenvolvido também apresenta resultados em torno da utilização da API do Terracotta para prover ambientes compartilhados de DSM, sob a ótica da facilidade e

expansão do ambiente disponível para computação de uma simulação distribuída.

## 5.2 Propostas para Trabalhos Futuros

Como sugestões para futuros trabalhos ao redor da plataforma Darfia sugere-se:

- Construir um gerenciador de carga de trabalho que consiga trabalhar com conceitos de granulosidade dinamicamente. Esse gerenciador de carga poderia interagir diretamente com a configuração do sistema a ser simulado, bem como com a análise constante do ambiente de simulação e o estado atual dos processos físicos, no que tange a disponibilidade e carga máxima de processamento. A definição aconteceria em tempo de execução, dinamicamente, equilibrando a equação de carga de processamento *versus* capacidade de trabalho dos processos [CHEN 2011].
- Além da construção de um balanceador de cargas, uma outra contribuição em torno da plataforma Darfia é a construção de um gerenciador de alocação de processos lógicos, com o objetivo da transferência dinâmica de processos lógicos para processos físicos. É sabido que a associação estática de processos lógicos com processos físicos pode levar a condições de baixo rendimento ou mal balanceamento de cargas na simulação [CHEN 2011], comprometendo o desempenho da simulação como um todo. A junção do balanceador de cargas com o gerenciador de processos lógicos pode resultar em um desempenho mais equilibrado em relação a modelos de simulação.
- Um outro ponto a ser construído na plataforma Darfia é a extensão da funcionalidade de processo lógico, para permitir que um PL possa simular redes de fila de espera, e não apenas uma rede básica no centro de serviço. Para essas filas, alguns perfis podem ser implementados, como filas FIFO (*First In, First Out*), LIFO (Last In First Out), etc. Além disso, a construção de filas prioritárias, bem como níveis de prioridades nas mensagens trocadas pelos LPs, permitindo preempções nos centros de serviço.
- Implementação de um mecanismo eficiente de cálculo do GVT (*Global Virtual Time*), após a implementação de todas as modificações mencionadas.
- Implementação de um mecanismo eficiente de *fossil collection*.
- Análise e otimização de *overhead* imposto pelo Terracotta, para diminuir tempo total real de

execução da simulação.

- Implementação de mecanismo de distribuição em processos remotos de carga para o módulo *web*. Na presente implementação só é possível criar processos lógicos locais, validando o funcionamento da ferramenta. Entretando, para propósitos de distribuição de carga, medição de tráfego e ganho de desempenho é necessário especificar quais nós da rede de simulação receberão os processos lógicos especificados na ferramenta de configuração *web*.

# Referências Bibliográficas


[AYA 92] AYANI, Rassul; RAJAEI, Hassan. Parallel Simultaion using Conservative Time Windows. In: WINTER SIMULATION CONFERENCE, WSC, 1992, Arlington. **Proceedings...** San Diego: SCS, 1992. p 709, 717.

[BAL 2005] BALIEIRO, Marta Oliveira da Silva. **Protocolo conservativo CMB para simulação distribuída**. Tese de Mestrado, Centro de Ciências Exatas e Tecnologia – UFMS, 2005.

[BAR 96] BARBOSA, Valmir C. Simulation. In: **An introduction to distributed algorithms**. Cambridge: MIT, 1996, p. 291-321.

[BAU 2008] BAUER Jr. , David W., An approach for the effective utilization of gp-gpus in parallel combined simulation . **Proceedings ...**. 2008.

[BAU 2009] BAUER Jr. , David W., Scalable Time Warp on Blue Gene Supercomputers . **ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation .** 2009 .

[BRU 03] BRUSCHI, Sarita Mazzini. **Um ambiente de simulação distribuída automático ASDA**. Tese de Doutorado, Instituto de Ciências Matemáticas e de Computação – USP, 2003.

[CAL 99] CALEGARIO, Vanusa Menditi. **Analise de desempenho de programação em lógica.** Rio de Janeiro: COPPE/UJRJ, 1999. Dissertação de Mestrado

[CAM 96] CAMPOS, Alvaro E. l ASTABURUAGA, Miguel Angel Castillo. **Checkpointing Through Garbage Collection**. 1996. Acesso em: <http://alesna.ing.puc.cl/~mca/chicago/paper.html>, 03 jul. 2009 .

[CHEN 2010] CHEN, Xiaowen; Lu, Zhonghai; Jantsch, Axel; Chen, Shuming. Supporting Distributed Shared Memory on Multi-core Network-on-Chips Using a Dual Microcoded Controller . Design, Automation & Test in Europe Conference & Exhibition (DATE). 39 – 44. Dresden.

[CHEN 2011] CHEN, Li-li; LÜ, Ya-shuai; YAO, Yi-ping; PENG, Shao-liang; WU, Ling-da. A Well-Balanced Time Warp System on Multi-Core Environments. Department of Computer Science , **National University of Defense Technology**, China , 2011.

[COP 96] COPSTEIN, Bernardo; Pereira, Carlos E; Wagner, Flavio R. The object-oriented Apporach and the Event Discrete Simulation Paradigms. In: EUROPEAN SIMULATION MULTICONFERENCE, 1996, Budapest. **Proceedings...** San Diego, SCS, 1996. p.57-61.

[DEAN 2004] DEAN, Jeffrey; Ghemawat, Sanjay; **MapReduce: Simplified Data Processing on Large Clusters.** OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

[FER 2011] FERNÁNDEZ, Javier; NÚÑEZ, Alberto; FILGUEIRA, Rosa; GARCÍA, Félix; CARRETERO, Jesús. SIMCAN: A flexible, scalable and expandable simulation platform for modelling and simulating distributed architectures and applications. **Simulation Modelling Practice and Theory** , 2011

[FER 96] FERSCHA, Alois. Parallel and Distributed Simulation of Discrete Event Systems. In: ZOMAYA, Albert Y. H. **Parallel and Distributed Computing Handbook.** New York: McGraw-Hill, 1996. p. 1003-1041.

[FLY 72] Flynn, M. (1972). "Some Computer Organizations and Their Effectiveness". *IEEE Trans. Comput.* **C-21**: 948.

[FUJ 90a] FUJIMOTO, Richard M. Parallel Discrete Event Simulation. **Communications of the ACM**, New York, 1990, Oct. p. 31-53.

[FUJ 90b] R. M. Fujimoto, “Performance of Time Warp Under Synthetic Work- loads,” in **Proceedings of the SCS Multiconference on Distributed Simulation**, vol. 22, SCS Simulation Series, 1990, pp. 23-28.

[FUJ 92] FUJIMOTO, Richard; NICOL, David. State of Art in Parallel Simulation. In: WINTER SIMULATION CONFERENCE, WSC, 1992, Arlington. **Proceedings...** San Diego: SCS, 1992. 1410p. p. 246-254.

[FUJ 2000] FUJIMOTO, Richard M. **Parallel and Distributed Simulation Systems**. New York: John Wiley & Sons, 2000.

[GEY 98] GEYER, Cláudio Fernando Resin. **Programação Paralela e distribuída**. Porto Alegre: CPGCC/UFRGS, 1998.

[GAMMA 95] GAMMA, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. **Design Patterns: Elements of Reusable Object-Oriented Software**. 1 ed. Estados Unidos da América: Addison-Wesley, 1995.

[GOSLING 2000] GOSLING, J.; Joy, B., Steele, G.; Brachda, G.; **The Java Language Specification,** 2nd Edition. Acesso em 08/03/2011; http://bit.ly/iVjmv.

[HICK 2005] Hickson, Ian; Web Applications 1.0; WHATWG; 2005; http://www.whatwg.org/specs/web-apps/current-work

[HICK 2012] Hickson, Ian; The WebSocket API; W3C; 2012; http://dev.w3.org/html5/websockets

[HWA 87] HWANG, K.; Briggs, F. A. **Computer architecture and parallel processing**. New York: McGraw-Hill, 1987. 846 p.

[IEEE 85] IEEE Standard for Binary Floating Point Arithmetic, **ANSI/IEEE** Std. 754-1985 (IEEE, New York)

[IETF 99] RFC 2616: HTTP/1.1; IETF; 1999; http://www.ietf.org/rfc/rfc2616.txt

[IOS 2004] IOSEVICH, Vadim; SCHUSTERM, Assaf. A comparison of sequential consistency with home-based lazy release consistency for software distributed shared memory. **ICS '04 Proceedings of the 18th annual international conference on Supercomputing**. 2004

[JAL 94] JALOTE, Pnakaj. Recovering a Consistent State. In: **Fault Tolerance in distributed systems**. Englewood Cliffs: Pretice Hall, 1994. p189-215.

[JEF 85] JEFFERSON, D. R. **Virtual Time.** IEEE Transactions on Programming Languages and Systems, v. 7, n. 3, p. 404-425, 1985.

[KUM 94] KUMAR, V. *et al*. Models of Parallel Computers. In: **Introduction to Parallel Computing: Design and Analysis of Algorithms.** Redwood City: The Benjamim, Cummings Publishing Company, 1994. p. 15-64.

[KAL 97] KALANTERY, N. Tentative Time Warp. In: **Proceedings of the 3th International Euro-Par Conference**. In Parallel and Distributed Algorithms, Lecture Notes in Computer Science, v. 1300, p. 458-467, 1997.

[KSH 2008] KSHEMKALYANI, A. D. Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press; 1 edition (May 19, 2008). p. 410-452.

[LAM 78] LAMPORT, L. **Time, clocks, and the ordering of events in a distributed system**. Commun. ACM 21,7 (July 1978), 558-565.

[LON 2010] LONG, Qingqi; LIN, Jie; SUN, Zhixun. Agent scheduling model for adaptive dynamic load balancing in agent-based distributed simulations. **Simulation Modelling Practice and Theory** , 2010.

[LU 95] LU, Honghui *et al*. Message Passing Versus Distributed Shared Memory on Networks of Workstations. In: THE INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS, 1995, San Diego. **Proceedings...** New York: ACM Press and IEEE Computer Society Press, 1995.

[LU 97] LU, Honghui *et al*. Quantifying the Performance Differences Between PVM and TreadMarks. **Journal of Parallel and Distributed Computation**, Orlando, p. 65-78, Jun. 1997

[LUM 97] LUMPP, James E. **Checkpointing and Rollback for Distributed Applications**. 1997. Acesso em: <http://www.dcs.uky.edu/~jeldemo/chkpt.html>, 05 jul. 2009 .

[MAC 87] M. H. MacDougall. **Simulation computer systems - techniques and tools**. The MIT Press, 1987.

[MAC 2006] MACÊDO, R. J. de A. . Computação Distribuída. In: Willian Giozza, Horácio

Hastenreiter, Adhvan Furtado, Péricles Magalhães Jr.. (Org.). **Relatório de Tendências em Tecnologia da Informação e Comunicação.** 1 ed. Salvador: Secretaria de Ciência, Tecnologia e Inovação - Bahia, 2006, v. 1, p. 157-169.

[MER 2010] MERAJI, Sina; ZHANG Wei; TROPPER , Carl. A Multi-State Q-learning Approach for the Dynamic Load Balancing of Time Warp . McGill University, Canada, 2010.

[MER 2011] MERAJI, S.; TROPPER, C.; , Optimization Techniques for Parallel Digital Logic Simulation, **Parallel and Distributed Systems**, IEEE Transactions. 2011

[MIS 86] MISRA, J. Distributed discrete-event simulation. ACM Computing Surveys, 1986.

[MOR 2009] Mordani, Rajiv; JSR-000315 JavaServlet 3.0; JCP, 2009; http://jcp.org/aboutJava/communityprocess/final/jsr315/index.html

[MUK 97] MUKHERJEE, Shubhendu S. Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulato. In PAID, 1997. **Proceedings …** Disponível em: <http://cs.wisc.edu/~shubu/papers.html#survey>. Acesso em: 28 nov. 2010.

[MUS 99] MUSSI, Philippe. Parallel Discrete Event Simulation. In: INTERNATIONAL SCHOOL ON ADVANCED ALGORITHMIC TECHNIQUES FOR PARALLEL COMPUTATION WITH APPLICATIONS, 1999, Natal. [**Papers**]

[OVE 91] OVEREINDER, Benno; HERTZBERGUER, Bob; SLOOT, Peter. Parallel Discrete Event Simulation. In: THE WORKSHOP COMPUTERSYSTEM, 3., 1991, Eindhover. **Proceedings...** [S.1.]: W J. Withagen, 19991. p 19-30.

[PHA 99a] PHAM, CongDuc. High Performance clusters: A promising environment for parallel discreve event simulation. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, PDPTA, 1999, Las Vegas. **Proceedings...** Las Vegas:[s.n], 1999. Disponível em <http://resam.univ-lyon1.fr/PUBLICATIONS/> , 08 ago.

2008.

[POR 99] PORRAS, Jari; IKONEN, Jouni. Aproaches to the analysis of Distributed Simulation. In: EUROPEAN SIMULATION SYMPOSIUM, 11., 1999, Erlangen. **Proceedings...** Ghent: SCS, ASIM, 1999. p. 551-555.

[POR 2005] PORFIRIO, R. **ETW: Um núcleo para simulação distribuída otimista**. Tese de Mestrado, Centro de Ciências Exatas e Tecnologia – UFMS, 2005.

[PRO 96] PROTIC, Jelica. Distributed Shared Memory: Concepts and Systems, University of Belgrade, IEEE Parallel & Distributed Technology, 1996.

[POL 88] POYCHRONOPOULOS, C.D. **Parallel programming and compilers**. Kluwer Academic Publishers, 1988.

[RAS 2010] RASMUSSEN, Morten Sleth. Support for Programming Models in Network-on-Chip-based Many-core Systems. Kongens Lyngby. 2010 .

[REB 2000] REBONATTO, Marcelo Trindade. **Simulação paralela de eventos discretos com uso de memória compartilhada distribuída.** Porto Alegre: PPGC/UFRGS, 2000. Dissertação de mestrado.

[REIH 90] REIHER, P. L.; FUJIMOTO, R.; BELLENOT, S.; JEFFERSON, David. Cancellation Strategies in Optimistic Execution Systems. In: Proceedings of the 1990 Distributed Simulation Conference, p. 112-121, 1990.

[SAN 94] SANTANA, R. H. C.; SANTANA, M. J.; ORLANDI, R. S., SPOLON, R.; Calônego, N. **Técnicas para avaliação de desempenho de sistemas computacionais.** Notas Didáticas - ICMSC – USP, 1994.

[SEI 98] SEIDEL, Criistiana Bendes. **A técnica Lock Acquirer Prediction e sua aplicação em Sistemas de Memória Compartilhada Distribuída**. Rio de Janeiro: COOPE/UFRJ, 1998. Tese de Doutorado

[SHA 92] SHANNON, Robert E. Introduction to Simulation. In: WINTER SIMULATION CONFERENCE, 1992, Arlington. **Proceedings...** San Diego: SCS, 1992. p. 65-73

[SIL 98] SILVA, Ermes Medeiros da et al. Simulação. In: **Pesquisa Operacional**. São

Paulo: Atlas, 1988. p. 143-155.

[SIL 99] SILVA, Márcio Gonçalves da. **Influência de parâmetros arquiteturais em sistemas paralelos de programação em lógica.** Rio de Janeiro: COPPE/UJRJ, 1999. Dissertação de Mestrado.

[SOA 92] Soares, L. F. G. **Modelagem e simulação discreta de sistemas**. Editora Campus LTDA, 1992

[SPO 01] SPOLON, R. **Um método para avaliação de desempenho de protocolos de sincronização otimistas para simulação distribuída**. Tese de Doutorado, Instituto de Física de São Carlos – USP, 2001.

[TAN 2002] TANENBAUM, A. S. Distributed Systems: Principles and Paradigms. Prentice-Hall, 2002. p. 244-251.

[TER 2008a] **Technical Introduction to Terracotta: open source network-attached memory for JVM high-availability and scalability**, 2008. Disponível em <www.terracotta.org>, 02 ago 2009.

[TER 2008b] Inc. Terracotta. **The Definitive Guide to Terracotta: Cluster the JVM for Spring, Hibernate and POJO Scalability**. Apress, 2008. p. 100-102

[TIB 96] TIBAUT, Andrej. **Parallel traffic simulation with an algorithm for dynamic load-balancing.** Nathan, Australia: School of Computing and Information Technology (CIT), Griffith University; Maribor, Solvenia: Faculty of Civil Engineering, 1996. Research Report Number CIT-96-02. Disponível em <http://www.cit.gu.edu.au/research/reports/postscript/CIT-96-02.ps>, em 04 jul. 2009.

[THU 2010] THULASIDASAN, Sunil; KASIVISWANATHAN, Shiva Prasad; EIDENBENZ, Stephan; ROMERO, Phillip; Explicit Spatial Scattering for Load Balancing in Conservatively Synchronized Parallel Discrete Event Simulations. **24th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation .**

[VAL 2008a] VALCHANOV, Hristo; RUSKOVA, Nadezhda; GENOV, Dimitar; NIKOLOV , Nedjalko; Partitioning Parallel Discrete Event Simulation ,

**International Conference on Computer Systems and Technologies.** 2008.

[VAL 2008b] VALCHANOV, Hristo; RUSKOVA, Nadezhda; RUSKOV , Trifon; Overheads Reduction of the Distributed Time Warp Simulation . **International Conference on Computer Systems and Technologies** , 2008.

# Apêndice A: Trechos de implementação da plataforma Darfia

## A.1 DarfiaLogicalProcess : Run()

O código abaixo apresenta o mecanismo de consumo da fila de eventos futuros de um processo lógico, que é executado em cada processo lógico da simulação.

```
public boolean run() throws InterruptedException
{
        DarfiaEvent evt = null;

        while
    (
    this.futureEventsQueue.size() > 0 &&
        (evt = this.getFutureEventsQueue().deQueue()) != null &&
        evt.getRealTime() <= this.getSimulationEndTime()
)
        {
            double eventTime = evt.getRealTime();

            if (this.getLVT() == 0 || this.getLVT() != eventTime) {
                getLogger().logInfo("", 1);
                getLogger().logInfo("  event time:" + evt.toString(), 1);
            }

            // gera nova entrada
            if (evt.getSubject() == DarfiaEventSubject.CUSTOMER_ARRIVAL &&
            this.getArrivalProducer() != null &&
            !this.getArrivalProducer().hasReachedLimitNumber()) {
                this.produceArrivalAndDeparture(eventTime);
            }

            // coloca para processar
            this.setLVT(this.processEvent(evt));

            // verifica se outra thread colocou uma notificação de rollback
            if (this.hasRollbackNotification())
        this.threatRollbackNotification();

      // verifica se outra thread colocou uma nova chegada de cliente nesse
    // processo antes de continuar.
    if (this.hasChainedArrival())
        this.threatChainedArrival();

    // atualiza lvt desse lp na simulação
    this.getPresentSimulation().updateStatusLogicalProcessLVT(this.getId(), this.getLVT());
```

```
            }
            // verifica ainda se sobrou alguém
        if (this.getLVT() >= this.getSimulationEndTime() ||
           (evt != null && evt.getRealTime() >= this.getSimulationEndTime())) {
            // antes de sair, verifica se não há nada em eventos encadeados.
            getLogger().logInfo("Simulation Done.", 0);
            return true;
        } else if (this.futureEventsQueue.size() == 0 && this.hasChainedArrival()) {
            this.threatChainedArrival();
            return false;
        } else if (
            this.canLogicalProcessReceiveChainedEvent() &&
            this.stillReceivingEvents()
        ) {
            // verifica o que fazer quando não há clientes.
            return false;
        }

        return true;
        }
```

## A.2 DarfiaLogicalProcess : processEvent(DarfiaEvent e)

O código abaixo demonstra o processamento de um evento dentro do processo lógico, após o seu consumo da lista de eventos futuros.

```
        public double processEvent(DarfiaEvent evt) {

        double currentEvtTime = evt.getRealTime();

       // analise o tipo de evento
       switch (evt.getSubject())
       {
           // evento de chegada de cliente
           case DarfiaEventSubject.CUSTOMER_ARRIVAL:
           {
               if (currentEvtTime < this.getSimulationEndTime()) {

                   // remove próximo da fila do recurso e aloca!
                   if (!this.isBusy())
                   {
                       // não há fila para o prestador de serviço.
                       getLogger().logInfo("    alocating service center until " +
evt.getRelatedEvent().getRealTime(), 1);
                       this.alocateServiceQueue(evt);

                       // marca como processado
                       this.processedEventsQueue.add(evt);
                   }
                   else
                   {
                       // prestador de serviço está ocupado. enfileire evento que chegou
                       this.serviceCenterQueue.queue(evt);

                       // guarda a evidência do tamanho da fila quando esse foi alocado.
                       getLogger().logInfo("    scheduling consumer. scheduled:" +
```

```
this.getServiceCenterQueue().size(), 1);
                        this.getProcessStatistic().addServiceCenterQueueSize(this.getServiceCenterQueue().size());
                        this.getProcessStatistic().incrementQueuesNumber();

                        // tempo médio de fila, depois que já realizou as
                        this.getProcessStatistic().addMeanServiceCenterQueueLength(
                            currentEvtTime,
                            this.serviceCenterQueue.size()
                        );

                        // remove evento de retirada da fila.
                        this.futureEventsQueue.remove(evt.getRelatedEvent());
                    }
                }
                break;
            }

            // evento de saída de cliente
            case DarfiaEventSubject.CUSTOMER_DEPARTURE:
            {
                // verifica se o evento de saída corresponde ao cliente alocado.
                if (this.getPresentProcessingEvent() == null ||
                    this.getPresentProcessingEvent().getRelatedEvent() != evt) {
                    break;
                }

                // libera prestador de serviço
                this.dealocateServiceQueue(evt);
                this.processedEventsQueue.add(evt);

                // guarda dados
                getLogger().logInfo("    release server", 1);
                this.getProcessStatistic().incrementReleasesNumber();

                // verifica se prestador de serviço tem alguém na fila
                if (this.serviceCenterQueue.size() > 0 && !this.isBusy())
                {
                    DarfiaEvent queuedArrivalEvent = this.serviceCenterQueue.deQueue();
                    DarfiaEvent queuedDepartureEvent = queuedArrivalEvent.getRelatedEvent();

                    // tempo médio de fila, depois que já realizou as
                    this.getProcessStatistic().addMeanServiceCenterQueueLength(
                        currentEvtTime,
                        this.getServiceCenterQueue().size()
                    );

                    this.scheduleArrivalAndDeparture(queuedArrivalEvent, queuedDepartureEvent, currentEvtTime);
                    getLogger().logInfo("    pipe out consumer. " + this.serviceCenterQueue.size(), 1);

                    DarfiaEvent rescheduledEvt = this.futureEventsQueue.deQueue();
                    if (rescheduledEvt.getRealTime() <= this.getSimulationEndTime())
                        this.processEvent(rescheduledEvt);
                }

                // verifica encadeamento de clientes
                if (this.canChainEvents() && this.chaining.mustChainEvent()) {
                    this.chainEvent(evt, this.chaining.randomTarget());
```

```
        }

        break;
    }
}

    return currentEvtTime;
}
```

## A.3 DarfiaBaseQueue

Classe responsável pela implementação da generalização de filas duplamente encadeadas.

```
public abstract class DafiaBaseQueue<QueueContent> {

    private ArrayList<QueueContent> queue = new ArrayList<QueueContent>();

    @Override
    public String toString() {
        String s = "";

        s += "[";
        for (int i = 0; i < this.getQueue().size(); i++) {
            s += (this.getQueue().get(i).toString());

            if (i + 1 < this.getQueue().size())
                s += ",";
        }
        s += "]";

        return s;
    }

    public synchronized void add(int i, QueueContent c)  {
        this.getQueue().add(i, c);
    }

    public synchronized void add(QueueContent c)  {
        this.getQueue().add(c);
    }

    public QueueContent get(int i ) {
        return this.getQueue().get(i);
    }

    public synchronized void remove(QueueContent c) {
        this.getQueue().remove(c);
    }

    public synchronized void remove(int i) {
        this.getQueue().remove(i);
    }

    public boolean isEmpty() {
        return this.getQueue().size() == 0;
```

```
    }

    public int size() {
        return this.getQueue().size();
    }

    public synchronized void setQueue(ArrayList<QueueContent> queue) {
        this.queue = queue;
    }

    public ArrayList<QueueContent> getQueue() {
        return queue;
    }
}
```

## A.4 DarfiaBaseConcurrentQueue

Classe responsável pela implementação da generalização de filas para serem utilizadas em áreas de concorrência dos processos lógicos. Ou seja, quando acontece a comunicação na abordagem produtor/consumidor.

```
public abstract class DarfiaBaseConcurrentQueue<QueueContent> {

    private LinkedBlockingQueue<QueueContent> queue = new LinkedBlockingQueue<QueueContent>();

    public Iterator<QueueContent> iterator() {
        return this.getQueue().iterator();
    }

    public void remove(QueueContent e) {
        this.getQueue().remove(e);
    }

    public int size() {
        return this.getQueue().size();
    }
    public void queue(QueueContent e) {
        try {
            this.getQueue().put(e);
        }
        catch (InterruptedException e1) {
            e1.printStackTrace();
        }
    }

    public synchronized QueueContent deQueue()
    {
        try {
            return this.getQueue().take();
        }
        catch (InterruptedException e) {
            e.printStackTrace();
            return null;
        }
    }
```

```
    public LinkedBlockingQueue<QueueContent> getQueue() {
        return queue;
    }

    public synchronized void setQueue(LinkedBlockingQueue<QueueContent> queue) {
        this.queue = queue;
    }
}
```

## A.5 DarfiaRndBase

Classe responsável por prover a geração de números aleatórios uniformemente distribuidos, sendo utilizada como base para geração de perfis de probabilidade.

```
public abstract class DarfiaRndBase {

    private static double[] getSeeds() {

        return new double [] {0L,   /* seeds for streams 1 thru 15  */
                  1973272912L,  747177549L,   20464843L,  640830765L, 1098742207L,
                    78126602L,   84743774L,  831312807L,  124667236L, 1172177002L,
                  1124933064L, 1223960546L, 1878892440L, 1449793615L,  553303732L};
    }

    private static int getSelectedSeed() {
        return 1;
    }

        /**
         * Retorna um número aleatório uniformemente distribuído entre 0..1
         * @return
         */
        public static double RRanf() {
            getSeeds()[getSelectedSeed()] = (16807.0 * getSeeds()[getSelectedSeed()]) % 2147483647.0;
                return getSeeds()[getSelectedSeed()] * 4.656612875E-10;
        }

        public abstract double generate();
}
```

## A.6 DarfiaRndExponential

Classe responsável por prover a geração de números aleatórios em distribuição exponencial.

```
public class DarfiaRndExponential extends DarfiaRndBase {

        private double mean;

        public DarfiaRndExponential(double aMean) {
                this.mean = aMean;
        }
        public double generate() {
                return (-mean * Math.log(RRanf()));
        }
```

```
    }
```

## A.7 DarfiaLib : getDarfiaRndFromParsedDistributionStrategy

Implementação responsável por realizar *parsing* nos valores informados para tempo de chegada e tempo de serviço e instanciar o gerador de números aleatórios apropriado, realizando também a passagem de parâmetros.

```
    /**
     * A partir de uma estratégia de distribuição (como as definidas no darfia-config.xml,
     * parseia os parametros e recupera sua geração.
     *
     */
    public static DarfiaRndBase getDarfiaRndFromParsedDistributionStrategy(String distributionStrategy) {
        try
        {
            String[] cpts = distributionStrategy.split(":");
            String strategy = cpts[0];
            String rawParams = cpts[1];

            String parsedParams[];

            if (strategy.equals("fixed")) {
                double value = Double.parseDouble(rawParams);
                return new DarfiaRndFixed(value);
            } else if (strategy.equals("random")) {
                parsedParams = rawParams.split(",");
                int start = Integer.parseInt(parsedParams[0]);
                int end = Integer.parseInt(parsedParams[1]);
                return new DarfiaRndRandom(start, end);
            }
            else if (strategy.equals("normal")) {
                parsedParams = rawParams.split(",");
                double mean = Double.parseDouble(parsedParams[0]);
                double deviation = Double.parseDouble(parsedParams[1]);
                return new DarfiaRndNormal(mean, deviation);
            }
            else if (strategy.equals("uniform")) {
                parsedParams = rawParams.split(",");
                int start = Integer.parseInt(parsedParams[0]);
                int end = Integer.parseInt(parsedParams[1]);
                return new DarfiaRndUniform(start, end);
            }
            else if (strategy.equals("erlang")) {
                parsedParams = rawParams.split(",");
                double mean = Double.parseDouble(parsedParams[0]);
                double deviation = Double.parseDouble(parsedParams[1]);
                return new DarfiaRndErlang(mean, deviation);
            }
            else if (strategy.equals("exponential")) {
                parsedParams = rawParams.split(",");
                double mean = Double.parseDouble(parsedParams[0]);
                return new DarfiaRndExponential(mean);
            }
```

```
        }
        catch(Exception e) {
            e.printStackTrace();
        }

        return null;
    }
```

## A.8 DarfiaLib : produceArrivalAndDeparture

Implementação responsável por realizar a produção do movimento de um cliente no sistema, através da criação do evento de chegada e o evento de partida.

```
    /**
     * Produz uma chegada e saída
     */
    public static DarfiaEvent[] produceArrivalAndDeparture(
            String lpId,
            double arrivalTime,
            double arrivalScheduledTime,
            double departureTime,
            double departureScheduledTime
        ) {

        DarfiaEvent arrivalEvt = DarfiaLib.produceEvent(
            DarfiaEventType.NORMAL,
            DarfiaEventSubject.CUSTOMER_ARRIVAL,
            lpId,
            lpId,
            arrivalTime,
            arrivalScheduledTime
        );

        DarfiaEvent departureEvt = DarfiaLib.produceEvent(
            DarfiaEventType.NORMAL,
            DarfiaEventSubject.CUSTOMER_DEPARTURE,
            lpId,
            lpId,
            departureTime,
            departureScheduledTime
        );

        arrivalEvt.setRelatedEvent(departureEvt);

        return new DarfiaEvent[] {arrivalEvt, departureEvt};
    }
```

## A.8 DarfiaXMLDoc

Implementação responsável por abstrair a extração de informações de documentos XML.

```
    public class DarfiaXMLDoc {
```

```
	private Document doc = null;

	public DarfiaXMLDoc(String xmlPath)
	{
		try
		{
			File file = new File(xmlPath);
			DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
			DocumentBuilder db = dbf.newDocumentBuilder();
			this.doc = db.parse(file);
			this.doc.normalize();
//			printDocument(this.doc, System.out);
		}
		catch(Exception e)
		{
			System.out.println("error opening config file: " + xmlPath);
			return;
		}
	}

	public NodeList getElementsByTagName(String tagname)
	{
		return this.doc.getElementsByTagName(tagname);
	}

	public static ArrayList<Node> getElementsFromNodeByTagName(String tagname, Node n)
	{
		NodeList c = n.getChildNodes();
		ArrayList<Node> nodes = new ArrayList<Node>();

		for (int i = 0; i < c.getLength(); i++)
		{
			Node item = c.item(i);
			String itemLocalName = item.getNodeName();
			if (item != null && itemLocalName != null && item.getNodeName().equals(tagname))
				nodes.add(c.item(i));
		}

		return nodes;
	}

	public static Node getElementFromNodeByTagName(String tagname, Node n)
	{
		NodeList c = n.getChildNodes();

		for (int i = 0; i < c.getLength(); i++)
		{
			Node item = c.item(i);
			String itemLocalName = item.getNodeName();
			if (item != null && itemLocalName != null && item.getNodeName().equals(tagname))
				return c.item(i);
		}

		return null;
	}

	public Node getElementByTagName(String tagname)
```

```
        {
                NodeList children = this.doc.getElementsByTagName(tagname);
                return children.item(0);
        }

        public static String getNodeAttributeValue(Node n, String attrName)
        {
                NamedNodeMap attrs = n.getAttributes();
                for (int i = 0; i < attrs.getLength(); i++)
                {
                        Attr attr = (Attr) attrs.item(i);
                        if (attr.getName().equals(attrName))
                                return attr.getValue();
                }

                return null;
        }

        public static void printDocument(Document doc, OutputStream out) throws IOException,
TransformerException     {
            TransformerFactory tf = TransformerFactory.newInstance();
            Transformer transformer = tf.newTransformer();
            transformer.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "no");
            transformer.setOutputProperty(OutputKeys.METHOD, "xml");
            transformer.setOutputProperty(OutputKeys.INDENT, "yes");
            transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
            transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "4");

            transformer.transform(new DOMSource(doc),
                 new StreamResult(new OutputStreamWriter(out, "UTF-8")));
        }
}
```

## A.8 DarfiaLogicalProcess : checkForRollbackExplosion

Implementação do procedimento de *rollback,* utilizando o cancelamento agressivo.

```
 /**
     * Checa pela ocorrencia encadeada de rollbacks
     *
     * @param lateArrivalEvent
     * @param lateDepartureEvent
     */
    public ArrayList<DarfiaEvent> checkForRollbackExplosion(DarfiaEvent inconsistentAntiEvt) {

        getLogger().logInfo(
            "    rollback. inconsistent evt " + inconsistentAntiEvt.getRealTime() +
            " current LVT: " + this.getLVT(),
            1
        );

        Date startTime = new Date();

        // enumera todos os eventos que estão inconsistentes em relação ao LVT
        ArrayList<DarfiaEvent> inconsistentEvents = new ArrayList<DarfiaEvent>();
```

```
        DarfiaServiceCenterQueue scq = this.getServiceCenterQueue();
        DarfiaProcessedEventsQueue peq = this.getProcessedEventsQueue();

        DarfiaAntiQueue outputQueue = this.getOutputQueue();

        // verifica se está na fila do centro de serviço
        for (int i = 0; i < scq.size(); i++)
            if (scq.get(i).getRealTime() >= inconsistentAntiEvt.getRealTime()) {
                inconsistentEvents.add(scq.get(i));
                inconsistentEvents.add(scq.get(i).getRelatedEvent());
                scq.remove(i);
            }

        // verifica se está na fila de processados
        for (int i = 0; i < peq.size(); i++)
            if (peq.get(i).getRealTime() >= inconsistentAntiEvt.getRealTime()) {
                inconsistentEvents.add(peq.get(i));
                inconsistentEvents.add(peq.get(i).getRelatedEvent());
                peq.remove(i);
            }

        // tira o evento do centro de serviço.
        if (this.presentProcessingEvent != null &&
            this.presentProcessingEvent.getRealTime() > inconsistentAntiEvt.getRealTime()) {
            inconsistentEvents.add(this.presentProcessingEvent);
            this.setPresentProcessingEvent(null);
        }

        // guarda o processo de rollback para estatisticas
        this.getProcessStatistic().addPrimaryRollbackLength(inconsistentEvents.size());

        String inconsistentEventsStr = "";
        for (int i = 0; i < inconsistentEvents.size(); i++) {
            inconsistentEventsStr += inconsistentEvents.get(i).toString();

            if (i+1 < inconsistentEvents.size() + 1)
                inconsistentEventsStr += ", ";
        }

        getLogger().logInfo(
            "    inconsistent events: " + inconsistentEventsStr + " old fel: " +
this.futureEventsQueue.toString(),
            1
        );
        // verifica se algum desses eventos que foram marcados como inconsistentes
        // gerou alguma antimensagem.
        // com certeza o encadeamento ainda não foi processado
        if (inconsistentEvents.size() == 0) {

            DarfiaChainingQueue cq = this.getChainingQueue();

            // verifica se está na fila de encadeados
            Iterator<DarfiaChainedData> it = cq.iterator();
            while (it.hasNext()) {
                DarfiaChainedData dcd = it.next();
                if (dcd.getAntiEvt().getRealTime() > inconsistentAntiEvt.getRealTime()) {
                    cq.remove(dcd);
```

```
                getLogger().logInfo(
                    "    this chained event was never processed! :)",
                    1
                );
                break;
            }
        }
    }
    else
    {
        for (int i = 0; i < inconsistentEvents.size(); i++)
        {
            DarfiaEvent inconsistentEvent = inconsistentEvents.get(i);

            // somente mensagens de saída geram antimensagens
            if (inconsistentEvent.getSubject() != DarfiaEventSubject.CUSTOMER_DEPARTURE)
                continue;

            for (int j = 0; j < outputQueue.size(); j++)
            {
                if (inconsistentEvent.getRelatedEvent() == outputQueue.get(j))
                {
                    DarfiaEvent propagatedInconsistentAntiEvent = outputQueue.get(j);

                    // ooooooops, tenho uma situação em que um evento que foi marcado
                    // como inconsistente devido o rollback gerou um evento de
                    // saída (antimensagem). Peça para a simulação ir até o processo
                    // destino e peça para esse iniciar o procedimento de
                    // checagem de rollback partir do evento que foi enviado.

                    this.getPresentSimulation().askLPToCheckForRollback(
                        propagatedInconsistentAntiEvent.getlPReceiverId(),
                        propagatedInconsistentAntiEvent // anti message
                    );
                }
            }
        }
    }

    // agrupa tempo gasto
    this.getProcessStatistic().incrementTotalPrimaryRollbackMiliseconds(
        new Date(new Date().getTime() - startTime.getTime()).getTime()
    );

    return inconsistentEvents;
}
```

# Apêndice B: Modelos Hipotéticos implementados em SMPL

## B.1 Modelo Hipotético 1

Implementação do modelo hipotético 1 em SMPL

```
#include "smpl.h"
main()
  {
    real Ta=200.0,Ts1=100.0,te=200000.0, Ts2=???;
    int customer=1,event,server1, server2;
    smpl(0,"M/M/1 Queue");
    server1=facility("server",1);
    server2=facility("server",1);
    schedule(1,0.0,customer);
    while (time()<te)
      {
        cause(&event,&customer);
        switch(event)
          {
            case 1:  /* arrival */
              schedule(2,0.0,customer);
              schedule(1,expntl(Ta),customer);
              break;
            case 2:  /* request server 1 */
              if (request(server1,customer,0)==0) then
                schedule(3,expntl(Ts1),customer);
              break;
            case 3:  /* libera server1, escalona evento para solicitar server2 */
              release(server1,customer,0);
              schedule(4,0.0,customer);
             break
            case4: /* solicita server2*/
              if (request(server2,customer,0)==0) then
                schedule(5,expntl(Ts2),customer);
              break;
            case 5:  /* release server2 */
              release(server2,customer);
              break;
          }
      }
    report();
  }
```

## B.2 Modelo Hipotético 2

Implementação do modelo hipotético 2 em SMPL

```
#include "smpl.h"
main()
  {
    real Ta=200.0,Ts1=100.0,te=200000.0, Ts2=???;
    int customer=1,event,server1, server2, r;
    smpl(0,"M/M/1 Queue");
    server1=facility("server",1);
    server2=facility("server",1);
    schedule(1,0.0,customer);
    while (time()<te)
      {
        cause(&event,&customer);
        switch(event)
          {
            case 1:  /* arrival */
              schedule(2,0.0,customer);
              schedule(1,expntl(Ta),customer);
              break;
            case 2:  /* request server 1 */
              if (request(server1,customer,0)==0) then
                schedule(3,expntl(Ts1),customer);
              break;
             case 3:  /* libera server1, escalona evento para solicitar server2 com probabilidade 50%, e
outros 50% vai embora */
              release(server1,customer,0);
              r = random(1,10);
              if (r>5)
                schedule(4,0.0,customer);
             break;
            case4: /* solicita server2*/
              if (request(server2,customer,0)==0) then
                schedule(5,expntl(Ts2),customer);
              break;
            case 5:  /* release server2 */
              release(server2,customer);
              break;
          }
      }
    report();
  }
```

## B.3 Modelo Hipotético 3

Implementação do modelo hipotético 3 em SMPL

```
#include "smpl.h"
main()
  {
    real Ta=200.0,Ts1=100.0,te=200000.0, Ts2=???; Ts3=???;
    int customer=1,event,server1, server2, server3, r;
    smpl(0,"M/M/1 Queue");
```

```
    server1=facility("server",1);
    server2=facility("server",1);
    server3=facility("server",1);
    schedule(1,0.0,customer);
    while (time()<te)
      {
        cause(&event,&customer);
        switch(event)
          {
            case 1:  /* arrival */
              schedule(2,0.0,customer);
              schedule(1,expntl(Ta),customer);
              break;
            case 2:  /* request server 1 */
              if (request(server1,customer,0)==0) then
                schedule(3,expntl(Ts1),customer);
              break;
             case 3:  /* libera server1, escalona evento para solicitar server2 com probabilidade 50% ou
server3 com probabilidade 50% */
              release(server1,customer,0);
              r = random(1,10);
              if (r>5)
                schedule(4,0.0,customer);
              else
                schedule(6,0.0,costomer);
             break;
            case4: /* solicita server2*/
              if (request(server2,customer,0)==0) then
                schedule(5,expntl(Ts2),customer);
              break;
            case 5:  /* release server2 */
              release(server2,customer);
              break;
            case6: /* solicita server3*/
              if (request(server3,customer,0)==0) then
                schedule(7,expntl(Ts3),customer);
              break;
            case 7:  /* release server3 */
              release(server3,customer);
              break;
          }
      }
    report();
  }
```