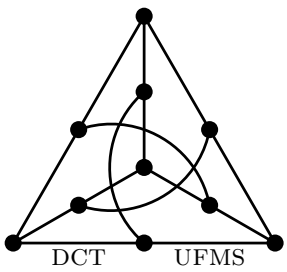
ETW: Um Núcleo para Simulação Distribuída Otimista

Rodrigo Porfírio da Silva Sacchi

Orientação: Prof[a]. Dr[a]. Renata Spolon Lobato

Área de Concentração: Simulação Distribuída

Dissertação apresentada ao Departamento de Computação e Estatística da Universidade Federal de Mato Grosso do Sul, como parte dos requisitos para obtenção do título de **Mestre em Ciência da Computação**.



Departamento de Computação e Estatística
Centro de Ciências Exatas e Tecnologia
Universidade Federal de Mato Grosso do Sul
Fevereiro/2005

Aos meus pais por todos
os seus conselhos e amor
a mim dedicados e a meus
irmãos, por sempre acreditarem em mim.

Agradecimentos

À minha orientadora, Prof^a. Dr^a. Renata Spolon Lobato, pela paciência, dedicação, confiança, apoio e principalmente por tornar possível a conclusão deste trabalho.

Aos meus pais pelo amor e ótima educação dada, além do apoio constante e compreensão.

Aos meus irmãos, Leonardo e Gustavo, que sempre deram forças nos momentos de maior desânimo.

Agradeço também à Flávia que, nas horas mais difíceis, confortou-me e deu-me forças para continuar esta jornada.

Às amigas Bianca e Marta, pelo apoio e importantes ajudas durante o desenvolvimento deste trabalho.

À Prof^a. Dr^a. Roberta Spolon Ulson, pelo auxílio com a simulação distribuída. e paciência em algumas correções desta dissertação.

Ao amigo Daniel Lobato pelas aulas dadas em períodos em que precisei me ausentar.

Ao amigo Carlos Juliano, pelas ajudas, principalmente com a ferramenta gráfica *GNUPLOT*.

Aos amigos Luciana, Delair, Jona, Carlos Alexandre e Aline, pela amizade e confiança depositada em mim durante esses anos.

Aos amigos do Residencial Verdes Matas, pelas festas churrascos na sacada, tereré e ótimas gargalhadas.

Ao Prof. Dr. Henrique Mongelli pelas inúmeras dúvidas solucionadas com a biblioteca LAM-MPI.

Ao Prof. Dr. Paulo Aristarco Pagliosa pelas constantes dicas de programação orientada a objetos.

A todos aqueles que contribuíram para a realização deste trabalho, direta ou indiretamente.

À CAPES, pelo apoio financeiro.

E à Deus, por guiar-me nessa longa jornada e proteção durante todo o tempo.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Siglas e Abreviaturas

APOSTLE	-	*A parallel object-oriented simulation language*
ASDA	-	Ambiente de Simulação Distribuída Automático
ASiA	-	Ambiente de Simulação Automático
bps	-	bits por segundo
CMB	-	Chandy, Misra e Bryant
CSS	-	*Copy State Saving*
DCT	-	Departamento de Computação e Estatística
ETW	-	*Basic Extensible Time Warp Kernel*
FEL	-	*Future Event List*
FIFO	-	*First-Come First-Served*
ICMC	-	Instituto de Ciências Matemáticas e de Computação
ISS	-	*Incremental State Saving*
LCG	-	*Linear Congruential Generator*
LEF	-	Lista de Eventos Futuros
LVT	-	*Local Virtual Time*
GTW	-	*Georgia Tech Time Warp*
GVT	-	*Global Virtual Time*
Moose	-	*Maisie-based object-oriented simulation environment*
JTL	-	*Jet Propulsion Laboratory*
Mbps	-	Mega bits por segundo
MPI	-	*Message Passing Interface*
MRIP	-	*Multiple Replication In Parallel*
Parsec	-	*Parallel simulation environment for complex systems*
PL	-	Processo Lógico
SMPL	-	*Simulation Language Program*
SRIP	-	*Single Replication In Parallel*
SSS	-	*Sparse State Saving*
TWOS	-	*Time Warp Operating System*
UCP	-	Unidade Central de Processamento
UFMS	-	Universidade Federal de Mato Grosso do Sul
USP	-	Universidade de São Paulo

Resumo

Este trabalho apresenta um núcleo para simulação distribuída otimista com base no protocolo *Time Warp*, denominado *Basic Extensible Time Warp Kernel* (`ETW`), o qual tem como objetivo permitir a avaliação de desempenho de sistemas discretos de uma forma geral, e em especial os computacionais, através da solução de modelos de redes de filas. A adição de *plugins* permite alterações e/ou extensões da funcionalidade do núcleo. Um dos aspectos da implementação do `ETW` merece atenção: o uso de programação orientada a objetos e, ao mesmo tempo, o uso de programação estruturada. Isso permite explorar amplamente a programação orientada a objetos para estender o `ETW` através da mudança de classes responsáveis, por exemplo, pelo cálculo do *GVT* (*Global Virtual Time*) ou salvamento de estados, ao mesmo tempo em que permite o uso de conhecimento prévio do modelador sobre a extensão funcional *SMPL* (*Simulation Programming Language*). Cada processo da simulação com o `ETW` implementa um processo lógico da simulação distribuída *Time Warp*. A estrutura da simulação seqüencial executada em cada processo da simulação distribuída segue a abordagem de orientação a eventos, com base na extensão funcional *SMPL*. Os processos interagem entre si através da troca de mensagens feita com as primitivas de comunicação do *LAM-MPI* (*Message Passing Interface*). Foram feitos estudos com modelos de redes de filas, que permitiram identificar uma possível métrica que poderá auxiliar na troca dinâmica de protocolos (o comprimento médio de *rollbacks* primários).

Abstract

This work presents the Basic Extensible Time Warp Kernel (ETW), a kernel for optimistic distributed simulation based on the Time Warp protocol, which aims to allow the performance evaluation of discrete systems, especially computational systems, through the solution of models of queue nets. Some kernel's features can be modified by the use of plugins. One of the implementation aspects of ETW deserve attention: the usage of object oriented programming and, at the same time, the usage of structured programming by the simulation developer. This allow the full power of object oriented programming for ETW extension by changing classes responsible for, e.g., GVT calculation or state saving, at the same time that allows the use of previous knowledge on SMPL that the simulation developer might have. Each simulation process on ETW implements one logical process on Time Warp distributed simulation. Each sequential simulation is event-driven and uses SMPL to manage the local queues. The LAM-MPI communication library is used for message exchanging among logical processes. We have conducted some test cases on queue nets models aiming to identify metrics that could suggest dynamically protocol changes (for instance, the size of primary rollbacks).

Capítulo 1

Introdução

1.1 Motivação

A avaliação de desempenho tem se tornado uma área de interesse, principalmente quando se trata do estudo de sistemas computacionais. A avaliação de desempenho pode determinar o grau de qualidade de um sistema, além de poder ser utilizada na seleção de sistemas alternativos, no projeto de novos sistemas e na análise comparativa de sistemas existentes.

As técnicas para a avaliação de desempenho de sistemas computacionais englobam as técnicas de aferição, através de *benchmarks*, construção de protótipos e coleta de dados, e as técnicas de modelagem, nas quais a solução analítica ou a solução por simulação podem ser utilizadas para resolver um modelo. As técnicas de aferição analisam o desempenho de um sistema através de sua experimentação, enquanto que as técnicas de modelagem consistem em abstrair as características de um sistema em um modelo e estudar e avaliar o comportamento desse modelo.

Avaliar o comportamento de um sistema real através de sua experimentação pode ser difícil ou impossível. É mais fácil e, às vezes, prudente, alterar o programa de simulação para medir seu comportamento do que alterar todo o sistema real. Assim, ao invés de avaliar o sistema real, pode-se construir um modelo que represente o sistema e resolver tal modelo utilizando a solução analítica ou por simulação.

A solução analítica pode oferecer resultados simples mas, em alguns casos, para modelos mais complexos, sua utilização pode tornar-se limitada e muitas vezes inviável. A simulação constitui uma ferramenta poderosa utilizada em diversas áreas, devido principalmente à sua grande flexibilidade e ao seu baixo custo [Spo94].

O avanço da complexidade de sistemas computacionais trouxe um aumento significativo no tempo de execução da simulação seqüencial, tornando seu tempo de processamento muitas vezes inviável, principalmente para a obtenção de uma grande quantidade de resultados. Esse avanço motivou a adoção da simulação distribuída, a qual tem como principal objetivo reduzir o tempo de processamento de um programa de simulação.

A pesquisa na área de simulação distribuída vem crescendo nos últimos vinte anos, principalmente com o surgimento de novas áreas de aplicação. Pode-se citar, como exemplo, controle de tráfego aéreo, treinamentos militares promovidos através de simulação, simuladores

de vôos, sistemas de transporte, simulação em circuitos lógicos digitais, simuladores de jogos de guerra [Oza01, Zho04], simulação de redes de comunicação e de computadores [Hus04], além de pesquisas em laboratórios industriais e em universidades por todo o mundo. Em conseqüência, surgiram protocolos para garantir a sincronização entre os processos da simulação distribuída e para permitir sua execução de maneira correta. Esses protocolos são divididos em conservativos e otimistas [Fuj00].

Os protocolos conservativos evitam a possibilidade de ocorrer um erro de causa e efeito, ou seja, verificam se nenhum evento ocorrerá fora da ordem das marcas de tempo dos eventos. Entre os protocolos conservativos o principal é o *CMB*, desenvolvido por Chandy, Misra [Cha79] e Bryant [Bry77].

Os protocolos otimistas utilizam um mecanismo de detecção e recuperação de erros de causa e efeito, isto é, os eventos são processados sem nenhuma verificação. Quando um erro ocorre, um mecanismo de *rollback* é necessário para restaurar a simulação para um estado consistente. A simulação continua a execução a partir desse estado recuperado. O principal protocolo otimista é o *Time Warp*, baseado no paradigma de *Virtual Time* [Fuj90, Jef85].

Os principais sistemas que implementam o protocolo *Time Warp*, descritos na literatura, são o *Warped* - para memória distribuída [Wil96, Wil03], e *GTW* (*GeorgiaTech Time Warp*) - para arquiteturas de memória compartilhada [Fuj00].

Além dos protocolos conservativos e otimistas, existem os protocolos híbridos, isto é, protocolos que possuem características de ambos os protocolos, conservativos e otimistas. Exemplos de protocolos conservativos com características otimistas incluem *Breathing Time Buckets* [Ste91, Ste92], *SRADS* [Dic91] e *Conservative Time Windows* [Aya92] e exemplos de protocolos otimistas com características conservativas incluem *Local Time Warp* [Raj93] e *Tentative Time Warp* [Kal97]. Esses últimos são denominados variantes do protocolo *Time Warp*.

1.2 Objetivos

O objetivo deste trabalho consiste no desenvolvimento e avaliação de um núcleo para simulação distribuída *SRIP* (*Single Replication In Parallel*), mais especificamente simulação distribuída otimista *Time Warp* para efetuar a avaliação de desempenho de sistemas computacionais (através da simulação de modelos de redes de filas) [Jef85]. Esse núcleo, denominado `Basic Extensible Time Warp Kernel - ETW`, pode ter sua funcionalidade estendida, através de *plugins* que, quando anexados ao núcleo básico, estendem/modificam seu comportamento, representando protocolos variantes do *Time Warp*, como ilustrado na Figura 1.1 [Spo01].

Protocolos variantes do *Time Warp* também poderão ser considerados. Essas variantes do *Time Warp* básico podem ser inseridas através do uso de *plugins* a partir de pontos de conexão, de forma a alterar seu comportamento. Desse modo, um usuário pode escolher entre utilizar o *Time Warp* básico ou uma variante para executar a simulação, ou seja, com a inserção desses, é possível efetuar comparações entre variantes e o modelo básico do *Time Warp*. Essa independência do usuário pode permitir, futuramente, a identificação de classes de aplicação para o *Time Warp* e suas variantes.

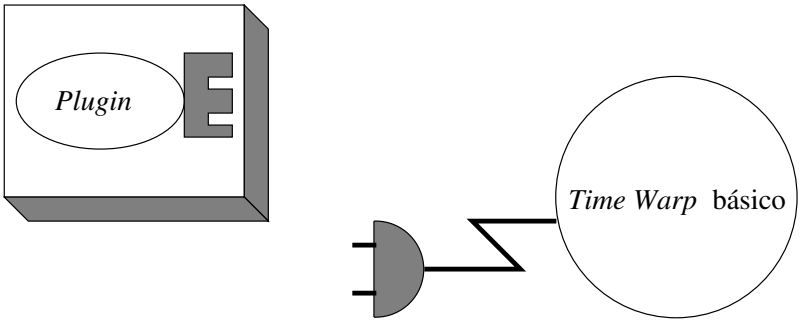


Figura 1.1: Representação do *Time Warp* básico e *plugin*.

Com a construção do núcleo, será possível também efetuar comparações entre o *Time Warp* e outros protocolos de simulação distribuída, como por exemplo o *CMB*, e comparações entre protocolos de simulação distribuída e programas de simulação seqüencial. O `ETW` foi implementado com base na especificação desenvolvida para representar os algoritmos que definem as atividades executadas por um processo lógico *Time Warp*, seus parâmetros e variantes, apresentada em [Spo01]. Estes algoritmos, descritos nas seções posteriores, envolvem, a recepção e execução de mensagens e antimensagens, o processo de salvamento de estados, a tarefa de cancelamento de mensagens e o mecanismo de *rollback*.

O `ETW` também será integrado, posteriormente, a um ambiente completo de simulação, denominado ASDA (Ambiente de Simulação Distribuída Automático), o qual está em desenvolvimento pelo Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo (ICMC–USP) [Bru03]. O objetivo do ASDA é proporcionar ao usuário uma interface de desenvolvimento de simulação distribuída, além de oferecer uma flexibilidade para auxiliar os diferentes perfis de usuários [Bru03]. Esse núcleo poderá, em trabalhos futuros, ser utilizado para o ensino de simulação e avaliação de desempenho.

A troca de mensagens entre os processos distribuídos no `ETW` é feita através da biblioteca *LAM-MPI* (*Message Passing Interface*). O principal motivo da utilização do *MPI* está no fato deste ser uma proposta de padronização. O `ETW` é executado em um ambiente de rede utilizando microcomputadores *Intel Pentium IV* com sistema operacional *Linux*, embora os requisitos mnimos sejam um microcomputador *Intel Pentium III*, com sistema operacional Linux, com *kernel* 2.4.6, 128 *MB* de memria *RAM* e *LAM-MPI* verso 7.0. A execução da simulação, em cada processo, é efetuada seguindo a orientação a evento, conforme a extensão funcional *SMPL* [Mac87].

1.3 Organização do Texto

O Capítulo 2 faz uma revisão sobre avaliação de desempenho e suas técnicas, descrevendo simulação como ferramenta para avaliação de desempenho. Apresenta vantagens e desvantagens da simulação seqüencial, a classificação dos modelos existentes, as fases de construção de um modelo, mostra a natureza da simulação discreta e suas alternativas. Além disso, também apresenta algumas das linguagens e pacotes existentes utilizados para simulação seqüencial.

São descritos os conceitos relacionados à simulação distribuída, apresentados os protocolos conservativos e otimistas e apresentadas algumas linguagens e pacotes de simulação distribuída.

No Capítulo 3 são apresentados os principais conceitos sobre o protocolo *Time Warp*, o qual é objeto de estudo neste trabalho, descrevendo o mecanismo de controle local, sua finalidade e suas funcionalidades e o mecanismo de controle global, o qual é utilizado, principalmente, para efetuar gerenciamento de memória. Além disso, é definido o uso de *plugins* e alguns exemplos, como para cancelamento de mensagens e salvamento de estados.

O Capítulo 4 descreve o `ETW`, apresenta seus módulos de gerenciamento, seus diagramas de classes e descreve as estruturas de dados implementadas através de orientação a objetos com a linguagem *C++*. É apresentado um exemplo da utilização do `ETW`, através da implementação de um modelo hipotético e apresentada a validação do `ETW` através do comportamento esperado e pela comparação com resultados obtidos da simulação seqüencial.

No Capítulo 5 são apresentados os modelos utilizados para a execução da simulação e obtenção dos resultados obtidos com a execução do `ETW`. São apresentadas análises de resultados aplicadas aos modelos hipotéticos implementados e apresentados estudos de caso, com o intuito de mostrar algumas potencialidades do `ETW`.

As conclusões, as contribuições e as sugestões para continuidade do trabalho são apresentadas no Capítulo 6.

Capítulo 2

Simulação

2.1 Considerações Iniciais

Uma das áreas que se tornou essencial com a evolução de sistemas computacionais foi a avaliação de desempenho, pois essa permite qualificar um trabalho executado por um sistema. Pode-se utilizar a análise de desempenho em diferentes abordagens, como em sistemas existentes, sistemas ainda em desenvolvimento ou para a comparação de sistemas. Para cada uma dessas situações existe uma técnica mais adequada.

As técnicas de avaliação de desempenho podem ser divididas em dois grupos, técnicas de aferição e técnicas de modelagem. Essas técnicas são métodos para a obtenção de informações para a análise de um sistema, sob uma dada carga de trabalho. As técnicas de aferição são utilizadas em situações onde o sistema ou um protótipo já existe e as informações são obtidas através de medidas efetuadas no próprio sistema. As técnicas de modelagem são aquelas utilizadas quando o sistema ainda não existe, está em fase de desenvolvimento ou não pode ser experimentado. As técnicas de modelagem envolvem a construção de um modelo, para que esse seja resolvido analiticamente (equações) ou através do uso de simulação [Soa92].

O crescente uso da simulação deve-se à sua grande flexibilidade, pois pode permitir respostas rápidas com as modificações efetuadas no modelo, e ao baixo custo aplicado a ela na solução de modelos, em relação a outras técnicas, como por exemplo a prototipação. Além disso, a simulação é versátil, pois muitos sistemas do mundo real podem ser modelados e resolvidos através dela. Custos podem ser reduzidos com a utilização da mesma, pois é possível realizar testes antes do desenvolvimento de sistemas. A simulação é utilizada no estudo de diversos tipos de sistemas, como por exemplo sistemas computacionais, sistemas de produção, serviços militares e governamentais, serviços financeiros, serviços sociais e ambientais [Kaw99].

Apesar das vantagens da simulação seqüencial, algumas características restringem a sua aplicação [Uls99]. A execução de uma simulação baseia-se na geração de variáveis aleatórias, fazendo com que esta seja executada por um longo período de tempo para que os resultados sejam válidos, garantindo assim que a aleatoriedade não influa nos resultados finais.

Quando se deseja simular um sistema mais complexo, o tempo da simulação pode se tornar praticamente inviável. Para diminuir esse tempo é necessário dividir a simulação em diversos processos e executá-los em paralelo, adotando a abordagem de simulação distribuída [Bag98a].

A simulação distribuída necessita de protocolos que garantam a coerência entre os processos e a execução correta da simulação [Uls99]. Tais protocolos são conhecidos como conservativos e otimistas. Os protocolos conservativos verificam quando é seguro executar um evento, já os protocolos otimistas executam os eventos até detectar um erro. Quando esse erro ocorre, um mecanismo de *rollback* é necessário, para desfazer a computação executada de forma errônea [Fuj00].

A Seção 2.2 descreve conceitos sobre redes de filas, apresentando como um modelo em redes de filas pode ser construído. A Seção 2.3 descreve as técnicas para a avaliação de desempenho, técnicas de aferição e técnicas modelagem. A Seção 2.4 define simulação seqüencial, suas vantagens e desvantagens, a classificação dos modelos existentes, as suas fases de desenvolvimento, destacando a análise de saída e a validação de modelos, conceitos sobre simulação discreta e linguagens e pacotes para simulação seqüencial. A Seção 2.5 descreve o problema fundamental e os conceitos básicos no uso da simulação distribuída, os principais protocolos de sincronização e linguagens e pacotes para simulação distribuída. A Seção 2.6 apresenta as considerações finais.

2.2 Redes de Filas

A técnica de rede de filas permite que previsões sobre o comportamento do sistema sejam feitas, viabilizando tomadas de decisões, tendo em vista a melhoria/adequação do sistema em análise. Como exemplo de algumas mudanças que podem ser feitas no sistema tem-se um aumento na taxa de chegada de clientes, uma redução no tempo de serviço, um aumento do número de servidores e uma diminuição no tamanho máximo da fila (fila finita).

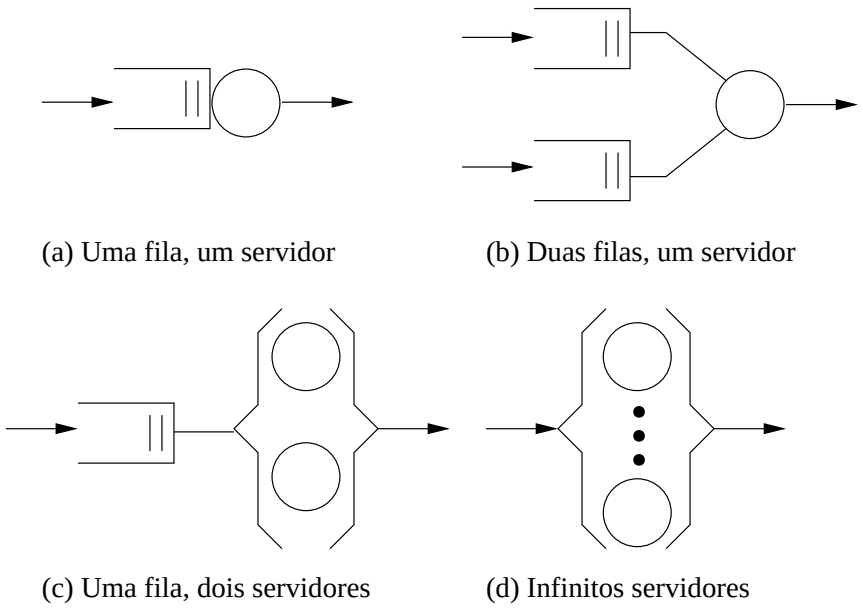
Um modelo de rede de filas é uma coleção de servidores, os quais representam recursos do sistema, e de clientes (requisições), dispostos em áreas de espera (filas), que solicitam a prestação de um serviço a um determinado servidor. Um ou mais servidores e uma ou mais filas formam um centro de serviço [Soa92]. O termo cliente refere-se a algum tipo de entidade que pode ser vista como algo ou alguém que requisita um serviço de um sistema, como por exemplo uma pessoa, um paciente, máquinas, aviões. O servidor refere-se a algum recurso que presta serviços [Ban01].

A disputa por um recurso dentro de um sistema é um ponto fundamental na operação do mesmo. Se não há disputa, a análise de desempenho torna-se mais fácil. Porém, a ausência de disputa em sistemas reais não é uma suposição realista. Como exemplos de recursos tem-se: UCP (Unidade Central de Processamento) de um computador, um enlace de comunicação, um terminal em um escritório, uma pista de vôo, uma estação de trabalho em uma linha de montagem, ou uma pessoa prestando algum serviço.

Os clientes fazem uso desses recursos ao visitá-los e requisitarem seus serviços. Durante o tempo que um cliente está recebendo um serviço, outros podem requisitar o mesmo serviço, o que causará a contenção para o uso do recurso, resultando em filas ou linhas de espera [Soa92].

A Figura 2.1 apresenta vários centros de serviços. A Figura 2.1(a) apresenta um centro de serviço com uma fila de espera e um servidor, a Figura 2.1(b) apresenta um centro de serviço com duas filas de espera e um servidor, a Figura 2.1(c) apresenta um centro de serviços com

uma fila de espera e dois servidores e a Figura 2.1(d) apresenta um centro de serviços sem filas e com infinitos servidores.

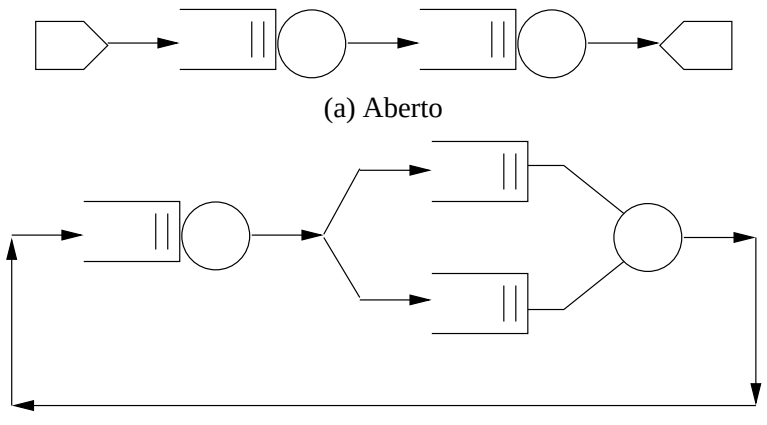


(a) Uma fila, um servidor (b) Duas filas, um servidor

(c) Uma fila, dois servidores (d) Infinitos servidores

Figura 2.1: Exemplos de centros de serviços.

As redes de filas podem ser classificadas como abertas, fechadas ou mistas. Uma rede de filas aberta possui chegadas e partidas externas de clientes, como pode ser observado na Figura 2.2(a). A rede de filas fechada não possui chegada nem partida e os clientes ficam circulando no sistema, como ilustrado na Figura 2.2(b). A rede de filas mista é aberta para alguns clientes e fechada para outros [Soa92].



(a) Aberto

(b) Fechado

Figura 2.2: Exemplos de centros de serviços.

2.3 Técnicas para a Avaliação de Desempenho

Com a evolução das arquiteturas de computadores, a avaliação de desempenho de sistemas computacionais tornou-se uma das mais importantes áreas da computação. Diversas ferramentas têm sido utilizadas para esse fim, como por exemplo os *benchmarks*. A escolha de qual ferramenta utilizar depende, fundamentalmente, do objetivo do estudo e do tipo de avaliação desejada [Uls99].

A avaliação de desempenho de sistemas computacionais pode ser utilizada na seleção de sistemas alternativos, no projeto de novos sistemas, ou ainda para verificar a qualidade e o estado atual de sistemas existentes. O desempenho é o principal foco de avaliação na seleção de um novo sistema computacional. Além disso, pode-se estimar o desempenho de um sistema que ainda não existe, permitindo uma avaliação prévia, isto é, fazer previsões sobre o comportamento do novo sistema. Quando o sistema já existe, é possível um monitoramento sobre ele. Esse monitoramento proporciona informações sobre o estado atual do sistema, podendo prever o impacto de mudanças.

2.3.1 Técnicas de Aferição

As técnicas de aferição podem ser utilizadas quando o sistema está pronto ou existe um protótipo do mesmo, pois essas técnicas analisam o desempenho de um sistema através de sua experimentação. As técnicas de aferição são classificadas em *benchmarks*, coleta de dados e construção de protótipos [San94].

Os *benchmarks* são programas escritos em uma linguagem de programação utilizados para avaliar o desempenho de diferentes sistemas de uma mesma classe de aplicação sob uma mesma carga de trabalho. Como exemplos de *benchmarks*, podem-se citar *Dhrystone* e *SPEC CPU* [Wei02]. A coleta de dados é uma técnica que deve ser efetuada em sistemas existentes. Os dados podem ser coletados utilizando algum tipo de *software* ou *hardware* e é a técnica que fornece resultados mais precisos, dentre as técnicas de aferição. A construção de um protótipo para o sistema permite avaliar o desempenho deste sem a necessidade que o sistema esteja pronto, isto é, não é necessário que esse já exista. O protótipo é uma parte do sistema com suas características mais importantes, descartando o que é irrelevante do sistema. Apesar da utilização de protótipos fornecer boa precisão nos resultados, a construção dessa técnica é de alto custo. A Figura 2.3 mostra a relação dessas abordagens com seus propósitos para a análise de desempenho de sistemas computacionais [San94].

A desvantagem em usar as técnicas de aferição está no fato de que não se pode utilizá-las quando o sistema ainda não existe, sendo necessário realizar testes prognósticos sobre o sistema em desenvolvimento.

2.3.2 Técnicas de Modelagem

A modelagem é geralmente utilizada quando se deseja avaliar um sistema ainda inexistente, trazendo como vantagem a possibilidade de antever o desempenho de um sistema computacional [Sil00].

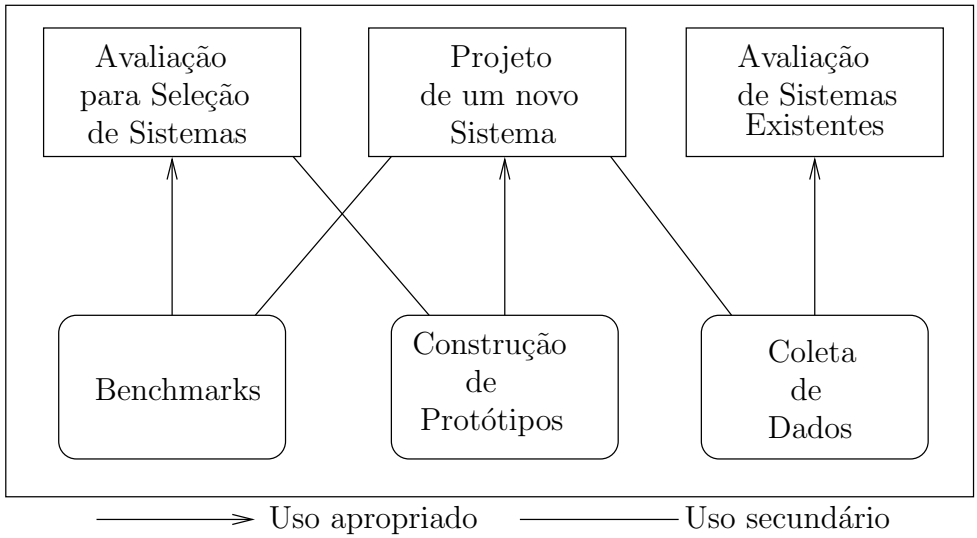


Figura 2.3: Aplicações das técnicas de aferição.

Um sistema é definido como um grupo de objetos que são colocados juntos em alguma interação regular ou interdependência para acomodar os propósitos desejados. Para o uso de técnicas de modelagem, deve-se definir o sistema a ser resolvido e seus problemas corretamente, além do modelo do sistema em estudo [Ban01].

A modelagem pressupõe um processo de criação e descrição, envolvendo um determinado grau de abstração que, na maioria das vezes, acarreta numa série de simplificações sobre a organização e o funcionamento do sistema real. Usualmente, essa descrição toma a forma de relações matemáticas ou lógicas que, no seu conjunto, constituem o que se denomina de modelos [Fre99].

Soares [Soa92] define modelo como sendo uma descrição de um sistema, isto é, o modelo é definido como uma representação de um sistema, com o propósito de estudá-lo. A construção de modelos é um processo complexo e, em muitos casos, uma arte. A partir da criação da abstração do sistema, é possível decidir quais os elementos desse sistema serão incluídos no modelo. Portanto, o modelador deve conhecer o propósito de estudo do sistema fazendo um refinamento, isto é, primeiro deve definir corretamente o sistema e depois definir o modelo para a resolução de problemas. Construído o modelo, esse pode ser resolvido através de solução analítica ou através de simulação.

A técnica analítica é o método mais rápido e o preferido, quando aplicável [Kob78]. Uma desvantagem em se utilizar a solução analítica é que, conforme a complexidade do problema aumenta, a dificuldade da solução também aumenta. Além disso, o tempo para obter a solução desejada depende da complexidade do problema. A simulação é mais geral e pode ser aplicada em situações complexas. Infelizmente, devido às simplificações feitas, o modelo resolvido com a solução analítica pode ter um resultado obtido desprovido de qualquer utilidade [San94].

A simulação é um método de solução de um modelo que tenta representar (imitar) o comportamento de um sistema real. Nesse caso, o modelo pode ser exercitado com dados

obtidos de medidas do sistema real ou de sistemas semelhantes (com objetivos parecidos).

Uma das dificuldades da simulação é a validação dos resultados obtidos. Pode-se comparar o comportamento e os resultados do modelo com o sistema real [Soa92].

2.4 Simulação Seqüencial

A simulação tem a finalidade de mostrar o que acontece em um sistema real ou o que se percebe em um sistema em fase de projeto. Banks [Ban01] descreve algumas vantagens e também algumas desvantagens na utilização da simulação. Como vantagem, é possível explorar novas políticas, procedimentos de operação, regras de decisão, fluxos de informações, procedimentos organizacionais, sem atrapalhar o caminho das operações no sistema real. Uma boa perspectiva pode ser obtida sobre a interação das variáveis e sobre a importância dessas no desempenho do sistema. Além disso, um estudo de simulação pode ajudar no entendimento de como o sistema opera em um todo em vez de como partes individuais são operadas no sistema. Como desvantagem, a construção de modelos requer um treinamento especial. Além disso, se dois modelos são construídos por duas pessoas diferentes, os mesmos podem ser similares, mas é improvável que sejam iguais [Ban01]. Os resultados de uma simulação podem ser de difícil interpretação, pois as saídas de uma simulação são essencialmente variáveis aleatórias. Outra desvantagem reside no fato de que modelagem e análise em simulação podem consumir um grande tempo.

A simulação consiste em resolver o modelo de um sistema para transformá-lo em um programa que represente o sistema real. A inclusão de muitos detalhes pode introduzir complicações desnecessárias e a exclusão de características importantes pode invalidar o modelo [Soa92]. Uma das dificuldades da simulação é a validação dos modelos, que visa determinar o grau de confiança e de corretude dos resultados demonstrando que o modelo é uma representação do sistema real.

A alteração nas variáveis de estado do sistema é a base para a classificação dos modelos em simulação. O tempo é a variável independente e as outras variáveis incluídas na simulação estão em função desse, isto é, são variáveis dependentes do tempo. Modelos de simulação podem ser determinísticos ou estocásticos e discretos ou contínuos.

Modelos de simulação determinísticos não possuem variáveis aleatórias, ao contrário de modelos de simulação estocásticos. Modelo de simulação discreta, ou apenas **modelo discreto**, é aquele em que as variáveis dependentes (variáveis de estado) variam discretamente em pontos específicos do tempo simulado, referidos como tempo de evento. A variável *tempo* pode ser discreta ou contínua. Modelo de simulação contínua, ou apenas **modelo contínuo**, é aquele em que as variáveis dependentes podem variar continuamente ao longo do tempo simulado.

2.4.1 Fases do Desenvolvimento de uma Simulação

O estudo de desempenho através da simulação envolve quatro fases. A primeira fase está relacionada com a formulação do problema e com plano de projeto e conjunto de objetivos,

isto é, é uma fase de descoberta e orientação. A segunda fase está relacionada com a construção do modelo e com a coleta de dados e inclui os passos de conceitualização do modelo, coleta de dados, tradução do modelo, verificação e validação. A terceira fase representa a execução do modelo e envolve o projeto experimental, análise e execução do modelo. Finalmente, a quarta fase, de implementação, relaciona os passos de documentação e relatório e a implementação. O ciclo de vida para construir um modelo para ser resolvido por simulação seqüencial é apresentado na Figura 2.4 [Ban01].

1. Fase de Descobrimento e Orientação

Formulação do Problema: Todo estudo deve começar com uma declaração do problema, isto é, os propósitos e objetivos do estudo devem ser claramente definidos. Se a declaração do problema é feita por um analista, é importante que as pessoas envolvidas entendam e concordem com a formulação.

Plano de Projeto e Conjunto de Objetivos: Os objetivos são definidos para indicar as questões que devem ser respondidas pela simulação. O projeto pode incluir os planos de estudo em termos do número de pessoas envolvidas, do custo do estudo e do número de dias necessários para concluir a fase de desenvolvimento com os resultados antecipados de cada estágio no desenvolvimento. Nesse passo, descreve-se também o que deve ser representado no modelo e é determinado o método que será utilizado para as medidas de desempenho, como por exemplo solução analítica ou por simulação [San94].

2. Fase de Construção do Modelo e Coleta de Dados

Conceitualização do Modelo: A arte de modelagem é realçada com a habilidade de abstrair as características essenciais de um problema, para selecionar e modificar a suposição básica que caracteriza um sistema e enriquecer e elaborar o modelo até resultar em uma aproximação com sucesso. Desse modo, é interessante começar com um modelo simples e construí-lo voltado para uma complexidade maior. Não é necessário que o modelo represente o sistema real de forma completa, podem ser necessárias apenas as características relevantes no estudo de desempenho.

Coleta de Dados: Com a construção do modelo completa, é possível identificar os parâmetros (dados) que serão necessários. Esses parâmetros podem ser classificados como parâmetros de carga (como por exemplo, tempo entre chegadas de novos clientes ao sistema, tempo de execução, tipos e tamanhos de registros) e parâmetros do sistema (tempo para operações de acesso à memória, por exemplo). Caso o sistema real ainda não exista, pode-se utilizar os parâmetros de um sistema existente, com características semelhantes [Mac87].

Tradução do Modelo: O programador deve selecionar o computador, a linguagem de programação e as ferramentas que irá utilizar, dependendo do sistema sendo simulado e dos recursos disponíveis [San89].

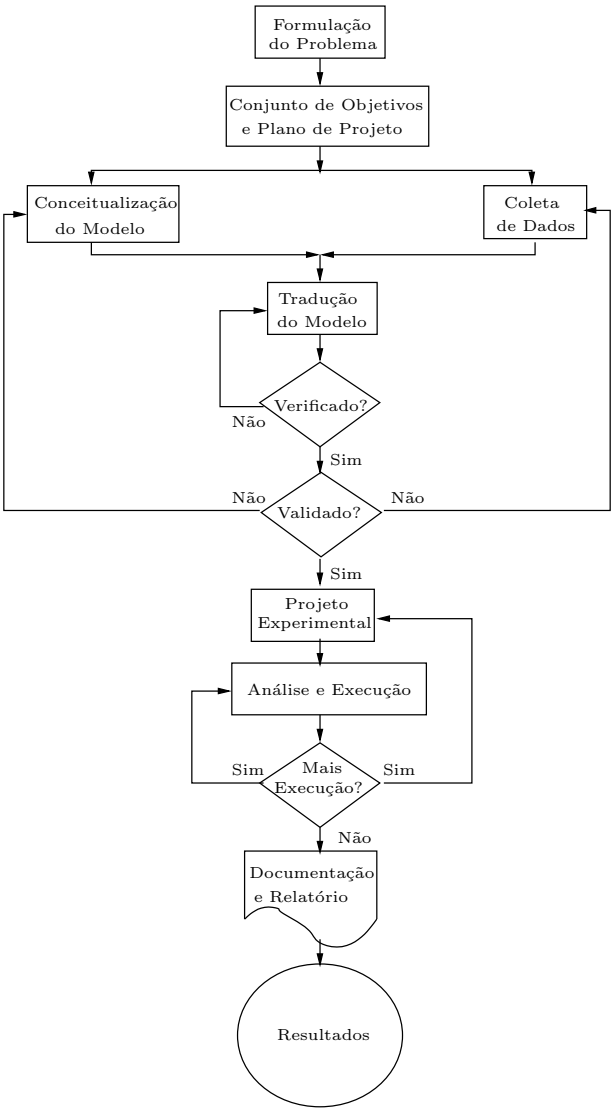


Figura 2.4: Fases de desenvolvimento da simulação seqüencial.

Verificação: A verificação é o processo de determinar que a implementação de um modelo representa exatamente a descrição e especificação conceitual do programador [Hu01]. Para modelos simples, a verificação pode ser feita por inspeção. Já para modelos mais complexos, é necessária uma análise mais detalhada. Uma maneira de efetivar a verificação é comparar o modelo construído com modelos analíticos [Mac87].

A saída do modelo pode ser examinada utilizando variações dos parâmetros de entrada e o resultado calculado representa valores estatísticos. Por exemplo, pode-se considerar várias amostras de um modelo complexo em redes de filas com muitos recursos (centros de serviços) em série, utilizando configuração paralela. Supõe-se que o modelador está interessado no tempo de resposta, definido como o tempo virtual necessário para o cliente (entidade) atravessar por

uma determinada parte da rede de filas. Durante a fase de verificação do modelo, recomenda-se que o programa colete e imprima estatísticas como, por exemplo, utilização de servidores e número médio de clientes na fila. O exame na utilização de um servidor pode mostrar que o tempo de resposta é excessivamente baixo (ou alto). Assim, um possível erro pode ter surgido por alguma especificação errada do tempo médio de serviço do servidor, ou um erro na lógica do modelo no envio de clientes para esse servidor, ou por alguma outra especificação errada [Ban01].

Geralmente, as técnicas mais simples e mais óbvias são as melhores, pois são mais fáceis de serem entendidas e utilizadas. Uma estrutura para verificação e validação é sugerida por Carson [Car02]:

- teste do modelo para *face validity*;
- teste do modelo sob um número de parâmetros de entrada;
- quando aplicável, pode-se comparar as predições do modelo com desempenhos passados do sistema atual ou um modelo base representando um sistema existente. No projeto de um novo sistema, pode-se comparar o comportamento do modelo implementado utilizando hipóteses e especificações.

Validação: A validação é o processo de determinar se um modelo é a representação exata de um sistema real, de acordo com a pretensão do modelo [Hu01]. A validação tem a finalidade de comparar o modelo construído com o sistema em estudo. Quando o sistema sendo modelado já existe, suas medidas são comparadas com resultados de uma simulação do sistema e, se os resultados coincidirem, o programa de simulação é considerado válido. Quando a simulação é feita sobre um projeto, a validação é feita examinando-se o projeto do sistema, justificando cada abstração efetuada [Mac87].

Pode-se considerar dois casos diferentes de validação [Mac87]:

- o sistema modelado existe e pode ser medido. O objetivo da análise é avaliar uma mudança proposta para o sistema e a validação é baseada na comparação dos resultados do modelo com os dados obtidos a partir do sistema;
- o sistema modelado existe apenas como um projeto e o objetivo da análise é estimar o desempenho do sistema projetado ou avaliar projetos alternativos. Quando o sistema modelado não existe, a validação pode ser feita através de uma revisão, onde o modelo é examinado em termos do projeto do sistema e cada hipótese e abstração é justificada.

A validação pode também basear-se em comparações com outros modelos de sistemas já validados, como por exemplo, os modelos resolvidos analiticamente [Spo94].

3. Fase de Execução do Modelo

Projeto Experimental: Envolve determinar as alternativas que serão simuladas. Para cada projeto do sistema que é simulado, é necessário tomar decisões para relacionar o tamanho do

período de iniciação, o tamanho da execução da simulação e qual o método para análise de saída será utilizado, replicação ou *batch means* (Seção 2.4.2) [Ban01, Bru03].

Análise e Execução: A execução e a análise são usadas para obter medidas de desempenho para o sistema que está sendo simulado. Com base em análises de execuções que são completadas, o analista determina as execuções adicionais necessárias e quais projetos os experimentos adicionais podem seguir.

4. Fase de Implementação

Documentação e Relatório: A documentação do programa é necessária por várias razões. Se o programa for utilizado pelo mesmo analista ou por outro, existe a necessidade de entender como o programa funciona. Se existir a necessidade de alguma modificação no programa, a documentação pode facilitar a adequação do analista. Outra necessidade da documentação está na possibilidade de ocorrerem alterações nos parâmetros de entrada de um modelo. Além disso, os relatórios fornecem uma ordem cronológica dos trabalhos efetivados e das decisões tomadas. O resultado da análise também pode ser relatado, permitindo aos usuários a revisão da formulação final, dos critérios de comparação, dos resultados de experimentos e de outras soluções recomendadas pelo problema.

2.4.2 Análise de Saída

A análise de saída é a verificação dos resultados gerados por uma simulação. Seu principal objetivo é permitir a realização de inferências e previsões sobre o comportamento e o desempenho do sistema real sob análise, isto é, tem o objetivo de predizer o desempenho de um sistema ou ainda comparar o desempenho de dois ou mais projetos de sistemas alternativos [Ban01].

A principal razão para uma maior atenção aos processos de análise dos resultados das simulações baseia-se no fato de, em geral, os modelos apresentarem um comportamento estocástico à semelhança dos sistemas que estão imitando. Dessa maneira, adota-se, com relação aos resultados dos modelos, o mesmo tratamento estatístico que pode ser empregado sobre sistemas reais [Fre99].

Para que se possa concluir sobre o desempenho de um sistema, por meio de resultados obtidos a partir de um modelo, é necessário observar o comportamento das variáveis de saída, na medida da realização de experimentos simulados. Uma vez realizados os experimentos, estima-se o comportamento do sistema real por um processo de inferência a partir do conjunto de resultados obtidos [Fre99].

O procedimento de análise dos resultados da simulação inicia com a seleção de variáveis relacionadas ao tipo de desempenho que se deseja aferir no sistema. Essas variáveis podem ser, por exemplo, elementos contadores de ocorrências ou ainda serem obtidas de alguma expressão incluindo médias, variâncias, etc [Fre99].

Em sistemas que possuem clientes, servidores e filas, as variáveis para medidas de desempenho mais utilizadas são [Soa92]:

- Utilização do servidor: fração de tempo que o servidor está ativo;
- *Throughput* ou vazão: número de serviços completos por unidade de tempo;
- Tamanho médio da fila: número médio de usuários que esperaram para serem atendidos pelo servidor;
- Tempo médio de espera na fila: razão entre o tempo total de espera na fila e o número de clientes na fila;
- Tempo médio de serviço: razão entre o tempo que o servidor foi utilizado e o número total de clientes servidos;
- Tempo médio no sistema: tempo médio que os clientes gastaram no sistema, contando o tempo de espera na fila e o tempo de serviço.

Geração de Números Aleatórios

Na simulação, os números aleatórios são utilizados, por exemplo, na geração dos tempos entre chegadas, tempos de serviços, entre outros. Além disso, a simulação utiliza várias funções de distribuição de probabilidade, tais como exponencial e normal, e essas funções utilizam as rotinas de geração de números aleatórios [Ban01].

A geração de números aleatórios é efetuada através de algoritmos determinísticos, ou seja, através de algoritmos que produzem uma seqüência de números que não é aleatória de fato mas que se comporta como aleatória. Essas seqüências são chamadas de pseudo-aleatórias e os programas que geram essas seqüências são chamados de geradores de números pseudo-aleatórios. No entanto, no contexto da simulação, os números pseudo-aleatórios são denominados de aleatórios por uma questão de simplificação da linguagem [Ban01].

Os geradores mais utilizados e conhecidos são os *LCG* (*Linear Congruential Generator*), onde um número da seqüência é uma função do número anterior, seguindo a Equação 2.1:

$$x_{i+1} = (ax_i + c) \mod m, i = 0, 1, 2, ... \tag{2.1}$$

O valor inicial x_0 é denominado semente, a é chamado de constante multiplicadora, c é um incremento e m é o módulo. Se $c \neq 0$, a Equação 2.1 é chamada de método congruente da mistura. Quando $c = 0$, é conhecida como método congruente multiplicativo. A seleção de valores para a, c, m e x_0 afeta as propriedades estatísticas [Ban01].

Período de *Warm-Up* (Aquecimento)

No período de *warm-up* ou período de aquecimento, os dados colhidos pelo programa durante o mesmo não são significativos à avaliação. Esse período de aquecimento chega ao final quando o sistema fica em equilíbrio. Um sistema fica em equilíbrio quando atinge o estado estacionário, que pode ser atingido, por exemplo, se o padrão de chegada não se altera e o sistema tem a capacidade de fornecer o serviço solicitado [Soa92]. Com a questão do

warm-up resolvida, efetua-se uma análise dos resultados da simulação (as variáveis de saída), possibilitando verificar a sua precisão [Orl95].

A maior dificuldade na remoção da fase de aquecimento é a impossibilidade de determinar o seu término com muita clareza. Alguns métodos são apresentados a seguir [Fre99]:

- Simulação Longa: consiste na execução de uma simulação longa, isto é, tão longa que assegure que as condições iniciais de instabilidade não afetem os resultados;
- Inicialização Apropriada: requer a inicialização das variáveis do sistema em um estado próximo das condições de estabilidade. Embora o método contribua para uma substancial redução do período de simulação, a determinação das condições iniciais nem sempre é uma tarefa fácil de ser realizada;
- Observação Visual: é considerado o método mais simples para a determinação do ponto de término do período de *warm-up*. A partir da construção de um gráfico, procura-se observar, de forma aproximada, em que momento as respostas passam a ter uma conduta mais estabilizada.

Métodos para a Análise de Saída

A simulação de sistemas estocásticos requer que a variabilidade dos elementos no sistema seja caracterizada utilizando-se conceitos probabilísticos. Questões importantes como o período de tempo ou quantas vezes executar o programa de simulação devem ser consideradas. As entradas, em um programa de simulação, são obtidas através da geração de variáveis aleatórias de distribuições de probabilidades, as saídas da simulação são funções dessas variáveis e, portanto, a análise de saída é um problema estatístico. Isso significa que os resultados das execuções de uma simulação devem ser cuidadosamente analisados para total compreensão de seu significado. Existem métodos descritos na literatura para determinar quando parar a execução de um programa de simulação, os quais se baseiam no cálculo de um intervalo de confiança para a média de uma variável de saída [Bru03].

Replicação/Deleção Essa abordagem executa k replicações independentes, cada uma com $l+n$ observações e descarta as primeiras l observações de cada execução. Assim, cada replicação i produz n observações (X_{i_1}, X_{i_2}, ..., X_{i_n}). A Equação 2.2 apresenta as médias amostrais:

$$Y_i(l,n) = \frac{1}{n-l} \sum_{j=l+1}^{l+n} X_{ij} \tag{2.2}$$

que são variáveis aleatórias independentes e identicamente distribuídas (iid), a Equação 2.3

$$\overline{Y}_k(l,n) = \frac{1}{k} \sum_{i=1}^{k} Y_i(l,n) \tag{2.3}$$

é um estimador não viciado da média amostral μ e da variância amostral dos Y_i's e $\overline{Y}_k(l,n) \pm t_{k-1,1-\alpha/2}\sqrt{\widehat{V}_R(n,l)/k}$ é o intervalo de confiança aproximado $1-\alpha$ para μ, onde $\widehat{V}_R(l,n)$ é a variância amostral de $Y_i(l,n)$'s.

O método é simples, mas envolve a escolha de três parâmetros importantes, l, n e k. Algumas observações importantes sobre os valores desses parâmetros são [Ale01]:

- Se l aumenta para um valor fixo de n, o erro sistemático em cada $Y_i(l,n)$ devido às condições iniciais diminui mas o erro amostral aumenta devido ao número pequeno de observações (a variância de $Y_i(l,n)$ é proporcional a $1/(n-1)$);
- Se n aumenta para um valor fixo de l, os erros sistemático e amostral em $Y_i(l,n)$ diminuem;
- O erro sistemático na média amostral $Y_i(l,n)$ não é reduzido aumentando o número de replicações k;
- Para um número fixo n, sob alguma condição que é satisfeita por uma variedade de saídas na simulação, o intervalo de confiança é assintoticamente válido apenas se l/ln com $k \to \infty$ for equivalente a $k \to \infty$.

Média dos Lotes (*Batch Means*) Ao contrário do método das replicações, este método é baseado em uma única e longa execução da simulação para a estimação do intervalo de confiança. A seqüência original de n observações (X_1, X_2, ..., X_n) é dividida em k lotes não sobrepostos de tamanho m. As médias desses lotes (*batch means*) podem ser utilizadas para o cálculo do intervalo de confiança. Considerando-se k lotes de m observações cada um, o i-ésimo lote será composto por $X_{(i-1)m+1}, X_{(i-1)m+2}, ..., X_{im}$ para $i = 1, 2, ..., k$ e a i-ésima média é dada pela Equação 2.4 [Ale01]:

$$Y_i(m) = \frac{1}{m}\sum_{j=1}^{m} X_{(i-1)m+j} \tag{2.4}$$

As Equações 2.5 e 2.6 representam a média dos lotes e a variância, respectivamente e o intervalo de confiança é representado por $\overline{X}_n \pm t_{k-1,1-\alpha/2}\sqrt{\widehat{V}_B(n,k)/k}$ [Ale01].

$$\overline{X}_n = \overline{Y}_k(m) = \frac{1}{k}\sum_{i=1}^{k} Y_i(m) \tag{2.5}$$

$$\widehat{V}_B(n,k) = \frac{1}{k-1}\sum_{i=1}^{k} (Y_i(m) - \overline{X}_n)^2 \tag{2.6}$$

Se após essas m observações o intervalo obtido alcançou a precisão desejada, a simulação é encerrada, caso contrário um novo lote é obtido e a média final é recalculada [Mac87].

A questão principal na utilização do método *batch means*, na prática, é a escolha do tamanho de um lote m.

2.4.3 Simulação Discreta

O estado de um sistema é definido por uma coleção de variáveis necessárias para descrever o sistema em algum tempo, relativo aos objetivos de estudo. Um evento é definido como uma ocorrência instantânea que pode mudar o estado do sistema [Ban01]. Na simulação discreta, o estado (variáveis de estado) do sistema só pode mudar nos instantes de ocorrência dos eventos. Entre os eventos, o estado do sistema permanece constante. Nesse caso, cada evento a ser executado possui uma marca de tempo e determina as mudanças no estado do sistema que está sendo modelado. Essa marca de tempo determina quando as mudanças irão ocorrer.

A lista de eventos consiste dos eventos futuros, ordenados por seus tempos de ocorrência (também conhecida como Lista de Eventos Futuros (LEF) ou FEL do inglês *Future Event List*).

A formulação de um modelo para simulação discreta pode ser realizada de três formas [Soa92]:

- pela definição das mudanças nos estados que podem ocorrer em cada tempo de evento;
- pela descrição das atividades nas quais as entidades do sistema se envolvem;
- pela descrição dos processos através do qual as entidades do sistema derivam.

Basicamente, atividades, processos e eventos são utilizados para descrever o comportamento dinâmico de sistemas discretos e sobre os quais as linguagens de simulação para esses sistemas são baseadas [Soa92]. Pode-se dizer que um evento acontece em um ponto isolado no tempo, no qual decisões devem ser tomadas de forma a iniciar ou terminar uma atividade. Um processo é uma seqüência de eventos e pode englobar várias atividades. A relação entre processos, atividades e eventos pode ser vista na Figura 2.5 [Soa92]. A abordagem de simulação orientada ao exame da atividade não é utilizada devido à sua ineficiência. As alternativas viáveis para simulação discreta são:

- Simulação Orientada a Evento;
- Simulação Orientada a Processo.

Simulação Orientada a Evento

Na simulação orientada a evento, um sistema é modelado pela definição das mudanças que ocorrem em tempos de ocorrência de evento. O modelador tem que determinar os eventos que podem causar as mudanças nos estados do sistema e desenvolver a lógica associada com cada tipo de evento.

O programa de simulação remove o evento da LEF com a menor marca de tempo para executá-lo. Quando o programa completa seu processamento em um instante de tempo, o próximo evento é retirado do início da lista, o relógio é avançado para o tempo de ocorrência

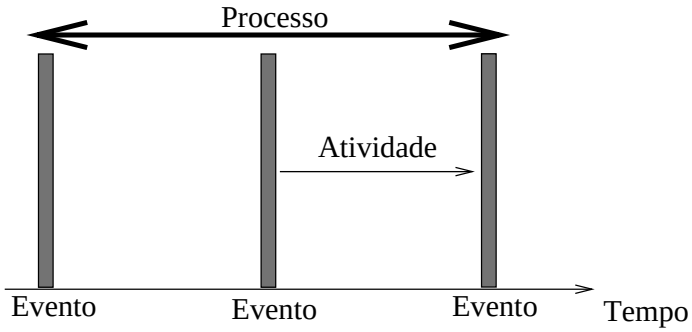


Figura 2.5: Processos, atividades e eventos.

do mesmo e o programa de simulação inicia a execução do evento retirado da lista, garantindo que todos os eventos ocorram em uma ordem cronológica [Ban01, Fuj90].

Como exemplo, considere um sistema de atendimento de clientes em um caixa de supermercado. Quando o cliente termina de efetuar suas compras, dirige-se para a fila do caixa e tem que esperar sua vez de ser atendido. Assim que o cliente passa todas as suas compras pelo caixa, paga o valor devido e deixa o sistema (caixa). Percebe-se que, nesse caso, é irrelevante conhecer quando e como o cliente leva suas compras embora. As mudanças das variáveis de estado no sistema podem ocorrer de duas formas:

- um cliente chega ao caixa do supermercado para pagar suas compras;
- um cliente paga suas compras e deixa o caixa.

O sistema, nesse caso, deve conter uma variável de estado que indica se o caixa está ocupado ou não e uma fila de espera para os clientes aguardarem um caixa se tornar livre [Tan94]. Além disso, o sistema deve conter variáveis de tempo para indicar o tempo de chegada do cliente ao caixa e o seu tempo de atendimento.

Como pode-se observar, ocorrem dois eventos, de acordo com a lógica do sistema. O primeiro evento é a chegada do cliente ao caixa do supermercado. Nesse caso, o sistema deve escalonar a próxima chegada de um cliente e em seguida verificar se o caixa está ocupado. Se isso ocorrer, o sistema deve incluir o cliente em uma fila de espera, caso contrário o caixa é dito como ocupado e o fim do atendimento é calculado. No segundo evento, saída do cliente do caixa, o sistema verifica se existe algum cliente esperando na fila. Em caso afirmativo, o sistema retira esse cliente da fila de espera para ser atendido e calcula o seu tempo de atendimento, caso contrário (não existe mais cliente na fila do caixa) o caixa fica desocupado.

Simulação Orientada a Processo

O programa de simulação é organizado como um conjunto de processos, os quais executam concorrentemente durante a simulação. A descrição de um processo tem a forma de um procedimento de uma linguagem de programação [Mac87].

Na simulação orientada a processo existe um mecanismo de escalonamento e uma estrutura de lista, com a diferença de que cada entrada na lista define um processo e seu ponto de reativação (para processos que são bloqueados e novamente postos para executar).

Basicamente, uma simulação orientada a processo deve seguir os seguintes passos [Spo94]:

- definir as entidades do sistema;
- criar um processo para cada objeto do sistema descrevendo suas etapas;
- executar concorrentemente os processos.

No exemplo dos clientes sendo atendidos por um caixa em um supermercado, o processo do cliente verifica se o caixa está desocupado. Se estiver, o estado do caixa torná-se ocupado, o cliente ocupa o caixa pelo tempo de atendimento (escalona processo para tempo_corrente mais tempo_atendimento) e em seguida o estado do caixa é ajustado para desocupado e o cliente deixa o caixa (sistema). No processo de chegada de clientes, é indicado a chegada de um novo cliente e um processo de chegada de clientes é escalonado para tempo_corrente mais tempo_entre_chegadas [Spo94].

2.4.4 Linguagens e Pacotes de Simulação

O intenso uso de simulação como uma abordagem de análise de desempenho de sistemas deu origem a uma série de ferramentas projetadas especificamente para esse fim. Além disso, linguagens para simulação podem facilitar o desenvolvimento e execução na simulação de sistemas reais complexos [Ban01]. Essas diversas linguagens e pacotes de modelagem impõem uma certa estruturação nos modelos e simplificam suas soluções [Soa92].

Um requisito importante para as linguagens de simulação é a presença de mecanismos para a representação do tempo. Enquanto o sistema sendo modelado executa em tempo real, a simulação trabalha com um relógio próprio, que marca a passagem do tempo no programa de simulação. Outro requisito importante é a capacidade de fornecer facilidades para coletas de estatísticas e emissão de relatórios [Bru97, Fuj90].

Para a implementação de uma simulação segundo uma das orientações apresentadas, existem linguagens e pacotes de simulação que possibilitam ao modelador escrever programas de simulação orientados a evento ou processo.

Pode-se usar os seguintes enfoques para a implementação de um modelo de simulação [San94]:

- linguagens de programação convencionais;
- linguagens de simulação;
- pacotes de uso específico;
- extensões funcionais.

Linguagens de programação convencionais

Toda linguagem de programação é uma candidata a linguagem de simulação. Porém, para desenvolver um programa de simulação em uma linguagem convencional (C/C++, Pascal, FORTRAN), o programador deve criar todo o ambiente de simulação, pois, além de não serem linguagens específicas para simulação, não oferecem todas as ferramentas necessárias para um ambiente de simulação (estrutura de dados, facilidades para manipulação de listas, coleta de estatísticas e emissão de relatórios). Além disso, o programador deve ter conhecimento tanto da linguagem de programação quanto da simulação para criar um ambiente adequado.

Linguagens de simulação

São linguagens projetadas para a simulação de vários tipos de sistemas [Soa92]. Essas linguagens já contêm todas as estruturas necessárias para a criação de um ambiente de simulação. Assim, o programador não necessita criar um novo ambiente de simulação. Essas linguagens podem ser classificadas em orientadas a evento ou a processo. Como exemplo de linguagens orientadas a evento tem-se Simscript II.5, Slam II, Siman V, ModSim III [Ban01] e SimJAVA [Kre97]. Como exemplo de linguagens orientadas a processo tem-se, RESQ [Soa92], Simula, Slam II [Ban01], GPSS/H [Cra99].

Pacotes de uso específico

Devido ao fato de serem muito específicos para uma dada aplicação, esses pacotes oferecem facilidades em pontos particulares da aplicação para a qual foram desenvolvidos, mas são pouco flexíveis a mudanças [Spo94]. Nesses pacotes, a formulação do modelo é constituída na própria ferramenta, com parâmetros definidos pelo pacote. Por exemplo, pacotes para a modelagem de sistemas computacionais e para sistemas operacionais, para a modelagem de operações de siderurgia, para a modelagem de sistemas aeroespaciais e pacotes para sistemas de produção. Dentre eles, pode-se citar:

- SimFactory II.5: é um pacote de uso específico que utiliza a linguagem de simulação orientada a eventos *Simscript II.5* e é utilizado para modelar sistemas para operações de manufaturação [Gob91];
- ProModel: é uma ferramenta de simulação e animação projetada para modelar sistemas de manufaturação de vários tipos, rapidamente e corretamente. Além disso, possui elementos de modelagem capazes de construir blocos para representar componentes físicos e lógicos, como entidades, recursos, chegadas, etc [Har03];
- AutoMod: é uma ferramenta de simulação que permite aos usuários construir modelos de tamanhos e complexidades que podem ser usados não apenas para planejar e projetar sistemas, mas para análises de operações do dia-a-dia e para controlar desenvolvimentos e testes. Combina gráficos 3D com conjuntos de *templates* e objetos para modelar diferentes aplicações. Além disso, permite o reutilização de objetos de modelos e pode ser conectado a outros softwares, como por exemplo o *ActiveX* [Roh02, Sch02, Sta01];

- Arena: é uma ferramenta que pode ser usada para simular sistemas discretos ou contínuos e possui um ambiente gráfico para desenvolvimento de modelos, construídos através de objetos chamados módulos. Esses objetos definem os componentes físicos e lógicos do sistema, como por exemplo máquinas, operadores, entre outros. [Rat02, Roh02, Sad99].

Extensões funcionais

Para facilitar o trabalho do programador, algumas bibliotecas podem ser inseridas em linguagens convencionais, compondo assim, um ambiente completo de simulação. O trabalho do programador é facilitado, pois esse não necessita aprender uma nova linguagem [Spo94]. As extensões funcionais unem as vantagens das linguagens convencionais e das linguagens de simulação [Bru00]. Como exemplo, pode-se citar:

- Extensões da linguagem C: *SMPL* (orientada a evento) [Mac87], *CSim* [Hla02, Pan97] e *EFC* (orientadas a processo);
- Extensões da linguagem C++: *SIMPACK* [Fis92], *C++Sim* [Lit93] e *SimKit* [Gom95];
- Extensões da linguagem Modula 2: *EFM2* [Spo91] e *HPSim* [Sha88] (orientadas a processo).

2.4.5 *SMPL*

O *SMPL* é uma extensão funcional da linguagem C para simulação discreta orientada a eventos. Além de facilitar o uso do usuário, pois o mesmo não precisa aprender uma nova linguagem de programação, as extensões funcionais apresentam grande facilidade na construção de um programa de simulação baseado em um modelo [Mac87, Uls99].

Estrutura *SMPL*

A extensão funcional *SMPL* possui três entidades básicas para a representação dos modelos [Mac87]:

- recursos (*facilities*);
- *tokens*;
- eventos.

Cada facilidade representa um recurso (ou centro de serviço) do sistema que está sendo modelado. Cada recurso no sistema pode conter um ou mais servidores e uma fila de espera. As primitivas *SMPL* permitem definir, reservar e liberar recursos e verificar seu estado. A interconexão dos recursos é determinada pelo roteamento dos *tokens* entre as mesmas [Uls99].

Os *tokens* representam as entidades ativas do sistema, isto é, é o *token* quem reserva ou não uma facilidade. A extensão *SMPL* fornece dois tipos de operações para o controle de

fluxo dos *tokens* na simulação, que envolvem a reserva de uma facilidade e o escalonamento de eventos. Se um *token* tenta reservar uma facilidade que não possui servidores livres, o mesmo é inserido na fila da facilidade e só sairá dessa quando algum servidor da facilidade tornar-se disponível. O único atributo que o *SMPL* disponibiliza aos *tokens* é a prioridade, se um *token* possui maior prioridade que outro, o de menor prioridade pode ser retirado de uma facilidade.

Um evento representa uma mudança nas variáveis de estado do sistema. Quando uma tarefa ou processo encerra sua execução e libera uma facilidade, ocorrerão dois eventos: o final da execução da tarefa e a liberação da facilidade. No *SMPL*, um evento é caracterizado pelo seu número, seu tempo de ocorrência e pela identificação do *token* relacionado a esse evento.

A Figura 2.6 apresenta a estrutura básica de um programa de simulação utilizando o *SMPL*. Os programas desenvolvidos com o *SMPL* são caracterizados por um laço principal que comanda as iterações da simulação. Dentro desse laço os eventos são executados e os *tokens* são escalonados para novos eventos e são realizadas operações para a coleta de dados estatísticos [Uls99].

O *SMPL* mantém um *pool* de elementos que podem ser requisitados como descritores para recursos, indicações para as filas desses recursos e para os elementos na lista de eventos futuros. Com esses elementos, o *SMPL* cria estaticamente as listas encadeadas e as estruturas necessárias para a simulação, lista de eventos futuros, filas de eventos e descritores dos recursos.

Primitivas do *SMPL*

O *SMPL* apresenta várias primitivas funcionais para o desenvolvimento de programas de simulação orientados a eventos, das quais as principais são: *smpl*, *facility*, *request*, *release*, *schedule* e *cause*:

- *smpl*: é responsável por iniciar o sistema para a execução da simulação. Inicia as estruturas de dados do *SMPL* e o relógio da simulação (tempo virtual);
- *facility*: cria, nomeia e devolve um descritor para cada facilidade no modelo. O descritor é utilizado para qualquer tarefa a ser realizada pela facilidade, isto é, o descritor representa o identificador da facilidade. Como exemplo de tarefas a serem realizadas pelo descritor pode-se citar a requisição, liberação e a preempção de eventos em servidores;
- *request*: é utilizada para requisitar o atendimento de um dos servidores de um recurso. Se todos os servidores da facilidade estão ocupados, o token solicitante é inserido na fila da facilidade, caso contrário, um servidor livre é reservado para o *token*;
- *release*: libera um servidor de uma determinada facilidade. Após a liberação, o sistema atualiza as variáveis estatísticas e decrementa o número de servidores ocupados. Se a fila da facilidade não estiver vazia, o primeiro evento dessa fila é retirado da mesma e é inserido no início da lista de eventos futuros, para execução imediata;
- *schedule*: essa primitiva é responsável pelo escalonamento de eventos na simulação. Todo evento escalonado é inserido na lista de eventos futuros, a qual está em uma ordem não decrescente dos tempos de ocorrência dos eventos;

```
#include ¨smpl.h¨
main()
{
    // Declaração de Variáveis
    // Inicialização de Variáveis
    facility()
    schedule()
    while()
    {
        cause(evento, token)
        {
            switch(evento)
            {
                case 1:
                case 2:
                case 3:
            }
        }
    }
    report()
}
```

Figura 2.6: Estrutura básica de um programa *SMPL*.

- *cause*: essa primitiva remove o elemento no início da lista de eventos futuros para execução e avança o relógio da simulação para o tempo de ocorrência do evento retirado.

2.5 Simulação Distribuída

A simulação distribuída refere-se à habilidade de executar um programa de simulação em sistemas computacionais paralelos ou distribuídos. Os programas paralelos executam em um conjunto de processadores dentro de um mesmo gabinete, muitas vezes, compartilhando memória. Já programas distribuídos executam em máquinas geograficamente distribuídas, interligadas por uma rede de comunicação.

Com o aumento do interesse no uso da simulação de sistemas computacionais, sistemas mais complexos passaram a ser utilizados em simulação, tornando a execução da simulação

seqüencial custosa. Assim, ao invés de simular sistemas computacionais utilizando simulação seqüencial, surgiu a necessidade de utilizar-se a simulação distribuída, com o principal objetivo de reduzir o tempo de execução da simulação. Como exemplo, pode-se considerar o controle do tráfego aéreo em um aeroporto, cujo objetivo é garantir um sistema de vôo seguro e eficiente. Infelizmente, várias tempestades podem ocorrer, causando grandes transtornos, como por exemplo atrasos de vôos que partem e chegam no aeroporto. A simulação do tráfego aéreo de um aeroporto proporciona várias estratégias para determinar qual a maneira mais efetiva para proceder em um desses casos, mas a simulação pode necessitar algumas horas para efetuar essa tarefa, enquanto o sistema deve tomar decisões em alguns minutos [Fuj00].

Para reduzir o tempo de execução da simulação, duas soluções são estudadas: *SRIP* (Simples Replicações em Paralelo) e *MRIP* (Múltiplas Replicações em Paralelo) [Ewi99, Fuj90]. O objetivo da simulação *SRIP* é eliminar a lista de eventos em sua forma tradicional, já que a natureza seqüencial da lista de eventos e o relógio global limitam o potencial de paralelismo da simulação. A técnica *MRIP* é uma alternativa quando se deseja utilizar diversos processadores para realizar uma simulação. Ao contrário das técnicas utilizadas na *SRIP*, na qual um único programa de simulação é dividido em processos lógicos e cada processo é simulado em um processador, na técnica *MRIP* várias instâncias (replicações) de um mesmo programa de simulação seqüencial são executadas independentemente em diferentes processadores [Ewi99].

Além do tempo de simulação, outras vantagens de utilizar simulação distribuída são [Fuj00]:

- a distribuição geográfica, pois um programa de simulação pode ser distribuído em amplas áreas;
- a interação de simuladores que executam em máquinas de diferentes arquiteturas;
- a tolerância a falhas, pois se um processador é desligado ou a energia elétrica onde está situado esse computador é interrompida, é possível que outros processadores passem a executar a tarefa que aquele processador estava executando.

A simulação seqüencial utiliza três estruturas de dados principais:

- variáveis de estados que descrevem o estado do sistema;
- lista de eventos futuros contendo os eventos escalonados e seu relógio global.

O mecanismo no qual um item da lista de eventos futuros é retirado a cada ciclo da simulação não permite uma execução paralela, mesmo em uma máquina com mais de um elemento de processamento, já que a lista de eventos não pode ser particionada.

Devido a essa situação, a estrutura utilizada na simulação seqüencial teve que ser adaptada para os propósitos da simulação distribuída. Surgiram, dessa forma, novos protocolos para garantir a execução da simulação distribuída da mesma forma que a simulação seqüencial, isto é, em ordem não decrescente do tempo de ocorrência da simulação. Algoritmos de sincronização para garantir que a simulação distribuída em um sistema gere os mesmos resultados que a simulação seqüencial levaram ao desenvolvimento de protocolos de sincronização, classificados como conservativos e otimistas, construídos para evitar os erros de causa e efeito.

O relacionamento de causa e efeito determina que, em um sistema real sendo simulado, se um evento E_1 ocorre antes que um evento E_2, o mesmo deve ocorrer na simulação, isto é, se um evento E_1 possui uma marca de tempo menor que E_2, E_1 deve ser executado antes de E_2. Além desses dois tipos de protocolos, vários autores propõem uma combinação dos mesmos, nos quais o sistema ora executa de maneira otimista, ora executa em modo conservativo. Esses protocolos são conhecidos como protocolos mistos, ou ainda, protocolos híbridos [Bag98a, Fuj00, Mor00].

Os protocolos conservativos evitam a ocorrência de erros de causa e efeito, isto é, não permitem a execução de eventos fora da ordem cronológica. Esses protocolos necessitam de estratégias que determinam quando é seguro processar um evento.

Já os protocolos otimistas usam uma estratégia de detecção e recuperação. Os eventos são executados sem que haja nenhuma verificação quanto à ordem cronológica. Quando um erro de causa e efeito é detectado, um mecanismo de *rollback* é utilizado para recuperar o estado da simulação para um estado consistente. A simulação continua a execução dos eventos partindo do estado da simulação recuperado.

Além da simulação distribuída, a execução otimista pode ser também utilizada em sistemas de banco de dados distribuídos, onde vários clientes podem iniciar transações simultaneamente, e em microprocessadores com a implementação de *pipelines* [Fuj00].

A simulação distribuída é composta de um conjunto de simulações seqüenciais, onde cada simulação é modelada em partes diferentes do sistema físico executando em processadores diferentes [Fuj00]. O sistema físico pode ser visto como uma coleção de N processos físicos interagindo um com o outro, modelados por N processos lógicos $PL_0, PL_1, ..., PL_n$ (cada simulação seqüencial). Cada processo lógico está relacionado a um processo físico. As interações entre os processos físicos são modeladas na simulação distribuída através da troca de mensagens, com marcas de tempo de eventos, entre seus processos lógicos correspondentes.

Cada processo lógico deve executar seus eventos[1]. Desde que um processo lógico execute seus eventos em uma ordem cronológica não decrescente da marca de tempo, não ocorrerão erros de causa e efeito. Como exemplo, considere um evento E_1 executado em um processo lógico PL_1, com marca de tempo igual a 10 e um evento E_2 sendo executado em um processo lógico PL_2 com marca de tempo igual a 20, como mostra a Figura 2.7(a) [Fuj00].

Se o evento E_1 escalonar um novo evento E_3 para ser processado em PL_2, com uma marca de tempo menor que 20, como por exemplo 15, é necessário executar esses três eventos de maneira seqüencial, como pode ser visto na Figura 2.7(b) [Fuj00].

As seções a seguir descrevem os protocolos conservativos e otimistas, apresentando suas principais características e estruturas de dados utilizadas.

2.5.1 Protocolos Conservativos

O problema básico associado aos protocolos conservativos é determinar quando é seguro processar um evento. Em outras palavras, um processo lógico saberá que pode processar um evento E_1 se nenhum outro evento com uma marca de tempo menor chegar, garantindo assim a

[1]Um processo lógico pode executar tanto eventos gerados por ele mesmo quanto por outros processos.

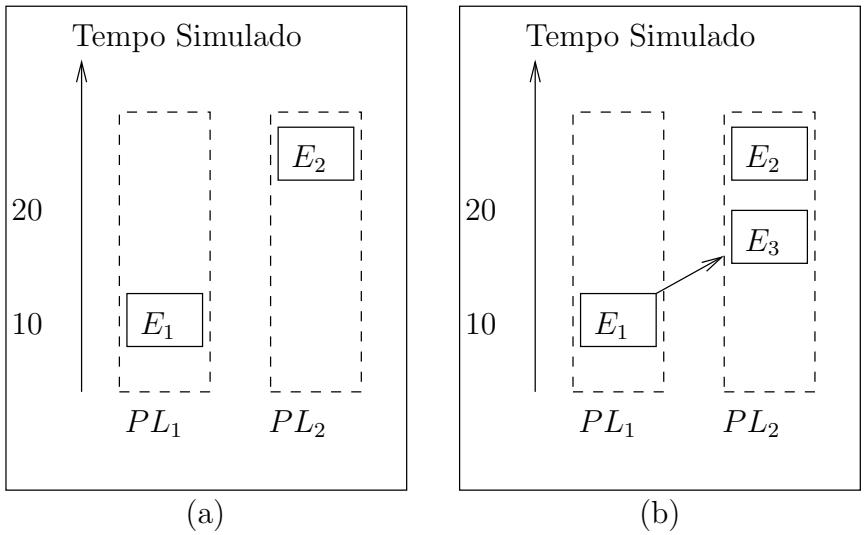


Figura 2.7: Uma possível ocorrência de erros de causa e efeito.

não ocorrência de erros de causa e efeito. Processos nos quais há eventos que não são seguros para processamento devem ser bloqueados, podendo ocorrer situações de *deadlock* entre os processos [Fuj00, Mis86].

Nesses protocolos, para que cada processo lógico execute seus eventos em uma ordem da marca de tempo não decrescente (evitando assim erros de causa e efeito), é preciso que canais sejam especificados estaticamente, permitindo que processos lógicos se comuniquem. Para determinar quando é seguro processar uma mensagem, é necessário que uma seqüência de marcas de tempo nas mensagens enviadas por um canal esteja em uma ordem não decrescente da marca de tempo, garantindo que a marca de tempo da última mensagem recebida por um canal será menor que a marca de tempo de uma mensagem recebida posteriormente. As mensagens que chegam em um canal também são armazenadas em ordem *FIFO (First-Come First-Served)*, na ordem das marcas de tempo. Cada canal possui um relógio associado a ele, o qual é igual à marca de tempo do início da fila, se a fila contém alguma mensagem, ou contém a marca de tempo da última mensagem recebida, se a fila está vazia. O processo repetidamente seleciona o canal com o menor valor do relógio e, se existe uma mensagem na fila, esta é processada. O processo é bloqueado se a fila selecionada está vazia.

Uma situação de *deadlock* surge quando um ou mais processos em um ciclo estão bloqueados esperando por ações de outros processos no ciclo, ficando um processo esperando por outro dentro do ciclo. Na Figura 2.8 [Fuj00], todos os processos estão bloqueados esperando para receber uma mensagem (canal de entrada) com o menor valor do relógio local (*LVT - Local Virtual Time*). Para que a simulação não entre em *deadlock*, um protocolo utilizando mensagens nulas foi desenvolvido por Chandy, Misra [Cha79] e Bryant [Bry77].

Uma mensagem nula com uma marca de tempo igual a t_{null}, enviada de um processo lógico PL_i para um processo lógico PL_j é uma garantia de que PL_i não enviará uma mensagem a PL_j com uma marca de tempo menor que t_{null}. Além disso, tais mensagens nulas são tratadas

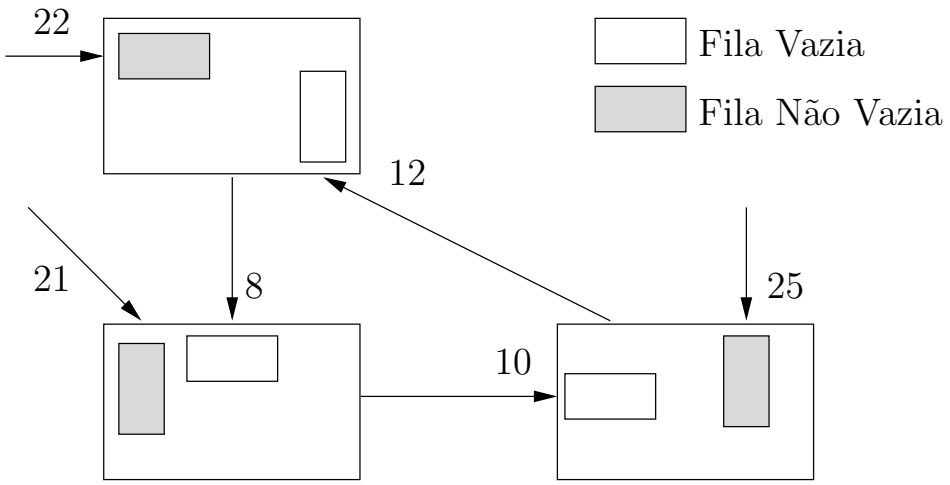


Figura 2.8: Uma situação de *deadlock*.

exatamente como qualquer outra mensagem recebida por um processo lógico e são utilizadas apenas para efeitos de sincronização, isto é, não afetam o progresso da simulação.

Quando um processo lógico recebe uma mensagem, seu LVT é atualizado. Nesse momento, seu processo lógico avança a simulação para o novo valor do LVT. Quando isso acontece, tal processo lógico indica, para cada canal de saída, uma seqüência de mensagens que o processo físico possa ter enviado, gerando uma seqüência de mensagens para transmitir para outros processos, em que existam canais de saída para estes.

Suponha, por exemplo, que um processo lógico PL_i possa indicar futuramente que, após a transmissão dessa seqüência de mensagens, o correspondente processo físico não irá enviar mensagens ao processo PL_j até um tempo t_j. Assim, PL_i envia a mensagem (t_j, nula) ao PL_j quando a seqüência correta de mensagens já foi enviada. Desde que PL_i conheça o estado de seu processo físico até seu LVT, ele pode indicar todas as mensagens que estão sendo enviadas, podendo detectar a ausência de mensagens pelo menos até seu LVT [Fuj90].

A maior desvantagem dos protocolos conservativos está no fato de que eles não podem explorar totalmente o paralelismo disponível na simulação. Por exemplo, se for possível que um evento E_1 possa afetar um evento E_2, diretamente ou indiretamente, o protocolo conservativo deve executar esses eventos seqüencialmente. Mas se a simulação faz com que E_1 raramente afete E_2, tais eventos poderiam ser executados concorrentemente, na maioria das vezes. No pior caso, o protocolo conservativo é pessimista, forçando a execução seqüencial quando ela não é necessária [Fuj90].

2.5.2 Protocolos Otimistas

Os protocolos otimistas, ao contrário dos protocolos conservativos, deixam que seus processos lógicos executem seus eventos sem se preocuparem com erros de causa e efeito. Quando esses ocorrem, são detectados e a simulação retorna a um estado consistente.

O protocolo *Time Warp* é baseado em um paradigma proposto por Jefferson [Jef85],

conhecido como *virtual time*. O *virtual time* é um sistema distribuído executando em coordenação com um relógio virtual igual ao *virtual time*. Os processos se comunicam através da troca de mensagens. Todas as mensagens são marcadas por quatro valores, o emissor, o tempo virtual de envio, o receptor e o tempo virtual de recebimento. O tempo virtual de envio, ou seja, o *LVT* local, é o tempo do momento de envio de uma mensagem e o tempo virtual de recebimento (marca de tempo do evento) é o tempo virtual onde uma mensagem deve ser recebida. Existem duas regras fundamentais utilizadas em sistemas de tempo virtual:

- O tempo virtual de envio de cada mensagem deve ser menor que seu tempo virtual de recebimento;
- O tempo virtual de cada evento em um processo deve ser menor que o tempo virtual do próximo evento a processar.

No protocolo *Time Warp*, um erro de causa e efeito é detectado quando um evento chega com uma marca de tempo menor que o *LVT*. Quando isso acontece, um mecanismo de *rollback* deve ser executado. O evento que causa um *rollback* é conhecido como *straggler* (mensagem atrasada). Além do mecanismo de *rollback*, um mecanismo de recuperação também é necessário para desfazer os efeitos de todos os eventos que foram executados com marca de tempo maior que a marca de tempo do *straggler*.

O protocolo otimista *Time Warp* possui dois mecanismos, o mecanismo de controle local e o mecanismo de controle global. O primeiro é responsável pela execução dos eventos em uma ordem cronológica das marcas de tempo desses, enquanto que o segundo resolve problemas tais como o gerenciamento de memória e gerenciamento de entrada/saída, bem como a detecção do término da simulação. O Capítulo 3 descreve o protocolo *Time Warp* detalhando seus mecanismos, suas principais características e estruturas de dados utilizadas.

2.5.3 Linguagens, Bibliotecas e Ambientes para Simulação Distribuída

Da mesma forma que na simulação seqüencial, a simulação distribuída também fornece opções na utilização de linguagens ou bibliotecas (extensões funcionais) ou ainda pela utilização de ambientes automáticos. As linguagens e bibliotecas têm sido desenvolvidas de modo a liberar o usuário dos detalhes de sincronização e permitir que concentre seus esforços na modelagem do sistema [Low99]. Já os ambientes automáticos possuem o objetivo de não somente liberar o usuário dos detalhes de comunicação e sincronização como também liberar da tarefa de transcrição do modelo em um programa de simulação. Essas linguagens e ambientes são projetos experimentais, em desenvolvimento. A área de simulação distribuída ainda é carente em termos de ferramentas para o usuário.

Linguagens para Simulação Distribuída

Uma linguagem de simulação fornece um conjunto completo de primitivas que permite ao usuário projetar seus modelos de simulação. A principal característica de uma linguagem para

simulação distribuída é a transparência do modelo de programação disponibilizado ao usuário, ou seja, a linguagem deve esconder as características do protocolo de sincronização utilizado. Algumas linguagens para simulação distribuída são [Low99]:

- Apostle (*A Parallel Object-oriented SimulaTion LanguagE*): é uma linguagem orientada a objetos baseada no protocolo *Time Warp* [Won96];
- Maisie: implementa diversos protocolos de sincronização conservativos e otimistas [Bag94];
- Moose (*Maisie-based Object-Oriented Simulation Environment*): é uma versão orientada a objetos da linguagem *Maisie* [Fis96];
- Parsec (*Parallel Simulation Environment for Complex Systems*): é uma evolução da linguagem *Maisie* [Bag98b];
- ModSim: é uma linguagem orientada a objetos baseada na linguagem *Modula-2* e utiliza o protocolo *Time Warp* desenvolvido na *Jade Simulations* [Ric91];
- Yaddes: é uma linguagem que implementa diversos protocolos de sincronização [Gho96].

Bibliotecas para Simulação Distribuída

Uma biblioteca, ou extensão funcional, para simulação distribuída fornece apenas o conjunto de primitivas que devem ser utilizadas em conjunto com a linguagem de programação origem e que possibilitam o desenvolvimento do programa de simulação. Algumas extensões funcionais para simulação distribuída são [Low99]:

- *GTW* e *TWOS*: bibliotecas da linguagem C que implementam o protocolo *Time Warp*. O *GTW* foi desenvolvido pela *Georgia Institute of Technology* e o *TWOS* foi desenvolvido pelo *JTL* (*Jet Propulsion Laboratory*) [Car00, Rei90b];
- *PSK*, *SPaDES*, *SPEEDES* e *WARPED*: bibliotecas acadêmicas orientadas a objeto utilizando a linguagem *C++* que implementam protocolos otimistas [Low99, Ril03, Wil03];
- *ParSMPL*: desenvolvido no Grupo de Sistemas Distribuídos e Programação Concorrente do ICMC-USP, é um ambiente baseado na extensão funcional *SMPL* (linguagem *C*) e utiliza o protocolo conservativo *CMB* [Uls99];

Ambientes de Simulação Distribuída Automáticos

Esses ambientes são ferramentas que oferecem ao usuário uma série de recursos para a elaboração de uma simulação. Por exemplo, o programa de simulação pode ser gerado e executado automaticamente a partir de uma representação gráfica do modelo do sistema a ser avaliado. Para isso, é necessária a inclusão de editores gráficos, interpretadores, compiladores e otimizadores. O ambiente de simulação é definido, assim, como um ambiente de simulação automático [Bru03].

Nesses ambientes de simulação automáticos uma característica importante é a programação visual. Utilizando a programação visual, o modelo de simulação é criado graficamente na tela e os resultados (comportamento do sistema) são observados através de gráficos ou animações. Quando as informações são apresentadas graficamente, o usuário pode manipulá-las de um modo mais eficiente, tornando a programação visual particularmente desejável e vantajosa [Bru03]. A seguir serão descritos alguns ambientes para simulação distribuída (comerciais ou acadêmicos), os quais apresentam todas ou algumas das características descritas:

- UCLA *Simulation Environment*: é dirigido à simulação distribuída e possui três componentes, a linguagem de simulação paralela Parsec, uma interface, denominada Pave (*Parsec Visual Environment*) e os sistemas que implementam os algoritmos de simulação [Bag98b];
- Akaroa-2: um ambiente orientado a objetos e escrito em C++ que utiliza a técnica *MRIP*, possibilitando a execução paralela de programas de simulação (replicações). Os resultados estimados obtidos nas replicações alimentam um analisador global que automaticamente pára a simulação quando a precisão desejada é atingida [Ewi99];
- ASDA: é um Ambiente de Simulação Distribuída Automático, em fase de implementação, o qual possui as principais características do ASiA (Ambiente de Simulação Automático), como por exemplo uma interface amigável e também permite a utilização de uma simulação distribuída por usuários com pouco conhecimento de computação distribuída [Bru03].

2.6 Considerações Finais

A simulação tornou-se uma ferramenta na resolução de sistemas computacionais. O uso de simulação como uma abordagem para a análise de desempenho permite explorar vários caminhos em sistemas computacionais, sem atrapalhar o desenvolvimento do sistema real. As duas principais dificuldades no uso de simulação estão focalizadas basicamente em sua verificação e validação. Esses dois passos na construção de um modelo para simulação são responsáveis tanto por determinar o grau de importância do programa construído quanto mostrar se os objetivos especificados foram alcançados durante o processo de construção do modelo.

O aumento na complexidade dos sistemas computacionais implicou na necessidade da utilização da simulação distribuída, pois conforme a complexidade dos sistemas aumenta, o tempo de execução de um programa de simulação seqüencial também aumenta. Por isso, o principal objetivo a ser alcançado, com a utilização da simulação distribuída, é a redução no tempo de execução na simulação.

Um problema a ser resolvido, quando da implementação de uma simulação distribuída, refere-se a adaptação que as estruturas de dados, componentes da simulação seqüencial, tiveram de sofrer para os propósitos da simulação distribuída, de modo que os eventos continuem sendo executados em uma ordem cronológica de suas marcas de tempo. Assim, surgiram os

protocolos de simulação distribuída *CMB*, *Time Warp* e híbridos. O protocolo *Time Warp* é otimista, permitindo que os eventos sejam executados sem a preocupação que ocorram erros de causa e efeito. O Capítulo 3 descreve as principais características desse protocolo, seus mecanismos de controle local e global e apresenta algumas variantes.

Capítulo 3

Time Warp

3.1 Considerações Iniciais

Os protocolos otimistas detectam e recuperam erros de causa e efeito, ao invés de evitá-los. Quando um erro é detectado, a simulação distribuída é restaurada para um estado consistente. Uma vantagem dessa abordagem em relação aos protocolos conservativos consiste que a mesma permite explorar o paralelismo em situações onde erros poderiam ocorrer, mas que de fato não ocorrem [Fuj00].

O protocolo de sincronização *Time Warp* baseia-se em um paradigma chamado *virtual time*, proposto por Jefferson. Esse paradigma tem o objetivo de organizar e sincronizar os processos lógicos em uma computação distribuída, utilizando coordenadas de tempo e espaços virtuais [Jef85].

Esse protocolo possui dois mecanismos de controle, um local e outro global. O primeiro tem a finalidade de manter a execução de eventos (mensagens) em uma ordem cronológica, enquanto o segundo possui a principal finalidade de efetuar gerenciamento de memória na execução do *Time Warp*. O gerenciamento é necessário quando ocorre um erro, sendo que a simulação deve ser restaurada para um estado consistente. Para que isso seja possível, existem algoritmos para efetuar salvamento de estados e para restaurá-los.

O grande crescimento na área de simulação distribuída otimista tem proporcionado vários estudos envolvendo o uso do *Time Warp*. Pode-se citar estudos com o *Time Warp* utilizando computação em *grid* [Isk04], orientado a conexão (utilizando canais lógicos, como no *CMB*) [Kal04], estudos envolvendo técnicas para redução de tempo para acessos em listas de eventos futuros [Li04], dentre outros.

A Seção 3.2 apresenta o mecanismo de controle local, suas principais características e os algoritmos que o envolvem, como por exemplo algoritmos para salvamento de estados e algoritmos para a restauração de estados consistentes. A Seção 3.3 apresenta o mecanismo de controle global, o conceito do *Global Virtual Time* e algoritmos para o seu cálculo. A Seção 3.4 apresenta o conceito de *plugins*, descrevendo os cancelamentos agressivo e preguiçoso, cálculo do *GVT* de Samadi e de Mattern, os mecanismos de salvamento de estados periódicos (*sparse state saving*) e incremental (*incremental state saving*) e um protocolo probabilístico que limita o otimismo do *Time Warp*. A Seção 3.5 apresenta as considerações finais.

3.2 Mecanismo de Controle Local

O mecanismo de controle local é responsável pela execução dos eventos escalonados em ordem cronológica da marca de tempo. A comunicação no *Time Warp* é feita por troca de mensagens e cada mensagem (evento a ser executado em um processo lógico remoto) é composta por quatro informações: processo lógico emissor, o tempo virtual de envio (tempo virtual em que a mensagem é enviada), processo lógico receptor e o tempo virtual de recebimento (tempo de execução de uma mensagem). Existem duas regras fundamentais relacionadas aos sistemas de tempo virtual [Jef85]:

- o tempo virtual de envio de uma mensagem deve ser menor do que o seu tempo virtual de recebimento;
- o tempo virtual de um evento em um processo lógico deve ser menor que o tempo virtual do próximo evento no mesmo.

Como um processo lógico *Time Warp* não faz verificação de segurança para executar um evento, eventos contendo uma marca de tempo maior que de uma mensagem atrasada (*straggler*) podem ser executados incorretamente. Uma mensagem atrasada é definida como sendo um evento que chegou com marca de tempo menor que do relógio virtual do processo lógico. O *Time Warp* resolve esse problema através de *rollback*. Quando um *rollback* é executado, deve-se considerar as possíveis modificações nas variáveis de estado e os possíveis escalonamentos de novos eventos (envio de mensagens para outros processos lógicos). A Figura 3.1 ilustra a situação em que uma mensagem atrasada chega para ser executada por um processo lógico [Fuj00]. Na figura, os eventos com marcas de tempo 12, 21 e 35 já foram processados quando chega um evento com marca de tempo igual a 18. Isso significa que os eventos com marcas de tempo maiores que 18, que já foram executados, devem ser desfeitos para que a simulação distribuída retorne para um estado consistente.

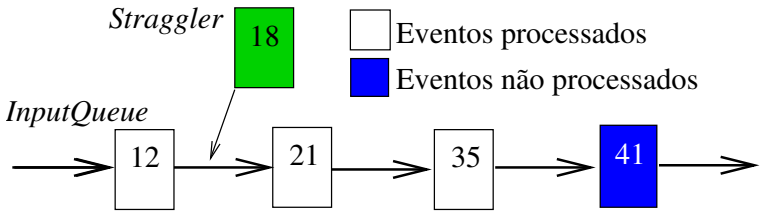


Figura 3.1: Mensagem atrasada.

A Figura 3.2 ilustra as principais estruturas de dados que compõem um processo lógico *Time Warp*, necessárias para o mecanismo de *rollback* e o funcionamento da simulação distribuída *Time Warp* [Jef85]:

- uma identificação do processo lógico, a qual deve ser única;

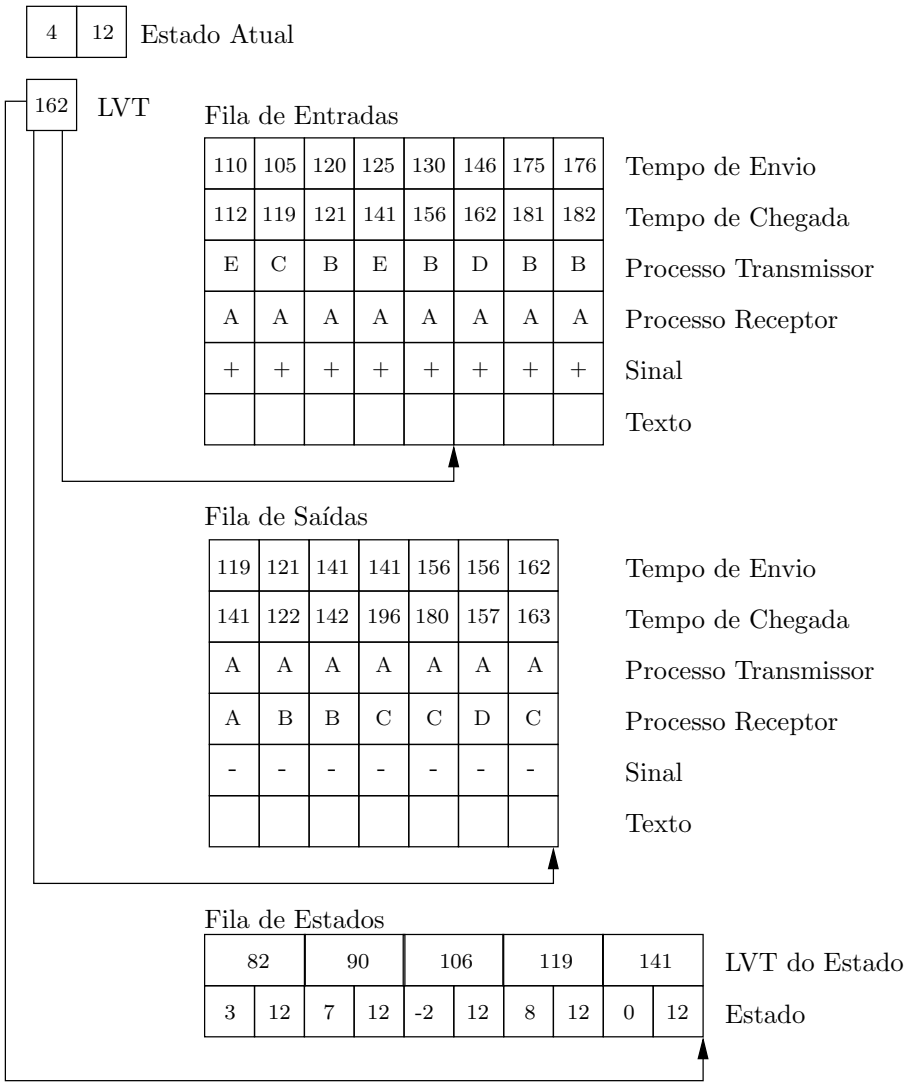


Figura 3.2: Um processo lógico no *Time Warp*.

- o *Local Virtual Time* (*LVT*) armazena o relógio virtual do processo lógico, o qual representa o tempo virtual do último evento executado pelo mesmo;
- um estado, armazenando todas as as informações do processo lógico, como por exemplo o *LVT*;
- uma fila de estados (*State Queue*), a qual armazena cópias dos estados mais recentes do processo. Essas cópias devem ser feitas periodicamente para ser possível efetuar o *rollback*;
- uma fila de entradas (*Input Queue*), contendo mensagens recebidas por um processo lógico e também eventos gerados pelo mesmo, ordenados pela marca de tempo. Cada

mensagem nesta fila mantém informações sobre o processo lógico emissor, sobre o processo lógico receptor, além de um sinal (positivo ou negativo, indicando uma mensagem ou uma antimensagem). Mesmo depois de executada, uma mensagem continua a fazer parte desta lista. Tais mensagens são consideradas mensagens positivas [Jef85];

- uma fila de saída (*Output Queue*), contendo cópias negativas das mensagens que um processo lógico enviou, denominadas antimensagens. Nessa lista, todas as mensagens tem marca de tempo menor ou igual ao *LVT*. Essas mensagens são utilizadas quando existe a necessidade de um *rollback*, pois caso um aconteça, um processo lógico deve enviar antimensagens a todos os processos lógicos remotos a quem esse enviou mensagens, com o objetivo de cancelar as mensagens enviadas erroneamente.

Uma situação especial no *rollback* é aquela na qual um processo lógico, que o sofreu, precisa avisar aos processos lógicos para os quais enviou mensagens, para que também desfaçam suas computações errôneas, causadas pelo envio de mensagens. O mecanismo de *rollback* é capaz de abordar tudo isso eficientemente, cancelando essas mensagens enviadas erroneamente. O cancelamento consiste no envio de antimensagens para outros processos lógicos, anulando essas mensagens enviadas.

As estratégias de cancelamento de mensagens são (na Seção 3.4, os cancelamentos agressivo e preguiçoso são abordados detalhadamente, pois são os mais utilizados):

- Cancelamento Agressivo: quando um processo efetua um *rollback* em um tempo t, antimensagens são imediatamente enviadas para todas as mensagens anteriormente enviadas contendo uma marca de tempo maior do que t [Fuj90];

- Cancelamento Preguiçoso: nesse mecanismo de cancelamento os processos não enviam antimensagens imediatamente após o início do *rollback*, eles esperam a verificação de que uma nova execução não gerará as mesmas mensagens. Se uma mesma mensagem for recriada, não existe a necessidade de cancelá-la [Fuj90];

- Cancelamento Dinâmico: técnica na qual cada processo decide qual estratégia irá utilizar, cancelamento agressivo ou preguiçoso [Wil97];

- Cancelamento Baseado em Lotes: esse mecanismo tem o propósito de melhorar o desempenho dos cancelamentos agressivo e preguiçoso e utiliza uma relação de ordenação parcial entre dois eventos. Para isso, é utilizado um novo tipo de mensagem na simulação *Time Warp*, chamada *CANCEL_MESSAGE*, a qual é capaz de cancelar múltiplos eventos em um mesmo processo lógico. Assim, o número de mensagens enviadas entre os processos lógicos pode ser reduzido e muitos *rollbacks* podem ser evitados [Zen04].

A Figura 3.3 [Fuj00] ilustra a situação onde uma mensagem atrasada acaba de chegar em um processo lógico. Os eventos com marcas de tempo 12, 21 e 35 já foram processados e seus estados foram salvos. Além disso, os eventos com marcas de tempo 12 e 35 geraram novos eventos para serem executados remotamente, isto é, enviaram mensagens a processos lógicos remotos. Isso faz com que ocorram inserções na *OutputQueue* do processo lógico. Assim,

quando a mensagem atrasada chega no tempo 18, um mecanismo de recuperação deve ser executado (*rollback*), de modo a cancelar as mensagens enviadas com marcas de tempo menores que da mensagem atrasada. Esse mecanismo de *rollback* deve desfazer todas as execuções erradas, marcando os eventos com marca de tempo maiores que da mensagem atrasada como ainda não executados, restaurar a simulação distribuída para um estado consistente (isto é, para um estado com marca de tempo menor que do *straggler*) e enviar antimensagens a todos os processos lógicos que geraram eventos para serem executados remotamente.

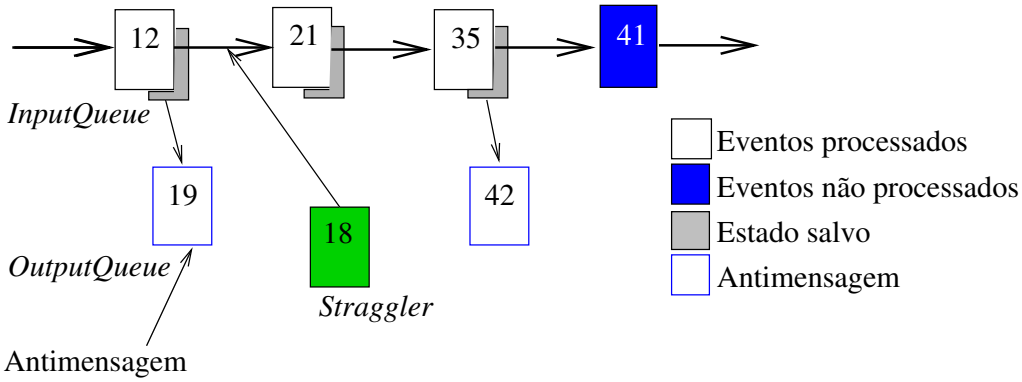


Figura 3.3: Processo lógico antes de um *rollback*.

A Figura 3.4 ilustra um processo lógico após a execução de um *rollback* [Fuj00]. Todos os eventos com marcas de tempo maiores que a marca de tempo da mensagem atrasada foram marcados como não executados, as antimensagens foram enviadas e a mensagem atrasada foi inserida na *Input Queue* do processo lógico.

Para desfazer as modificações nas variáveis de estado, quando um *rollback* é efetuado, é necessário salvar o estado através de cópias dessas variáveis. Como exemplo de variáveis de estados, utilizadas pelo *Time Warp*, pode-se citar o *LVT*, a lista de recursos do processo lógico, juntamente com suas filas e servidores, informações de armazenamento da *Input Queue* e da *Output Queue*, entre outras. O salvamento de estados pode ser efetuado das seguintes formas:

- *Copy State Saving*: faz uma cópia de todas as variáveis de estado que foram modificadas no processo lógico a cada avanço do *LVT*;
- *Incremental State Saving*: apenas as variáveis de estado que foram modificadas são salvas a cada avanço do tempo;
- *Sparse State Saving*: salva o estado periodicamente, introduzindo um *checkpoint* para forçar o processo lógico a registrar seu estado antes do incremento do *LVT*.

Salvar o estado através de cópias é mais eficiente quando muitas variáveis de estado são modificadas a cada evento. Por outro lado, se uma pequena porção dessas variáveis é modificada em cada evento, salvar o estado de maneira incremental é mais eficiente, pois trata as variáveis conforme a necessidade. Além disso, é possível fazer uma combinação entre o *incremental state*

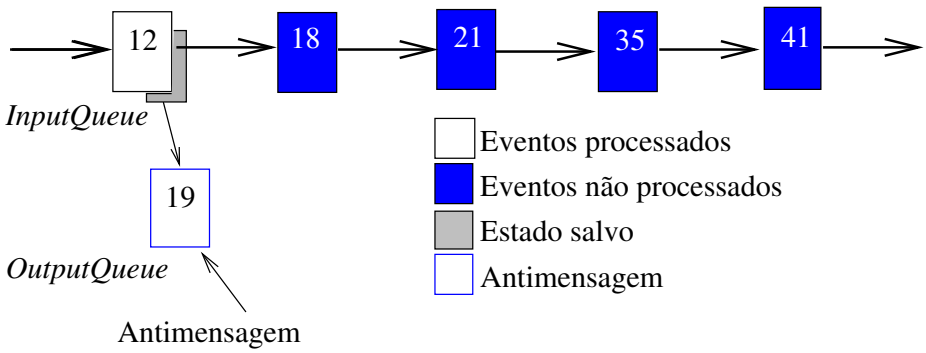


Figura 3.4: Processo lógico depois de um *rollback*.

saving e o *sparse state saving*, no qual apenas as variáveis de estado que sofreram modificações são salvas, apenas periodicamente e não a cada avanço do tempo. Os mecanismos *sparse state saving* e *incremental state saving* são apresentados detalhadamente na Seção 3.4.

No *Time Warp*, os processos lógicos trocam eventos entre si através do envio de mensagens. Quando uma mensagem é enviada, ela é inserida na lista de entrada do processo lógico receptor e uma cópia dessa mensagem é inserida na lista de saída do processo lógico emissor (antimensagem é inserida na *Output Queue*). Assim, para cada mensagem enviada, deve existir uma antimensagem que é praticamente igual à mensagem, exceto pelo sinal, pois em uma mensagem tem-se um sinal positivo e em uma antimensagem tem-se um sinal negativo [Fuj00].

Quando uma mensagem e sua antimensagem ocorrem na mesma fila, elas são imediatamente eliminadas. Deve-se então, considerar três casos quando um processo lógico recebe uma antimensagem [Fuj00]:

- se a mensagem positiva ainda não foi processada, mensagem e antimensagem podem ser eliminadas e removidas da fila de entrada;
- se a mensagem positiva já foi processada, um *rollback* é executado. Assim que o *rollback* for executado, ambas (mensagem e antimensagem) são eliminadas. Esse *rollback*, causado pelo recebimento de uma antimensagem, é conhecido como *rollback* secundário. O *rollback* resultante da mensagem inicial (*straggler*) é chamado de *rollback* primário;
- a mensagem positiva ainda não foi recebida pelo processo lógico. Isso acontece quando o meio de comunicação não pode garantir que as mensagens são recebidas na mesma ordem em que são enviadas. Nesse caso, a antimensagem chega antes que a mensagem. Se isso ocorre, a antimensagem é colocada na fila de entrada. Quando a mensagem positiva correspondente é recebida, ambas são eliminadas.

O mecanismo de antimensagens habilita o *Time Warp* a recuperar mensagens enviadas corretamente. Além disso, não existe possibilidade de *deadlock*, pois não existe bloqueio.

Um processo lógico no *Time Warp* funciona praticamente da mesma forma que um processo da simulação seqüencial, exceto por duas diferenças:

- os eventos, em um conjunto de eventos, podem ser gerados por processos lógicos remotos;
- o *Time Warp* não descarta os eventos já processados, esses são armazenados em uma fila de eventos executados, pois quando acontece um *rollback* pode existir a necessidade de alguns desses eventos serem re-executados.

3.3 Mecanismo de Controle Global

O mecanismo de controle local é suficiente para garantir que a simulação distribuída execute da mesma maneira que uma simulação seqüencial, isto é, executar a simulação em ordem cronológica da marca de tempo. Entretanto, existem dois problemas que devem ser analisados antes que o mecanismo de controle local seja viabilizado [Fuj00]:

- a cada criação de um novo evento um espaço de memória é alocado, assim, a computação pode consumir uma alta quantia de memória durante a execução. Como essa memória não pode ser liberada até que se tenha certeza de que a mesma não vai mais ser utilizada, um mecanismo para a recuperação de recursos de memória para eventos já processados, antimensagens e informações sobre variáveis de estado é necessário. Esta recuperação é denominada *fossil collection*;
- em certas computações executadas em um programa de simulação não pode ser efetuado um *rollback*. Por exemplo, uma vez que uma operação de Entrada/Saída (impressão) é executada, ela não pode ser desfeita.

O mecanismo de controle global resolve esses problemas apresentados. Por exemplo, como garantir que eventos com a marca de tempo menor que um tempo t não sofrerão *rollback*? Isto é, algum mecanismo poderia garantir que nenhum *rollback* será executado em um tempo simulado menor que um tempo t. Similarmente, operações de Entrada/Saída geradas por um evento com marca de tempo menor que t poderiam ser executadas sem maiores problemas. Além desses dois gerenciamentos, esse mecanismo também pode ser efetuado para detectar o fim de uma computação distribuída [Jef85].

O conceito fundamental nesse mecanismo é o *Global Virtual Time* (GVT), o qual corresponde ao menor valor calculado entre todos os *LVT's* dos processos lógicos na simulação distribuída, todas as marcas de tempo das mensagens enviadas e que ainda estão em trânsito e todas as marcas de tempo das mensagens reconhecidas, ou seja, o menor valor entre todos os mínimos locais dos processos lógicos da simulação distribuída [Fuj00]. O mínimo local em um processo lógico corresponde entre a menor marca de tempo dos eventos não executados, menor marca de tempo das mensagens em trânsito e menor marca de tempo das mensagens reconhecidas.

Assim, se acontece um *rollback* em um processo lógico no tempo t, apenas eventos com marca de tempo menor que o GVT não sofrem *rollbacks*. Portanto, as informações com marca de tempo menores que o GVT podem ser descartadas [Spo01].

O cálculo do GVT é uma tarefa difícil, pois existem mensagens/antimensagens circulando pela rede de comunicação. Essas mensagens/antimensagens devem ser consideradas no momento de se determinar qual a menor marca de tempo da simulação distribuída [Fuj00].

3.3.1 O Cálculo do GVT [Fuj00]

Pela definição de GVT, se for possível capturar rapidamente todos os eventos ainda não processados (e parcialmente processados) e antimensagens no sistema, com um tempo t, o cálculo do GVT pode ser feito trivialmente. Existem duas maneiras de efetuar o cálculo do GVT, síncrona e assíncrona.

O cálculo do GVT de modo síncrono traz o problema de mensagens em trânsito. Esse refere-se àquelas mensagens que foram enviadas por um processo lógico mas ainda não foram recebidas por outro. Como mostrado na Figura 3.5, tais mensagens poderiam resultar em *rollbacks*. Portanto, essas mensagens influenciam no cálculo do GVT, necessitando, assim, que sejam incluídas em seu cálculo para que não causem erros. Por exemplo, se a mensagem em trânsito com o tempo igual a 10 não é considerada, o GVT entre PL_1 e PL_2 é calculado incorretamente com o valor 15, quando deveria ser 10.

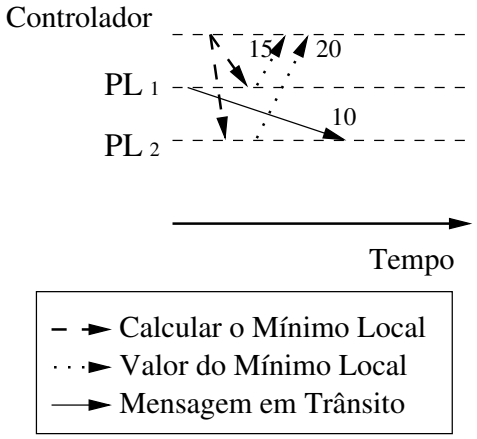


Figura 3.5: O problema de mensagens em trânsito.

Uma solução simples para este problema é o uso do reconhecimento de mensagens. A idéia principal deste mecanismo é ter certeza que toda mensagem em trânsito é contada por pelo menos um processador quando o GVT está sendo calculado. Se um aviso de reconhecimento é enviado para cada mensagem, o emissor de cada mensagem é responsável por contar com essa mensagem em seu mínimo local até que o aviso de reconhecimento seja recebido. A partir do momento que o receptor recebe a mensagem, essa é de responsabilidade dele.

Um algoritmo síncrono para o cálculo do GVT é descrito nos passos a seguir. O algoritmo usa um controlador central para iniciar o cálculo do GVT, calcula o mínimo global e relata para os outros processos o GVT calculado:

- o controlador envia um *broadcast* para os processadores com uma mensagem para iniciar a computação do GVT;
- com o recebimento desta mensagem, cada processo lógico pára sua execução do evento atual e responde ao controlador, indicando um recebimento para início do cálculo do GVT. O processo lógico fica bloqueado até receber uma outra mensagem do controlador;

- quando o controlador recebe todas estas mensagens, novamente ele envia um *broadcast* para que os processos lógicos iniciem o cálculo de seus mínimos locais;
- com o recebimento dessa mensagem, cada processo lógico calcula a marca de tempo mínima entre os eventos não processados e antimensagens, a marca de tempo mínima de alguma mensagem que um processador enviou mas ainda não recebeu um aviso de reconhecimento. Quando o cálculo chega ao fim, o processador envia seu mínimo local para o controlador;
- quando o controlador recebe todos os mínimos locais, ele calcula o mínimo global e envia um *broadcast* com o novo valor calculado do GVT.

Como pode ser observado no segundo passo do algoritmo, o intervalo entre as mensagens de iniciar a computação do GVT e a mensagem de calcular o mínimo em cada processador gera um bloqueio. Esse bloqueio causa uma desvantagem no cálculo do GVT, pois alguns processos lógicos podem atrasar no momento de responder, ao controlador, a mensagem de iniciar o cálculo. Isto acontece devido ao fato que o processador pode estar executando um evento quando a mensagem chega. Essa desvantagem causa o atraso de todos os processos lógicos no sistema, pois o controlador deve receber as mensagens de todos os processos lógicos antes de efetuar o terceiro passo.

O cálculo do GVT de modo assíncrono permite que a execução de eventos nos processos lógicos continue, enquanto o cálculo do GVT está sendo efetivado. Isso introduz um problema conhecido como problema da transmissão simultânea, o qual surge porque nem todos os processadores conseguem calcular seu mínimo local precisamente no mesmo instante do tempo de simulação, resultando que uma ou mais mensagens não processadas não serão contadas pelo processador que envia ou que recebe uma mensagem.

A Figura 3.6 descreve esse problema, no qual o controlador envia um *broadcast* para calcular o GVT. Um processador PL_1 recebe a mensagem e calcula seu mínimo local, relatando ao controlador um tempo de 35 unidades. A mesma mensagem enviada para PL_2 sofre atraso no meio de comunicação e chega algum tempo depois. Em seguida, PL_2 envia uma mensagem a PL_1 com uma marca de tempo igual a 30 e inicia o processamento de um evento com marca de tempo igual a 40. Neste ponto PL_2 recebe a mensagem para calcular seu mínimo local, relatando ao controlador que seu valor é 40, o qual é igual ao tempo de seu evento local. Assim, o controlador calculará um falso valor para o GVT (35), pois houve uma falha em uma mensagem com marca de tempo igual a 30. Algoritmos assíncronos serão abordados no Capítulo 4.

3.4 *Plugins* do *Time Warp*

Os protocolos de simulação distribuída conservativos e otimistas, mais especificamente o *CMB* e o *Time Warp*, têm sido largamente discutidos nos últimos anos. Pesquisadores optaram por desenvolver protocolos explorando as vantagens de ambos, inserindo características otimistas em um protocolo conservativo ou então limitando o otimismo de um protocolo otimista

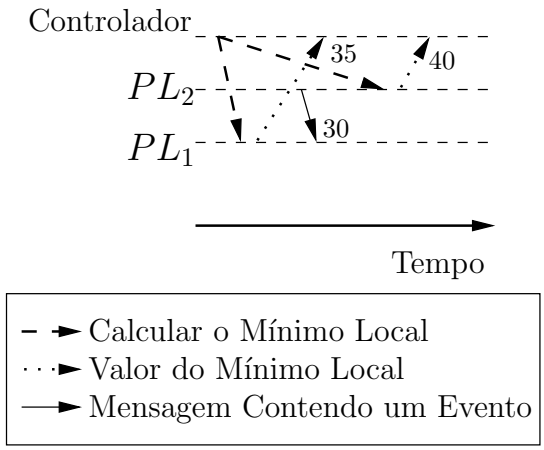


Figura 3.6: O problema de transmissão simultânea.

utilizando características conservativas, sendo que essa última opção denomina as variantes do protocolo *Time Warp*.

A taxonomia construída em [Spo01] e o estudo de algoritmos do *Time Warp* permitiram a utilização de domínios de *plugins*, onde cada domínio é responsável por uma atividade e cada elemento do domínio responsável pela implementação de um desses algoritmos (o *plugin*). Um *plugin* tem como objetivo permitir a alteração do modelo básico do *Time Warp* de forma a alterar seu comportamento, por exemplo, limitando seu otimismo. Desse modo, um usuário pode escolher entre utilizar o *Time Warp* básico ou uma variante para executar a simulação. Esta independência do usuário pode permitir, futuramente, a identificação de classes de aplicação para o *Time Warp* e suas variantes [Spo01].

Para esclarecer a utilização de *plugins*, a Figura 3.7 ilustra o *Time Warp* básico relacionando-se com *plugins*, através de pontos de conexão. O uso desses permite anexar ao modelo do *Time Warp* básico variantes do mesmo, podendo limitar o otimismo do modelo básico [Spo01]. Por exemplo, pode-se utilizar o *Time Warp* básico salvando os estados dos processos lógicos periodicamente e efetuando um cancelamento preguiçoso, ou ainda utilizar esse modelo básico salvando seus estados com o modo incremental e aplicando um cancelamento agressivo, etc.

As seções seguintes apresentam variantes do modelo básico do *Time Warp*, para o cancelamento de mensagens, cálculo do GVT, salvamento de estados da simulação distribuída e para limitar o otimismo do *Time Warp*.

3.4.1 Cancelamento Agressivo (*Agressive Cancellation*)

Nesse mecanismo, a execução do *rollback* envia antimensagens para cancelar todas as mensagens enviadas incorretamente, isto é, tão logo a simulação distribuída detecta o erro, as antimensagens são enviadas [Lin91, Raj95].

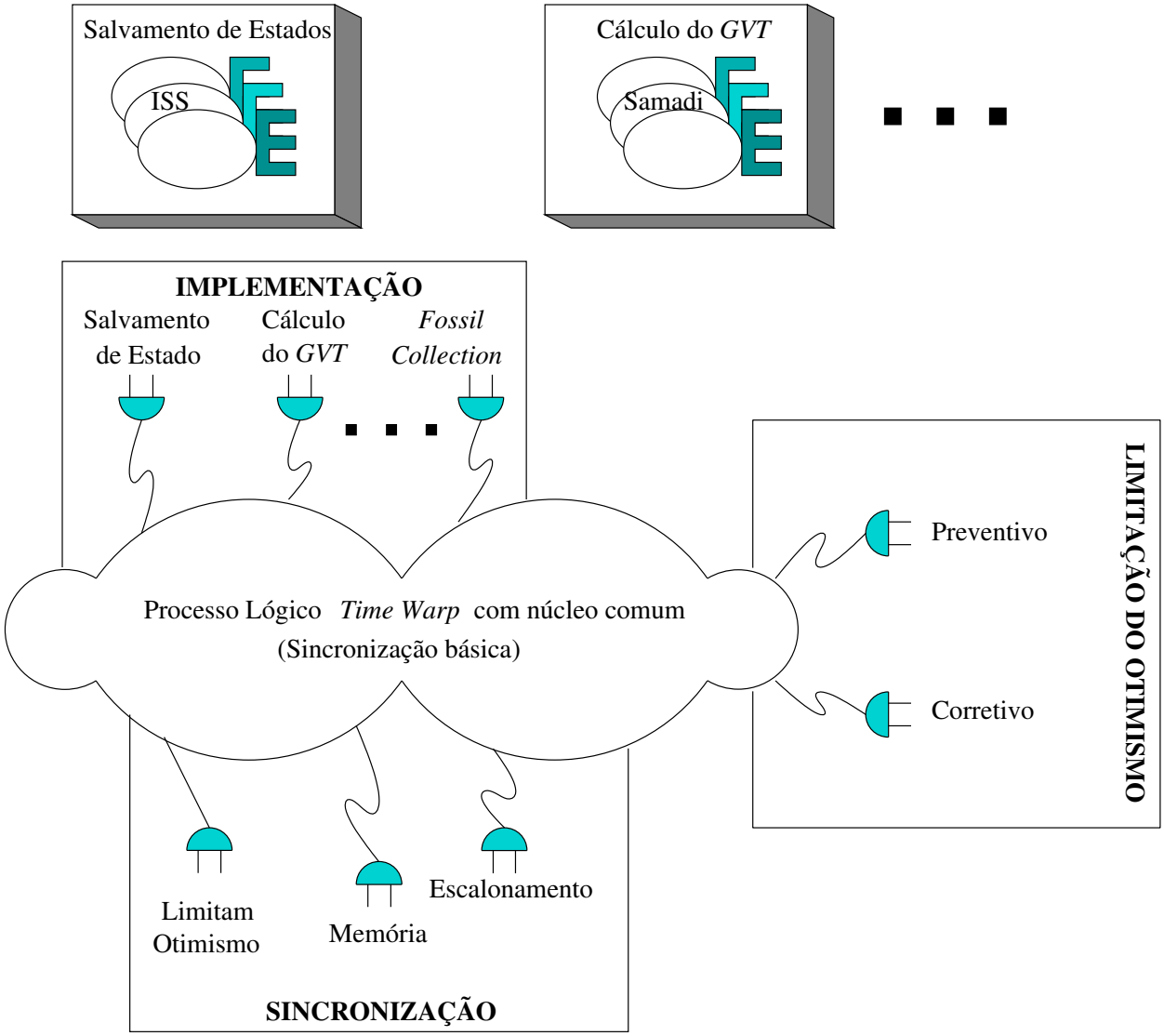


Figura 3.7: Relacionamento entre o modelo básico e o Domínio de *Plugins*.

A Figura 3.8 ilustra um exemplo do cancelamento agressivo. Na Figura 3.8(a), o processo lógico A recebe uma mensagem atrasada após ter enviado uma mensagem ao processo lógico B, resultando na execução incorreta do evento enviado para o processo lógico B. Na Figura 3.8(b), A cancelou a mensagem incorreta e enviou a B a versão correta da mensagem. Isso significa que B já executou a mensagem incorreta e, nesse momento, A sofreu *rollback* e enviou a antimensagem para B. Na Figura 3.8(c) A executa corretamente e B executou um *rollback* secundário e está esperando pela mensagem correta, vinda de A. Assim que o processo lógico B recebe a mensagem correta, essa é inserida em sua *InputQueue* (Figura 3.8(d)) [Rei90a].

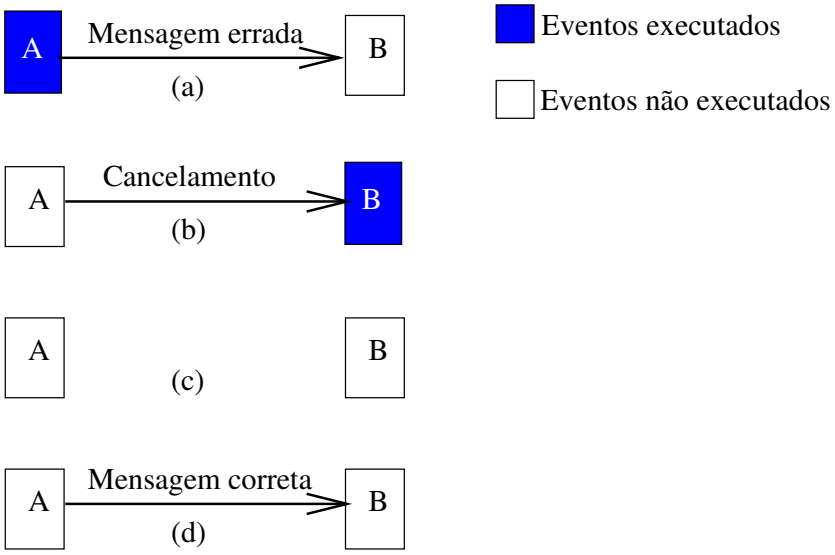


Figura 3.8: Cancelamento agressivo.

3.4.2 Cancelamento Preguiçoso (*Lazy Cancellation*)

O cancelamento preguiçoso, ao contrário do agressivo, não envia antimensagens imediatamente na execução de um *rollback*. Após sofrer um *rollback* sem enviar antimensagens, o processo lógico inicia a re-execução de seus eventos. Quando o evento é re-executado o conteúdo da mensagem original é comparado com o da mensagem correta. Se essas duas mensagens são iguais, nada é efetuado, caso contrário, uma antimensagem é enviada para cancelar a mensagem original e a mensagem correta é enviada para o processo lógico remoto [Lin91, Raj95].

A Figura 3.9 ilustra o funcionamento do cancelamento preguiçoso. O processo lógico A recebe uma mensagem incorreta. Como resultado dessa execução incorreta o mesmo envia uma mensagem incorreta para o processo lógico B, como ilustrado na Figura 3.9(a). Na Figura 3.9(b), A recebe o cancelamento da mensagem incorreta e não cancela a mensagem incorreta enviada para B. Ao invés disso, o processo lógico A executa o evento novamente e B continua a executar a mensagem incorreta. Na Figura 3.9(c) A termina de executar o evento e vai realizar o envio de uma nova mensagem. Essa nova mensagem é comparada com a mensagem original (incorreta). A percebe que as duas mensagens são diferentes e envia uma antimensagem a B e uma cópia da mensagem correta. Finalmente, na Figura 3.9(d), B cancela a mensagem incorreta, anteriormente executada, e inicia a execução da mensagem correta [Rei90a].

3.4.3 Cálculo do *GVT* de Samadi

Nesse algoritmo, cada processo lógico tem um modo de operação, *Normal* ou *Find*. Além disso, o algoritmo assume que mensagens de reconhecimento devem ser utilizadas para todas

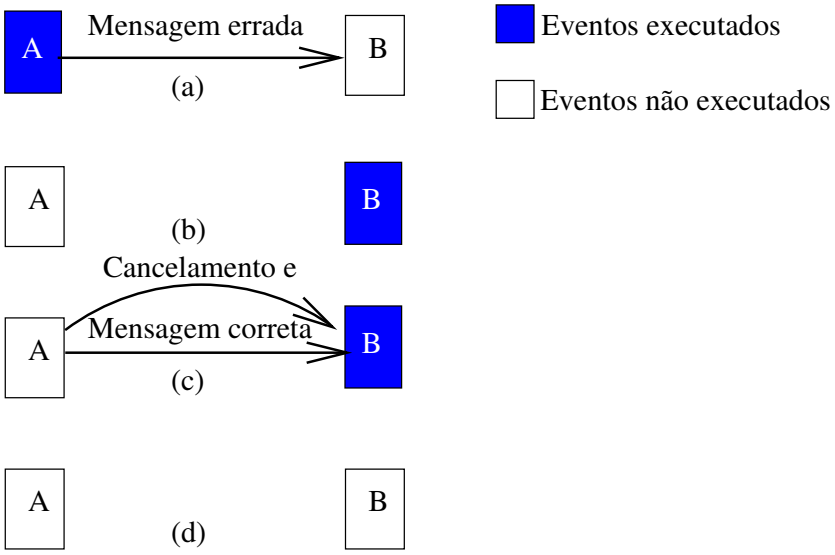


Figura 3.9: Cancelamento preguiçoso.

as mensagens circulando entre os processos lógicos. Desse modo, o processo lógico emissor de uma mensagem torna-se responsável por considerar cada mensagem que enviou, até receber um reconhecimento da mesma.

O mínimo local em cada processo lógico consiste na menor marca de tempo dos eventos não processados ($LVTs$) e antimensagens no processo lógico e as marcas de tempo de suas mensagens em trânsito, ou seja, marcas de tempo de mensagens que o mesmo enviou a outro processo lógico mas ainda não recebeu um reconhecimento. O problema das mensagens em trânsito, discutido na Seção 3.3, é resolvido considerando essas mensagens de reconhecimento do algoritmo. Devido à aplicação dessa solução, são necessárias três estruturas de listas, uma de mensagens enviadas por um processo lógico mas ainda não recebidas, lista de mensagens não reconhecidas e uma lista de mensagens reconhecidas pelo processo lógico.

Para que o cálculo do GVT seja iniciado, um dos processos lógicos da simulação deve ficar responsável por coordená-lo. A idéia mais simples para resolver isso é definir, explicitamente, um processo lógico coordenador permanente. Outra alternativa é aplicar um algoritmo para a definição de um coordenador [Lyn96, Sam85].

O cálculo é iniciado com o processo lógico coordenador enviando um *broadcast* a todos os processos lógicos, para que iniciem os cálculos de seus mínimos locais. Conforme os processos lógicos recebem essa informação, iniciam automaticamente os cálculos de seus mínimos locais. Após esse cálculo, cada processo lógico envia seu mínimo local ao coordenador e assim que esse recebe os mínimos locais, calcula o novo valor do GVT [Fel03, Sam85].

Nesse algoritmo, as marcas de tempo das mensagens reconhecidas são incluídas na computação do mínimo local do processo lógico para prevenir problemas que possam surgir, como o ilustrado na Figura 3.10 [Sam85]. Os marcadores verticais T_1 e T_2 indicam o instante de tempo em que os processos receberam a mensagem Calcule_Seu_Mínimo_Local e logo em

seguida devolveram esses mínimos. Note que o processo PL_2 recebe um reconhecimento referente a mensagem com tempo t_1. Porém, o reconhecimento é enviado depois que o processo PL_1 devolve seu mínimo local. Se o processo PL_2 considera somente as mensagens não reconhecidas, não leva em conta o tempo t_1 e o resultado do GVT fica incorreto.

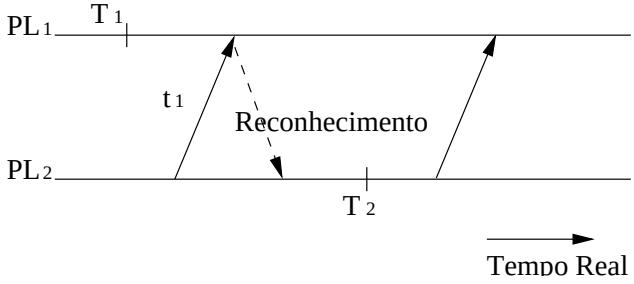


Figura 3.10: Exemplo de um resultado incorreto.

O problema ocorre porque os processos lógicos não devolvem seus mínimos locais ao mesmo tempo, e o processo lógico que envia o reconhecimento da mensagem de tempo t_1 o faz depois de ter calculado seu mínimo local. Embora o reconhecimento seja recebido pelo processo lógico PL_2, a sua marca de tempo não é considerada por ele no cálculo do seu mínimo local. Para evitar esse problema, os processos lógicos precisam saber se o reconhecimento para uma mensagem é enviado antes ou depois do cálculo do mínimo local do processo lógico receptor da mensagem. Se o reconhecimento é enviado depois do cálculo do mínimo local desse processo lógico receptor (PL_1), o processo lógico emissor da mensagem (PL_2) ainda é responsável por considerar essa marca de tempo.

Um reconhecimento que carrega o modo *Normal* informa ao processo lógico que o recebe, que sua marca de tempo é considerada pelo processo que envia o reconhecimento. Porém, um reconhecimento que leva o modo *Find* requer que o processo lógico que envia a mensagem (PL_2) inclua o tempo dessa mensagem no cálculo do seu mínimo local.

3.4.4 Cálculo do *GVT* de Mattern

O algoritmo para cálculo do GVT de Mattern foi proposto para resolver o problema das mensagens em trânsito.

O algoritmo GVT de Mattern também é assíncrono, pois evita sincronizações globais, mas não requer mensagens de reconhecimento. A idéia básica do algoritmo é dividir a computação distribuída por dois cortes que a separam em cores, como sendo o passado e o futuro da simulação [Mat93].

Como ilustrado na Figura 3.11, cada processo lógico define um ponto em sua execução chamado de ponto de corte. Todas as ações (computações, mensagens enviadas e mensagens recebidas) antes do ponto de corte do processo lógico são definidas como sendo o passado do processo lógico, e todas as ações ocorrendo depois do ponto de corte são definidas como sendo o futuro do processo lógico. O conjunto de pontos de corte através de todos os processos lógicos define um corte da computação distribuída. Um corte consistente é definido como um corte

no qual não há nenhuma mensagem que foi enviada do futuro do processo lógico emissor e recebida no passado do processo lógico receptor, isto é, se no corte da computação distribuída existirem apenas mensagens cruzando o ponto do passado para o futuro. A Figura 3.11(a) ilustra um corte consistente, enquanto a Figura 3.11(b) ilustra um corte inconsistente [Fuj00].

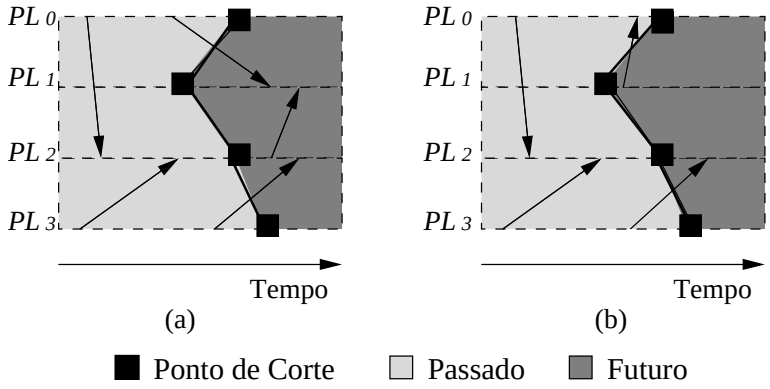


Figura 3.11: Cortes dividindo a computação distribuída em passado e futuro

Um algoritmo GVT assíncrono não precisa construir um corte consistente. Pode simplesmente ignorar todas as mensagens que fariam o corte inconsistente. Essas podem ser ignoradas porque precisam ter uma marca de tempo maior do que o GVT calculado, usando o corte consistente. A marca de tempo de qualquer mensagem/antimensagem enviada por um processo lógico depois de seu ponto de corte no tempo global T, deve ser pelo menos tão grande quanto o mínimo da [Fel03, Mat93]:

- menor marca de tempo de qualquer evento não processado no processo lógico no tempo T;
- a menor marca de tempo de qualquer mensagem recebida pelo processo lógico depois do tempo T.

Para resolver o problema de mensagens em trânsito nesse algoritmo, foi proposto que o trecho da computação distribuída imediatamente anterior ao primeiro corte assume a "cor branca", ou seja, tudo o que cada processo lógico realizar antes do seu primeiro ponto de corte tem a "cor branca"e, logo após o primeiro corte, assume a "cor vermelha". Mensagens em trânsito são definidas como as mensagens brancas que foram recebidas por processos lógicos vermelhos. Dessa forma, sempre que um processo lógico vermelho recebe uma mensagem branca, uma cópia dessa mensagem é enviada para o processo lógico coordenador. Portanto, apenas as mensagens que realmente estão em trânsito têm a necessidade de enviar um reconhecimento para o processo coordenador [Mat93].

Um problema com esse método descrito é determinar quando esse reconhecimento de mensagens chega ao fim. O processo coordenador recebe uma cópia de todas as mensagens em trânsito, mas não tem como determinar quando recebeu a última. Para resolver esse problema, torna-se necessário a utilização de um algoritmo de detecção de fim de computação distribuída,

onde a parte de "cor branca"da computação distribuída é considerada concluída quando nenhum processo lógico está branco e não há mais nenhuma mensagem branca em trânsito. Para a implementação dessa detecção, deve-se utilizar um contador em cada processo lógico que armazena o número de mensagens que esse processo enviou para qualquer um dos processos lógicos, menos o número de todas as mensagens recebidas. Armazenando esses contadores, juntamente com os instantes locais, o processo lógico coordenador pode ter uma visão consistente da quantidade de mensagens em trânsito. Assim, sabe-se quantas mensagens estão em trânsito no momento do corte e, com o número de cópias de mensagens recebidas, pode-se determinar o fim do algoritmo de instante local [Mat93].

Com o fim do algoritmo de instante local, todos os processos estão vermelhos e não há mais nenhuma mensagem branca em trânsito.

Para o cálculo do GVT, deve-se determinar dois cortes na computação distribuída, C_1 e C_2, sendo que C_2 é feito após C_1, como ilustrado na Figura 3.12 [Fuj00].

Para tanto, um algoritmo que simula uma topologia em anel pode ser utilizado. Inicialmente, o processo lógico coordenador i determina seu corte C_1 e envia uma mensagem de controle para o processo lógico $(i + 1)$ mod n do anel, onde n é o número de processos lógicos na simulação distribuída, para que esse possa também determinar seu corte C_1, e assim sucessivamente. Quando a mesma mensagem chega ao processo coordenador, tem-se o primeiro corte da simulação definido em cada um dos processos lógicos. A Figura 3.13 ilustra o funcionamento de um algoritmo que simula uma topologia anel, e o Algoritmo 1 apresenta o pseudo-código [Fel03, Mat93].

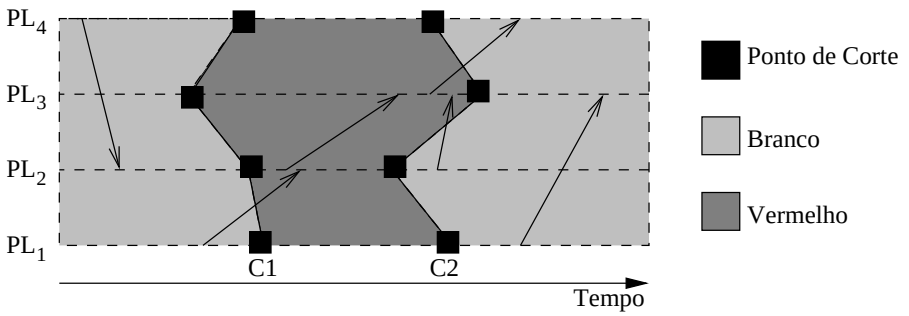


Figura 3.12: Exemplo de computação distribuída dividida em dois cortes.

Algoritmo 1 Determina o corte C_1

Para $i = 0$ até n **Faça**
  Determina o corte C_1;
  Armazena todos os tempos do processo lógico;
  Envia mensagem de controle para $(i + 1)$ mod n;
Fim Para

O objetivo do algoritmo é calcular o valor de GVT durante o corte C_2. Para isso, é necessário determinar o menor dos valores dos tempos locais e o menor valor das marcas de tempo de todas as mensagens que cruzam C_2. As mensagens que podem tornar o corte C_2

inconsistente, ou seja, enviadas por um processo lógico após C_2 e recebidas antes de C_2, podem ser ignoradas. As mensagens que cruzam o corte denotado por C_2 podem ser divididas em dois grupos [Mat93]:

- mensagens que foram enviadas entre C_1 e C_2;
- mensagens que foram enviadas antes de C_1.

Armazenando a quantidade de mensagens com "cor branca"recebidas, é possível assegurar se todas essas mensagens foram recebidas antes do corte C_2 ser determinado. Para isso, compara-se o número de mensagens que foram recebidas entre os cortes com o número de mensagens que os processos armazenaram como sendo de "cor branca"e que cruzam o corte C_1. Se esses valores forem diferentes, o resultado provavelmente será inválido e outra rodada deve ser iniciada para calcular o GVT. Ou seja, o algoritmo determina que enquanto todas as mensagens de "cor branca"não forem recebidas, o corte C_2 não deve ser feito.

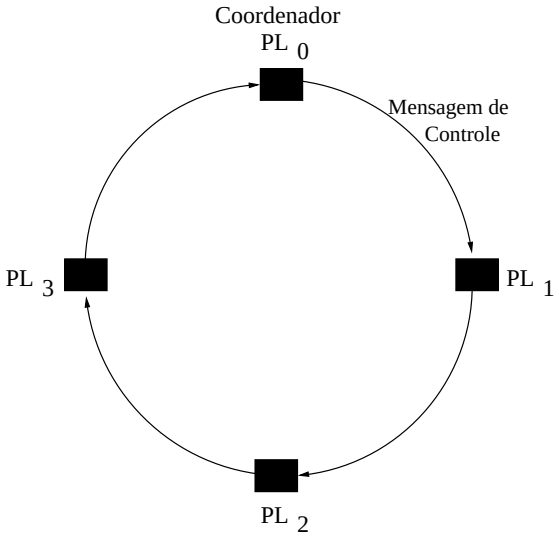


Figura 3.13: Determina o corte: Topologia Anel.

A solução proposta por Mattern é baseada no algoritmo de Chandy e Lamport [Cha85]. Para essa solução devem ser feitas duas importantes considerações [Mat93]:

- uma mensagem enviada por um processo lógico PL_i para um outro processo lógico PL_{i+1} depois que o processo lógico emissor recebeu a mensagem para mudar sua cor;
- uma mensagem enviada por um processo lógico PL_i para o processo lógico PL_{i+1} após este ter recebido a mensagem para mudar sua cor.

No primeiro caso foi proposto o seguinte: um processo lógico passa para a cor vermelha no exato momento em que ele armazena seu estado local, sendo assim todas as mensagens enviadas por esse processo lógico a partir desse instante irão "carregar"a cor do processo lógico emissor.

Portanto, quando o processo lógico PL_2, que estará branco, receber a mensagem vermelha, ele saberá que já deveria ter recebido a mensagem de controle. Desse modo, quando o processo lógico PL_2 receber a mensagem vermelha, ele também passará a ser vermelho e fará todas as ações que devem ser feitas para essa ocorrência (salvamento de estado, propagação da mensagem de controle). Quando a mensagem de controle for recebida pelo processo lógico PL_2 será simplesmente ignorada [Mat93].

Para o segundo caso, o problema causado é o de quando a última mensagem de "cor branca"que está em trânsito chega ao processo lógico receptor. A solução deste caso está em garantir que não existem mais mensagens de "cor branca"em trânsito. Para isso torna-se necessário que todos os processos lógicos armazenem o número de mensagens de "cor branca"enviadas para cada processo lógico. As soluções para os dois casos descritos podem ser combinadas "anexando"o número de mensagens de "cor branca"enviadas pelo processo lógico em suas mensagens de "cor vermelha".

Para se obter um estado global da simulação distribuída, e assim, poder calcular o valor do GVT, deve-se armazenar todos os instantes locais dos processos lógicos conseguidos através do algoritmo de instante local descrito. Garante-se que um processo lógico pode enviar o valor de seu instante local, assim que esse processo lógico tiver recebido todas as mensagens de "cor branca"que estavam em trânsito. Com isso, ele pode enviar uma mensagem para o processo lógico que lhe enviou a mensagem de controle ou a mensagem vermelha contendo seu instante local. Por sua vez, esse processo lógico armazena o instante local recebido e envia uma mensagem com seu instante local e o instante local recebido anteriormente. Percebe-se que em algum momento a mensagem com todos os instantes locais chegará ao processo lógico coordenador.

3.4.5 *Sparse State Saving*

O algoritmo *sparse state saving*, ou ainda *infrequent state saving*, é baseado na observação de que um estado da simulação distribuída pode ser recriado através de um estado anterior, reexecutando-se os eventos entre esses dois estados. Desse modo, os estados na simulação distribuída não precisam ser salvos a cada execução de um evento, reduzindo o *overhead* de memória para armazenar esses estados [Sko96].

Para assegurar a restauração dos estados e que a simulação distribuída progrida normalmente, é necessário que pelo menos um estado, antes do evento que causa um *rollback* (o *straggler*), seja salvo, como pode ser ilustrado pela Figura 3.14 [Sko96]. Quando chega um *straggler*, o estado salvo no tempo X deve ser restaurado e um *coast forward* ("deslizamento"para frente) deve ser efetuado até a marca de tempo do evento que causou o *rollback*. Assim, o progresso da simulação em um processo lógico pode ser dividido em três fases:

- Fase de execução (*forward execution phase*): corresponde à execução da simulação distribuída;
- Fase de *rollback* (*rollback phase*): corresponde à execução de *rollback*, cancelando mensagens e restaurando estados incorretos;

- Fase de "deslizamento"(*coast forward phase*): corresponde à execução dos eventos entre o último estado salvo e a mensagem atrasada.

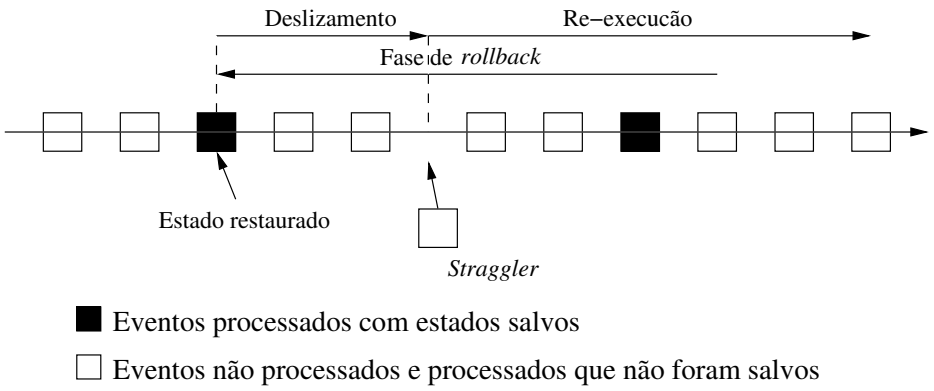


Figura 3.14: Simulação com $\chi = 5$, onde χ é o intervalo de checagem do processo lógico.

Dentre os algoritmos *sparse state saving* existentes, o mais simples é aquele que usa um intervalo fixo para o salvamento de estados, isto é, um salvamento de estados estático. O estado da simulação distribuída sempre é salvo, em cada processo lógico, a cada χ eventos executados [Lin93, Qua98].

Os algoritmos adaptativos surgiram para, dinamicamente, calcular o valor desse intervalo de *checkpoint*. O primeiro algoritmo, proposto por [Rön94] é baseado em um modelo analítico do tempo de execução de um processo lógico. O segundo algoritmo, proposto por [Fle95], mede o tempo para executar N eventos e compara essa medida com a medida calculada anteriormente. Se o tempo de execução aumenta significativamente, o intervalo de *checkpoint* é decrementado de uma unidade, caso contrário, é incrementado de uma unidade [Qua98].

O algoritmo de Rönngren [Rön94] altera o valor do intervalo de *checkpoint* de acordo com o comportamento do *rollback*. Para cada processo lógico, os parâmetros utilizados são: o número de eventos processados, incluindo os que já sofreram *rollback* (R_{obs}) e o número de *rollbacks* (K_{obs}) durante um período de observação. Assim, é possível calcular o intervalo de *checkpoint* χ_{min}, como pode ser visto na Equação 3.1, onde δ_s é o tempo médio necessário para salvar um estado na simulação distribuída e δ_c é o tempo de execução média de um evento na fase *coast forward*:

$$\chi = \lceil \sqrt{2\frac{R_{obs}\delta_s}{k_{obs}\delta_c}} \rceil \tag{3.1}$$

O algoritmo de Fleischmann e Wilsey [Fle95] é baseado na Equação 3.2, onde $C_{ss,n}$ e $C_{cf,n}$ representam o tempo gasto de UCP (Unidade Central de Processamento) para salvar um estado em um processo lógico e as operações de *coasting forward* durante o período, respectivamente:

$$E_{c,n} = C_{ss,n} + C_{cf,n} \tag{3.2}$$

Esse algoritmo inicia o valor de χ com 1 ($\chi_{initial} = 1$). Após isso, sucessivas observações são feitas, calculando o valor da função $E_{c,n}$. Se $E_{c,n}$ não aumenta significativamente, χ é incrementado de uma unidade. Se o valor de $E_{c,n}$ em um período corrente é muito maior que o valor anterior da mesma, χ é decrementado de uma unidade [Fle95, Qua98].

3.4.6 *Incremental State Saving*

Em algumas situações envolvendo a simulação distribuída, o número de variáveis de estados utilizadas pode ser muito alto. Além disso, a cada avanço no tempo de simulação, provocado pela execução de um evento, apenas uma fração dessas variáveis de estados necessitam ser salvas, tornando o salvamento de estados (salvando todas as variáveis) ineficiente e às vezes impraticável.

O *incremental state saving* aborda justamente o problema descrito, isto é, apenas as variáveis de estados que sofrem alterações são inseridas na fila de estados (*StateQueue*), isto é, um *log* registra as mudanças ocorridas nas variáveis de estado a cada execução de um evento e cada registro nesse *log* indica [Fuj00]:

- o endereço da variável de estado que foi modificada;
- o valor dessa variável de estado antes da modificação.

A maior vantagem na utilização do *incremental state saving* está no fato de que ambos os custos de sua execução e de memória estão relacionados com a execução de eventos, ou seja, o número de operações envolvendo a modificação de uma variável de estado tende a ser uma pequena fração de todas as operações durante essa execução [Gom96].

Quando ocorre um *rollback*, com a chegada de um *straggler*, a simulação distribuída restaura o processo lógico para um estado consistente. Para isso, os estados antigos são restaurados e os valores das variáveis de estado mais recentes são substituídos por esses valores antigos [Rön96].

Para desfazer as modificações nas variáveis de estado ocorridas devido à execução de um evento, o processo lógico *Time Warp* percorre a fila de estados salvos de trás para frente, já que essa está ordenada por suas marcas de tempo. Para cada evento sendo desfeito, o *Time Warp* verifica os estados salvos e restaura as variáveis modificadas. A Figura 3.15 ilustra o uso do *incremental state saving* [Fuj00].

Na Figura 3.15, o estado salvo para o evento E_{21} contém um registro com o endereço da variável X com o valor 1 e o evento E_{35} contém um registro com os endereços das variáveis X, com o valor 4 e Z, com o valor 3. Quando esses eventos são desfeitos, essas variáveis de estado são restauradas com seus valores antes da execução do evento E_{21}, ou seja, com os valores 1 e 2, respectivamente [Fuj00].

3.4.7 Protocolo Probabilístico para Sincronização Otimista

O *PDSP* (*Probabilistic Distributed Discrete Event Simulation Protocol*) pode ser considerado um *plugin* do protocolo *Time Warp* e baseia-se em análises probabilísticas para prever

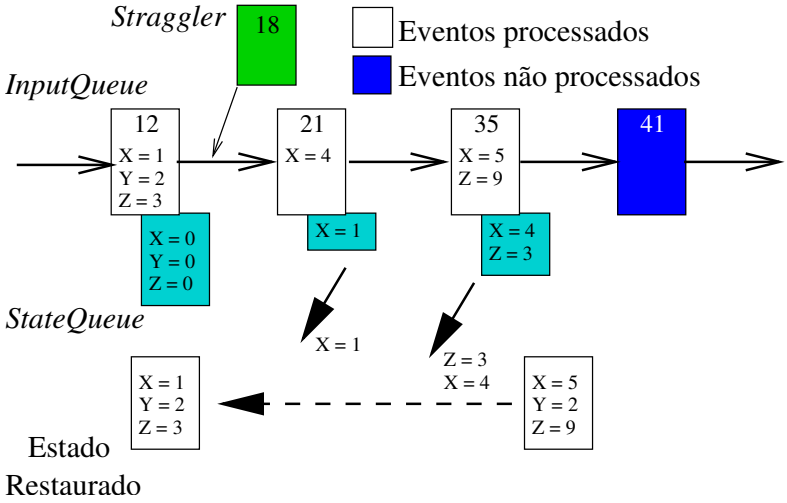


Figura 3.15: *Incremental State Saving.*

o comportamento futuro dos processos lógicos, para tomar decisões sobre a sincronização dos mesmos.

O objetivo desse protocolo é reduzir o número de mensagens de sincronização, evitando ao máximo a ocorrência de *rollbacks*. Consiste em um esquema de janelas cujos tamanhos variam dinamicamente, dependendo do paralelismo inerente ao modelo (*Local Virtual Time Windows - LVTW*). O protocolo adapta probabilisticamente o tamanho da *LVTW*, de acordo com a variação das marcas de tempo dos eventos que chegam [Fer94].

Esse protocolo reduz a interação entre os processos lógicos causada por *rollbacks*. Como vantagens, tem-se [Fer95]:

- o protocolo explora localmente a probabilidade da mensagem que chega se uma mensagem atrasada;
- leva em conta as características arquiteturais da plataforma, como velocidade da UCP e latência de comunicação;
- é adaptativo.

3.5 Considerações Finais

A utilização da simulação distribuída trouxe benefícios, pois pode ser utilizada no estudo de desempenho de sistemas complexos nos quais a simulação seqüencial é impraticável, pode ser executada em quaisquer regiões no mundo executando em arquiteturas diferentes e pode resolver vários problemas do mundo real. Os protocolos desenvolvidos para a simulação distribuída fazem com que essa seja executada de maneira correta, com o intuito de diminuir o tempo de processamento, ou seja, impedindo computações errôneas.

Para os propósitos da simulação distribuída, as estruturas de dados utilizadas na simulação seqüencial tiveram que ser adaptadas, surgindo protocolos de sincronização, classificados como

conservativos e otimistas, para garantir que a execução da simulação distribuída se comporte da mesma maneira que a simulação seqüencial.

O protocolo otimista *Time Warp* é dividido nos mecanismos de controle local e de controle global. O controle local é responsável pela execução da simulação em uma ordem não decrescente da marca de tempo, permitindo a execução de uma simulação seqüencial em cada processo lógico. O mecanismo de controle global utiliza o conceito do GVT para efetuar gerenciamento de memória e para a determinação do término da computação distribuída.

O avanço nos estudos relacionados à simulação distribuída proporcionou o surgimento de variantes dos seus protocolos que permitem a inserção de características otimistas em protocolos conservativos e características conservativas em protocolos otimistas. A taxonomia serviu para organizar as variantes, de forma que se possa escolher entre classes de variantes, não entre protocolos em si. Essa também permitiu identificar *plugins*. Um *plugin* tem por objetivo, depois de anexado ao núcleo básico `ETW`, alterar o seu comportamento. Como exemplo desses, pode-se citar *plugins* para o salvamento de estados, cálculo do GVT e algoritmos para o escalonamento de eventos [Spo01].

O Capítulo 4 descreve o `Basic Extensible Time Warp Kernel - ETW`, apresentando o modelo básico do protocolo, as principais estruturas de dados e os *plugins* implementados.

Capítulo 4

Implementação e Validação do ETW

4.1 Considerações Iniciais

Conforme o modelo de *plugins* apresentado no Capítulo 3, este capítulo descreve o ETW, organizado em núcleo básico e *plugins*. O ETW foi implementado com base na especificação desenvolvida para representar os algoritmos que definem as atividades executadas por um processo lógico *Time Warp*, seus parâmetros e variantes, apresentada em [Spo01]. Os algoritmos envolvem a recepção e execução de mensagens e antimensagens, o processo de salvamento de estados, a tarefa de cancelamento de mensagens e o mecanismo de *rollback*.

O ETW é composto por um conjunto de N processos (programa em execução) que interagem entre si através da troca de mensagens. Na Figura 4.1 é ilustrada uma visão geral do ETW, dividindo a simulação em N processos, no qual cada processo executa um processo lógico. Cada processo é responsável por uma única simulação seqüencial, isto é, cada processo é responsável pelo escalonamento, requisição e liberação de seus eventos. A troca de mensagens entre os processos é realizada pela utilização da biblioteca *LAM-MPI*. Se um processo deseja escalonar um evento em um processo remoto, uma mensagem/evento deve ser escalonada remotamente e uma função de escalonamento remoto fica responsável por enviar a mensagem/evento ao processo remoto [Lam04, Sni96].

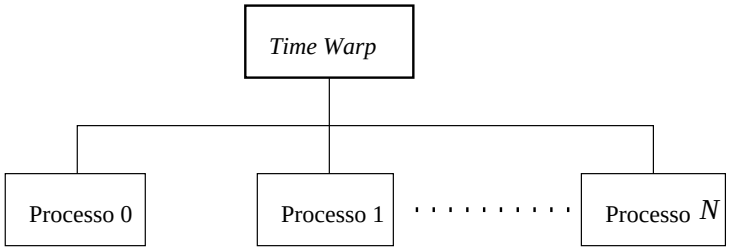


Figura 4.1: Visão geral do ETW.

O ETW foi implementado na linguagem de programação C++ utilizando o compilador gcc. Embora boa parte desenvolvida seja orientada a objetos, as funções de interface com o usuário

foram todas desenvolvidas utilizando uma linguagem estruturada C. Isso deve-se ao fato que nem todos os usuários de simulação estão habituados com a orientação a objetos.

A Figura 4.2 ilustra o relacionamento das estruturas de dados utilizadas no ETW com as atividades executadas pelo mesmo. A *Input Queue* de um processo lógico é implementada como uma estrutura de dados fila, que contém eventos a serem executados ou que já foram executados (utilizados por um possível *rollback*). Um evento pode ser gerado dentro do processo lógico ou por um processo lógico remoto. Quando um processo lógico recebe uma mensagem (evento), enviada por um processo lógico remoto, essa é inserida na *Input Buffer* do processo lógico para posterior tratamento de mensagens recebidas. Para cada evento retirado da *Input Queue* para execução, um estado deve ser salvo na *State Queue*, já que o mecanismo utilizado pelo ETW para salvamento de estados é o *Copy State Saving*. Além disso, quando um evento é executado, esse pode gerar ou não novos eventos para serem executados local ou remotamente. Quando um novo evento é gerado para ser executado remotamente, o mesmo deve ser enviado a outro processo lógico em forma de mensagem. Além disso, para todo evento enviado a um processo lógico remoto, em forma de mensagem, existe uma antimensagem correspondente, a qual deve ser inserida em uma fila de mensagens enviadas, ou seja, a *Output Queue*.

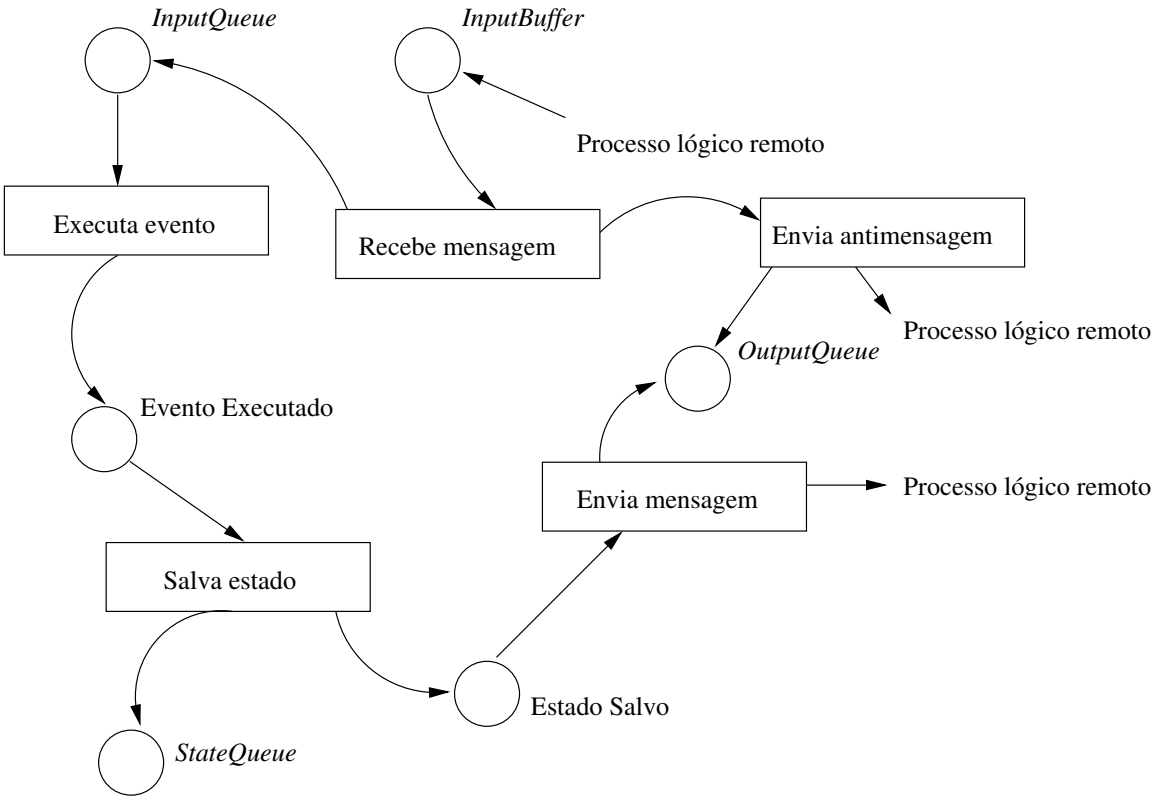


Figura 4.2: Atividades de um processo ETW.

Os processos possuem como principais estruturas de dados a *Input Queue*, a *Output Queue* e a *State Queue*, além de um relógio local (LVT) e o estado atual desse. A Figura 4.3 ilustra

o diagrama estrutural do ETW com dois processos comunicando-se trocando mensagens para a execução de eventos na simulação. Mensagens atrasadas podem chegar a qualquer momento em um processo lógico, sendo necessário efetuar *rollbacks* para recuperar e restaurar a simulação distribuída a um estado correto e para cancelar mensagens enviadas incorretamente.

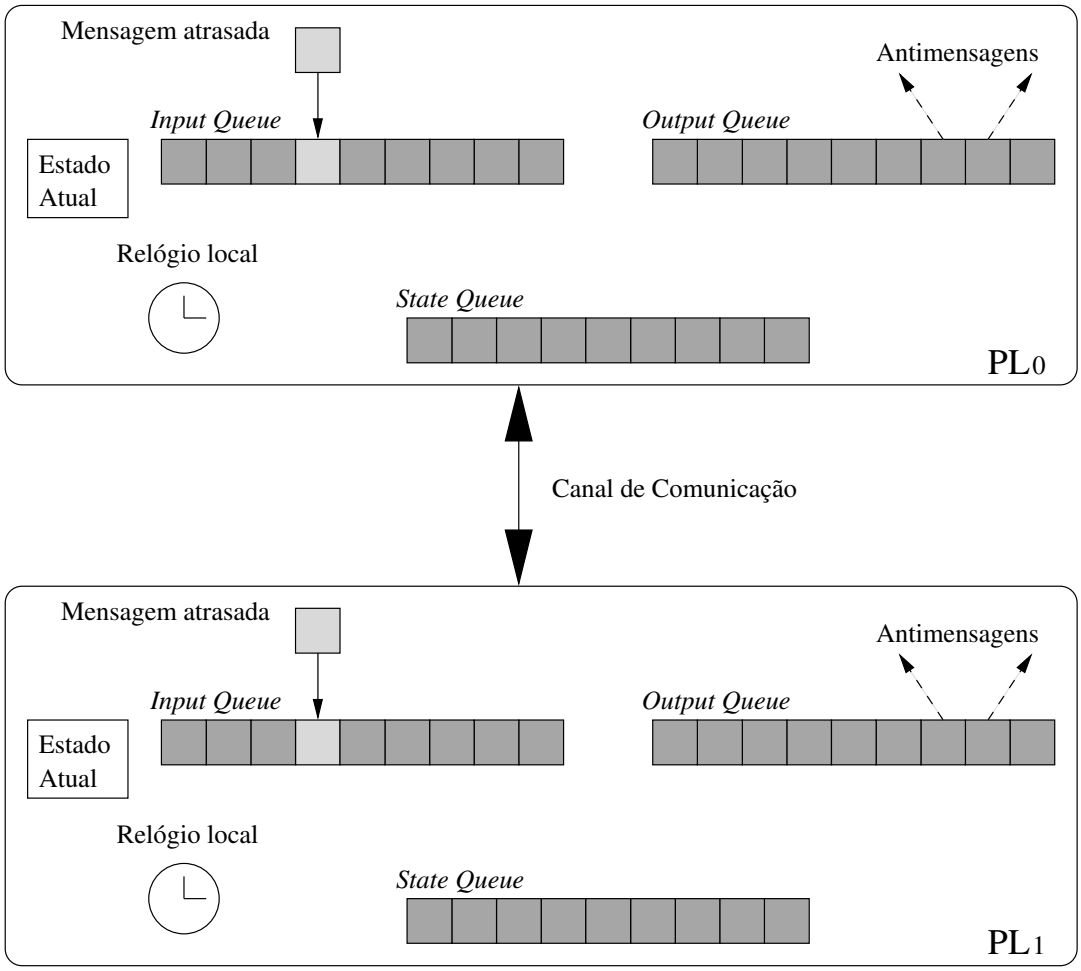


Figura 4.3: Diagrama estrutural do ETW.

A Figura 4.4 ilustra um diagrama de classes do ETW, apresentando apenas classes não parametrizadas. Por simplicidade, são apresentados apenas os principais atributos e métodos de cada classe. As classes não parametrizadas são:

- a classe tLogicalProcess, que representa a estrutura de dados de um processo lógico;
- a classe tState, que representa a estrutura de dados de um estado no ETW;
- a classe tEventMessage, que representa a estrutura de dados de uma mensagem/evento;
- a classe tResource, que representa a estrutura de dados de um recurso;

- a classe tServer, que representa a estrutura de dados de um servidor;
- e a classe tRand, que representa a estrutura de dados utilizada para a geração de números pseudo-aleatórios e para efetuar distribuições.

Essa última foi baseada nas funções desenvolvidas para geração de números aleatórios e distribuições da extensão *SMPL* [Mac87].

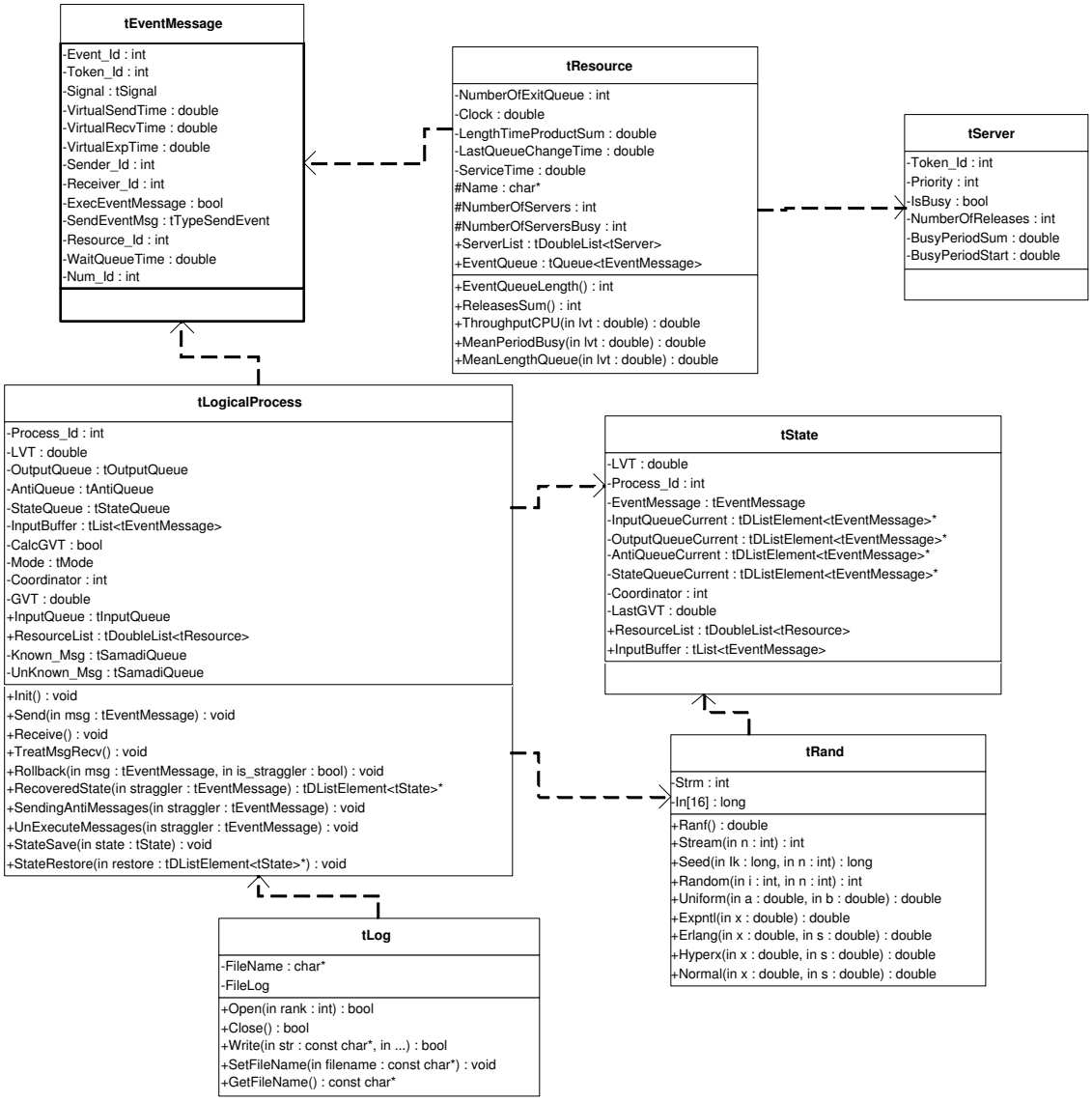


Figura 4.4: Diagrama de Classes.

A implementação do protocolo de simulação distribuída *Time Warp* é constituída dos seguintes módulos:

- Módulo de gerenciamento de interface: o gerenciamento do *Time Warp* é utilizado para dispor uma interface entre o `ETW` e o usuário, permitindo que esse escolha quais os parâmetros de entrada da simulação e quais os *plugins* irá utilizar;
- Módulo de gerenciamento de recursos: o gerenciamento de recurso possui funcionalidades para o *Time Warp* básico;
- Módulo de gerenciamento de filas e listas: o gerenciamento de filas e listas possui funcionalidades para o *Time Warp* básico;
- Módulo de gerenciamento de processos: o gerenciamento de processos lógicos é dividido em envio e execução de eventos (*Time Warp* básico), salvamento de estados (*plugin*), execução de *rollback* (*plugin*) e cálculo do GVT (*plugin*).

A Seção 4.2 apresenta o módulo de gerenciamento de interface, descrevendo suas principais funções (funções para escalonamento de eventos, requisição de recursos e liberação de servidores). A Seção 4.3 descreve o módulo de gerenciamento de recursos no `ETW`, apresentando suas estruturas de dados. A Seção 4.4 descreve o módulo de gerenciamento de filas e listas utilizadas no desenvolvimento do núcleo e apresenta o diagrama de classes das mesmas, ilustrando suas estruturas de dados. A Seção 4.5 descreve o módulo de gerenciamento de processos lógicos *Time Warp*, apresentando seus principais mecanismos, como por exemplo, o salvamento de estados e mecanismos de cancelamento utilizados pelo mesmo, bem como suas estruturas de dados. A Seção 4.6 descreve um exemplo de como utilizar o `ETW`, para apresentar os testes efetuados com um modelo de redes de filas. A Seção 4.7 descreve como o `ETW` foi validado. A Seção 4.8 apresenta as considerações finais.

4.2 Módulo de Gerenciamento de Interface

O `ETW` é composto por vários processos, cada um representando um processo lógico *Time Warp* que, por sua vez, representa um processo físico do mundo real. Cada processo possui quatro *threads*, como ilustrado na Figura 4.5. O núcleo básico, responsável pela execução de eventos e tratamento de mensagens recebidas, é a *thread* principal. A *thread* `Receive_Message` é responsável pelo recebimento de mensagens remotas. Nesse caso, essa *thread* fica em um laço infinito verificando se chega uma nova mensagem. Quando chega uma mensagem, ela é inserida em um *buffer* de mensagens recebidas e uma mensagem de reconhecimento é enviada de volta ao processo lógico emissor. A *thread* `Receive_AckMessage` é responsável pelo recebimento de mensagens de reconhecimento, utilizadas no algoritmo de cálculo do GVT implementado (Samadi). A *thread* `Calc_GVT` é responsável por efetuar o cálculo do GVT, descrito na Seção 3.3. Esse módulo descreve funções de manipulação pertencentes à *thread* principal, funções para escalonamento, requisição, liberação de eventos e criação de recursos.

O `ETW` possui algumas variáveis globais, apresentadas na Tabela 4.1. A Figura 4.6, que constitui o módulo de gerenciamento de interface, ilustra como a *thread* principal está dividida, isto é, quais são as tarefas básicas que o núcleo realiza para a execução de eventos. O

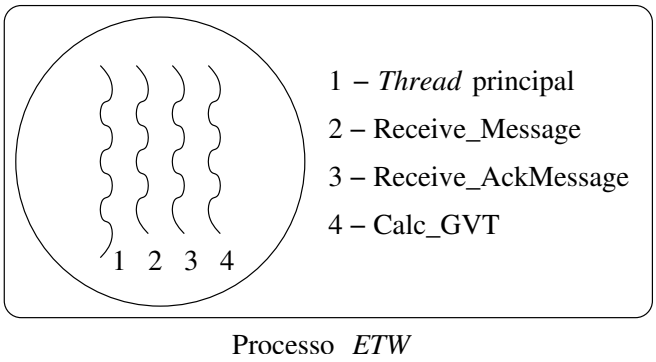


Figura 4.5: *Threads* em um processo lógico `ETW`.

item *EventMessage* representa as funções relacionadas à chegada, escalonamento, requisição, liberação e busca de eventos em um programa de simulação.

Tabela 4.1: Variáveis globais do `ETW`.

Nome	**Descrição**
simulation_name	Nome da simulação.
lp	Representa um processo lógico *Time Warp*.
rc_msg	Variável contendo o resultado da criação da *thread* Receive_Message.
rc_ack	Variável contendo o resultado da criação da *thread* Receive_KnowMessage.
rc_gvt	Variável contendo o resultado da criação da *thread* Calc_GVT.
thread_msg	*Thread* para recebimento de mensagens.
thread_ack	*Thread* para reconhecimento de mensagens.
thread_gvt	*Thread* para efetuar o cálculo do *GVT*.
NumberOfEvents	Número de eventos executados.

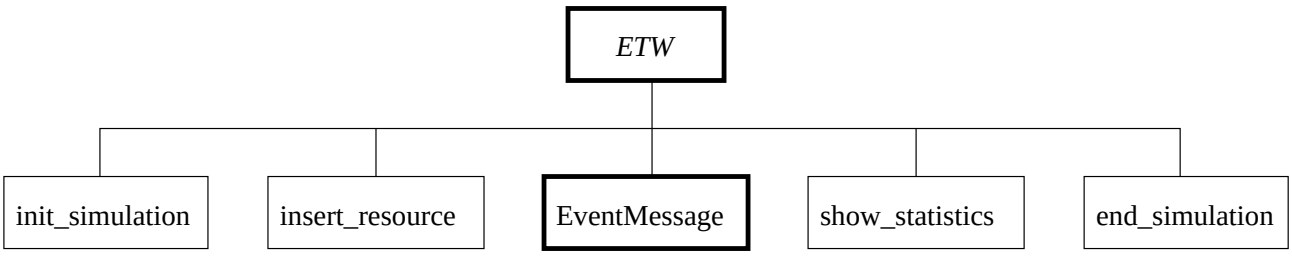


Figura 4.6: Módulo de Gerenciamento de Interface.

As principais funções utilizadas para inicializar o `ETW` e manipular recursos e servidores em um programa de simulação são apresentadas na Tabela 4.2.

Tabela 4.2: Funções utilizadas para inicialização e manipulação de recursos e servidores.

Nome	Descrição
init_simulation	Tem a finalidade de inicializar o ETW e possui como parâmetros o nome da simulação, o número de processos lógicos que serão utilizados na simulação, o tempo virtual máximo em que a simulação trabalhará e o número máximo de tarefas que serão efetuadas pela simulação.
insert_resource	O objetivo da função é criar um recurso e seus servidores. A função tem como argumentos o nome do recurso a ser criado, o número de servidores a serem inseridos nesse recurso, o tempo de serviço do mesmo e o processo lógico no qual esse será inserido.
end_simulation	Finaliza a execução do ETW e também o uso do *LAM-MPI*.

A *EventMessage* destacada da Figura 4.6 é apresentada na Figura 4.7. Por exemplo, o *Time Warp*, que executa tarefas correspondentes ás operações na simulacao sequencial, como a execução de mensagens/eventos. A Tabela 4.3 apresenta essas operaçóes.

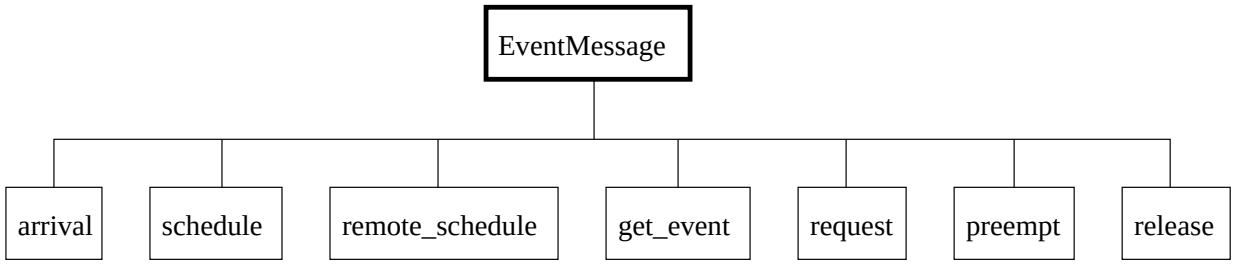


Figura 4.7: Funções Relacionadas à Mensagens/Eventos.

As principais funções do ETW para manipulação de *threads* e outras operações são apresentadas na Tabela 4.4. A Figura 4.8 ilustra essas funções do módulo de gerenciamento de interface.

4.3 Módulo de Gerenciamento de Recursos

Este módulo representa um recurso em um modelo de redes de filas. Um recurso, como pode ser ilustrado pela Figura 4.9, possui uma fila com entidades (clientes) esperando para serem atendidos em um ou mais servidores [Soa92, Tan94]. Como exemplo de recursos em um sistema de redes de filas, pode-se citar uma UCP em um modelo de sistema computacional, uma bomba de gasolina em um modelo de sistema de posto de combustível ou ainda um caixa em um modelo de sistema bancário, entre outros. Além disso, um recurso em um sistema pode ser reservado e liberado a qualquer momento da simulação.

Tabela 4.3: Funções utilizadas para execução da simulação.

Nome	Descrição
arrival	Representa a chegada de um cliente em um recurso do modelo de redes de filas.
schedule	Cria um novo evento que será escalonado para execução local.
remote_schedule	Cria um novo evento para ser executado remotamente.
get_event	Devolve o identificador do próximo evento a ser executado pelo processo lógico e o *token* (argumento passado por referência) correspondente ao mesmo, ou faz um tratamento das mensagens que estão na *InputBuffer* do processo lógico.
request	Faz uma requisição de um evento por um *token* para ser executado em um recurso do processo.
preempt	Efetua um bloqueio em um servidor de um recurso.
release	Libera um servidor do recurso passado como parâmetro.

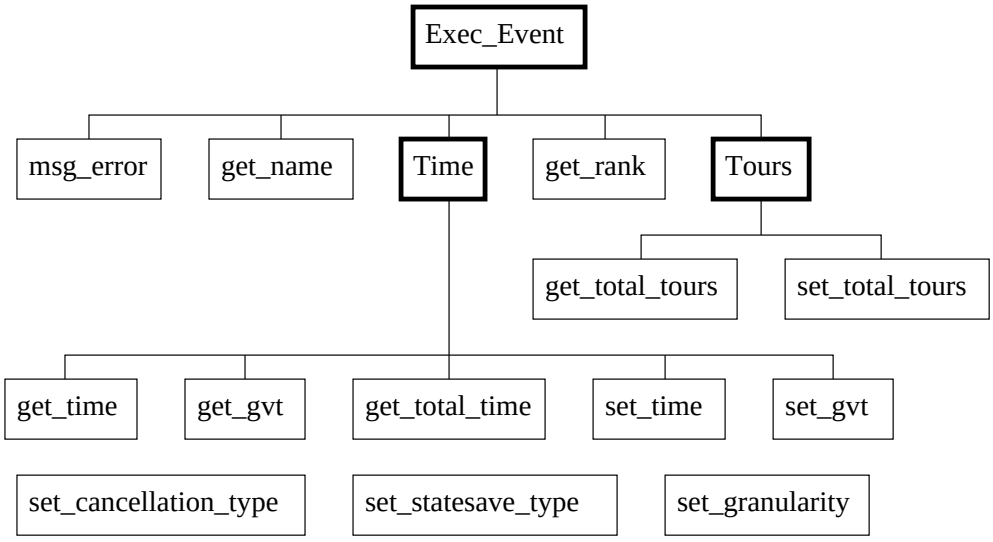


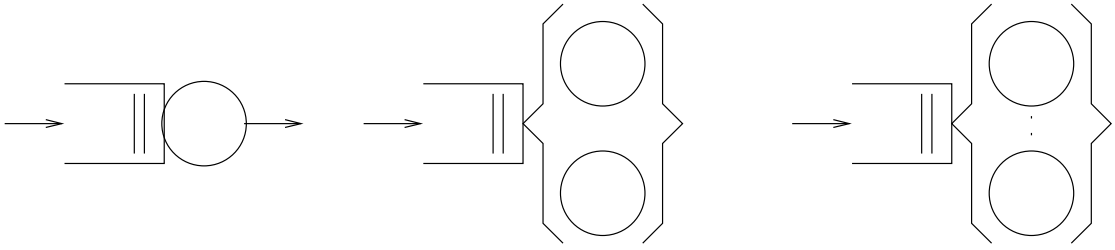
Figura 4.8: Demais funções do ETW.

Um processo ETW possui uma lista de recursos e cada um possui uma lista de servidores e uma fila de eventos. Na execução de um evento para requisição de um servidor de um recurso, se existir pelo menos um servidor livre, esse servidor é ocupado pelo cliente, mas se todos esses servidores estão ocupados, o cliente é inserido na fila do recurso. Quando um servidor é liberado, o primeiro evento, na fila do recurso, é retirado e imediatamente escalonado para ser o próximo da lista de eventos futuros para execução.

A classe tServer é utilizada para representar a lista de servidores de um recurso, isto é, um objeto dessa classe representa um servidor. A classe tResource é utilizada quando uma mensagem/evento requisitar um servidor do recurso e esse contiver servidores livres, a

Tabela 4.4: Funções utilizadas para execução da simulação.

Nome	Descrição
thread_recvmsg	Responsável pelo recebimento de mensagens via processos lógicos remotos.
thread_knowmsg	Responsável pelo recebimento de reconhecimento mensagens via processos lógicos remotos.
Calc_GVT	Responsável por efetuar o cálculo do GVT, aplicando o algoritmo de Samadi, descrito na Seção 3.4.
msg_error	Imprime mensagens de erros ocorridos durante a execução do programa de simulação distribuída.



(a) Recurso: 1 Fila e 1 servidor. (b) Recurso: 1 Fila e 2 servidores. (c) Recurso: 1 Fila e infinitos servidores.

Figura 4.9: Exemplos de recursos em redes de filas.

mensagem/evento irá ocupar um desses servidores livres, caso contrário, a mensagem/evento irá esperar na fila do recurso. Um evento pode ficar no servidor até seu término de atendimento ou até que alguém o retire do servidor, isto é, até que seja bloqueado.

4.3.1 Atributos de tServer

A classe possui quatro atributos protegidos e quatro atributos privados, como pode ser visto na Tabela 4.5. Os atributos protegidos estão relacionados com a alocação do servidor, enquanto os atributos privados estão relacionados com as suas estatísticas.

4.3.2 Métodos de tServer

A classe tServer possui três construtores:

- o construtor *default*, que inicia todos os atributos da classe com zero e *IsBusy* com o valor **false**;
- um construtor, que passa como parâmetros a identificação do servidor, o número do *token* e a prioridade do evento no mesmo. Esse construtor inicia os atributos *Server_Id*, *Token_Id* e *Priority* com os valores passados como parâmetro e os atributos restantes com zero;

Tabela 4.5: Atributos de um recurso.

Nome	Descrição
Server_Id	Identifica um servidor.
Token_Id	Identifica o *token* do evento que está fazendo a alocação do recurso.
Priority	Indica a prioridade do evento (utilizado em caso de bloqueio).
IsBusy	Indica se o servidor está ocupado ou não.
NumberOfReleases	Armazena o número de vezes que um servidor é liberado.
BusyPeriodSum	Soma dos períodos de ocupação do servidor.
BusyPeriodStart	Início do tempo de ocupação do servidor.
BusyPeriodEnd	Tempo final de ocupação de um servidor.

- o construtor de cópia, que faz com que os atributos do objeto *this* recebam os valores dos atributos do objeto passado como parâmetro.

A Tabela 4.6 apresenta os principais métodos implementados para a classe tServer.

Tabela 4.6: Métodos de tServer.

Nome	Descrição
GetNumberOfReleases	Devolve o número de liberações efetuadas pelo servidor até o instante.
GetBusyPeriodSum	Esse método retorna a soma dos períodos ocupados do servidor.
GetBusyPeriodStart	Esse método retorna o tempo inicial de ocupação do servidor. Esse tempo é ajustado assim que o servidor é alocado para um cliente.
operator ==	O operador de igualdade utiliza os atributos *Token_Id*, *Priority*, *NumberOfReleases* e *IsBusy* do objeto *this* para fazer uma comparação com o objeto passado como parâmetro.
operator =	Faz uma atribuição, colocando cada atributo do objeto passado como parâmetro no objeto *this*.

4.3.3 Atributos de tResource

Para representar um recurso no ETW, foi necessário o desenvolvimento da classe tResource. Essa classe implementa as características de um recurso utilizado na solução de um modelo através da simulação.

Os atributos da classe podem ser vistos na Tabela 4.7. A classe tResource possui oito atributos privados, sendo que os quatro primeiros são utilizados para controle e definição do recurso, e os outros representando dados estatísticos. Além disso, dois atributos públicos

que representam a lista de servidores do recurso e a fila do mesmo. A Figura 4.10 ilustra a estrutura de dados desenvolvida para definir um recurso. As estruturas de dados da lista de servidores e da fila do recurso são apresentadas na Seção 4.4.

Tabela 4.7: Atributos de um recurso.

Nome	**Descrição**
Id	Identificação do recurso.
Name	Nome do recurso
NumberOfServers	Número de servidores.
NumberOfServersBusy	Número de servidores ocupados.
NumberOfExitQueue	Número de eventos que saíram da fila.
LengthTimeProductSum	Soma dos períodos ocupados dos servidores do recurso.
LastQueueChangeTime	Tempo da última alteração na fila do recurso.
ServiceTime	Tempo de serviço do recurso.
ServerList	Lista de servidores.
EventQueue	Fila do recurso.

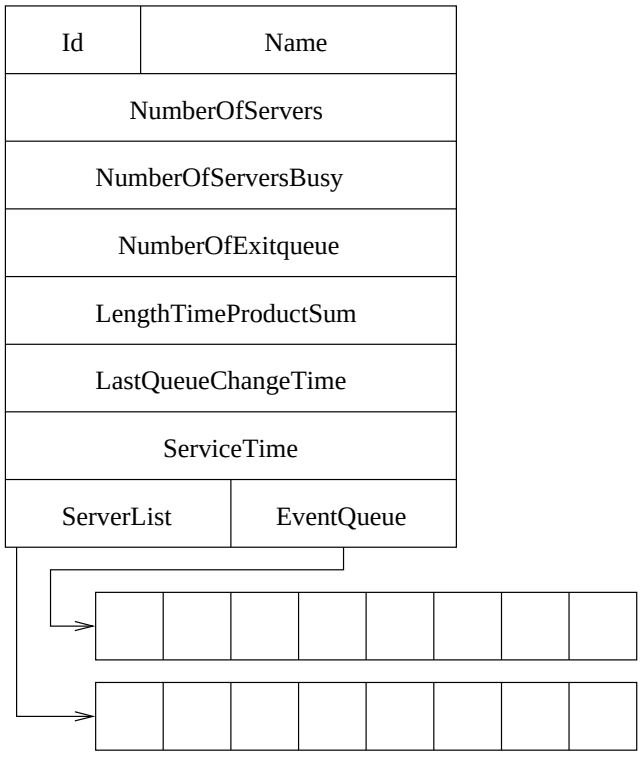


Figura 4.10: Estrutura de dados de um recurso no `ETW`.

4.3.4 Métodos de tResource

Esta classe possui três construtores e um destrutor. O primeiro construtor é o *default*, o qual inicializa todos os atributos da classe com o valor zero. O segundo construtor passa como parâmetro um ponteiro para caracter e um inteiro. O primeiro parâmetro representa o nome do recurso e o segundo parâmetro representa o número de servidores que estarão contidos no recurso. Os atributos restantes são inicializados com o valor zero. Finalmente, o terceiro construtor é o de cópia. O objeto passado como parâmetro é atribuído ao objeto *this*. O destrutor é utilizado para desalocar a memória alocada pelo nome do recurso. Os principais métodos de tResource são apresentados na Tabela 4.8.

Tabela 4.8: Métodos de tResource.

Nome	Descrição
EventQueueLength	Devolve o conteúdo do atributo *NumberOfExitQueue*.
ReleasesSum	Percorre a lista de servidores do recurso somando o número de liberações efetuadas nesses servidores.
ThroughputCPU	Percorre todos os servidores do recurso somando seus períodos ocupados e dividindo esse valor pelo *LVT* da simulação.
MeanPeriodBusy	É utilizado para calcular a média dos períodos ocupados dos servidores de um recurso.
MeanLengthQueue	É utilizado para calcular o tamanho médio da fila do recurso.
NumberOfPreempt	Devolve o número de eventos que sofreram bloqueio em algum servidor.
operator ==	Implementa o operador de igualdade da classe tResource.
operator =	Implementa o operador de atribuição da classe tResource.
operator !=	Verifica se o recurso *this* é diferente do recurso passado como parâmetro.

4.4 Módulo de Gerenciamento de Filas e Listas

O ETW necessita de um módulo gerenciador para manipular suas filas e listas. Esse módulo permite gerenciar, entre outras, a lista de eventos futuros (*InputQueue*), a filas de recursos (*ResourceList*), a lista de antimensagens (*OutputQueue*) e a lista de estados da simulação (*StateQueue*). O gerenciamento dessas filas e listas permite, basicamente, a inserção, remoção e busca de elementos. Por exemplo, um recurso possui, além da fila de eventos aguardando a execução (*EventQueue*), uma lista de servidores (*ServerList*), permitindo que um recurso possua um ou mais servidores atendendo aos clientes na simulação.

Para representar listas e filas foram desenvolvidas classes parametrizadas. A primeira delas, a classe tList<>, implementa listas simplesmente encadeadas, enquanto a classe tDoubleList<>

implementa listas duplamente encadeadas. Ainda, a classe tQueue<> implementa a estrutura de uma fila. O diagrama de classes para as classes parametrizadas é ilustrado na Figura 4.11. Além dessas classes parametrizadas, outras quatro classes, todas herdadas da classe tDoubleList<>, são descritas: a tInputQueue, a tOutputQueue, a tAntiQueue, a tStateQueue e tSamadiQueue.

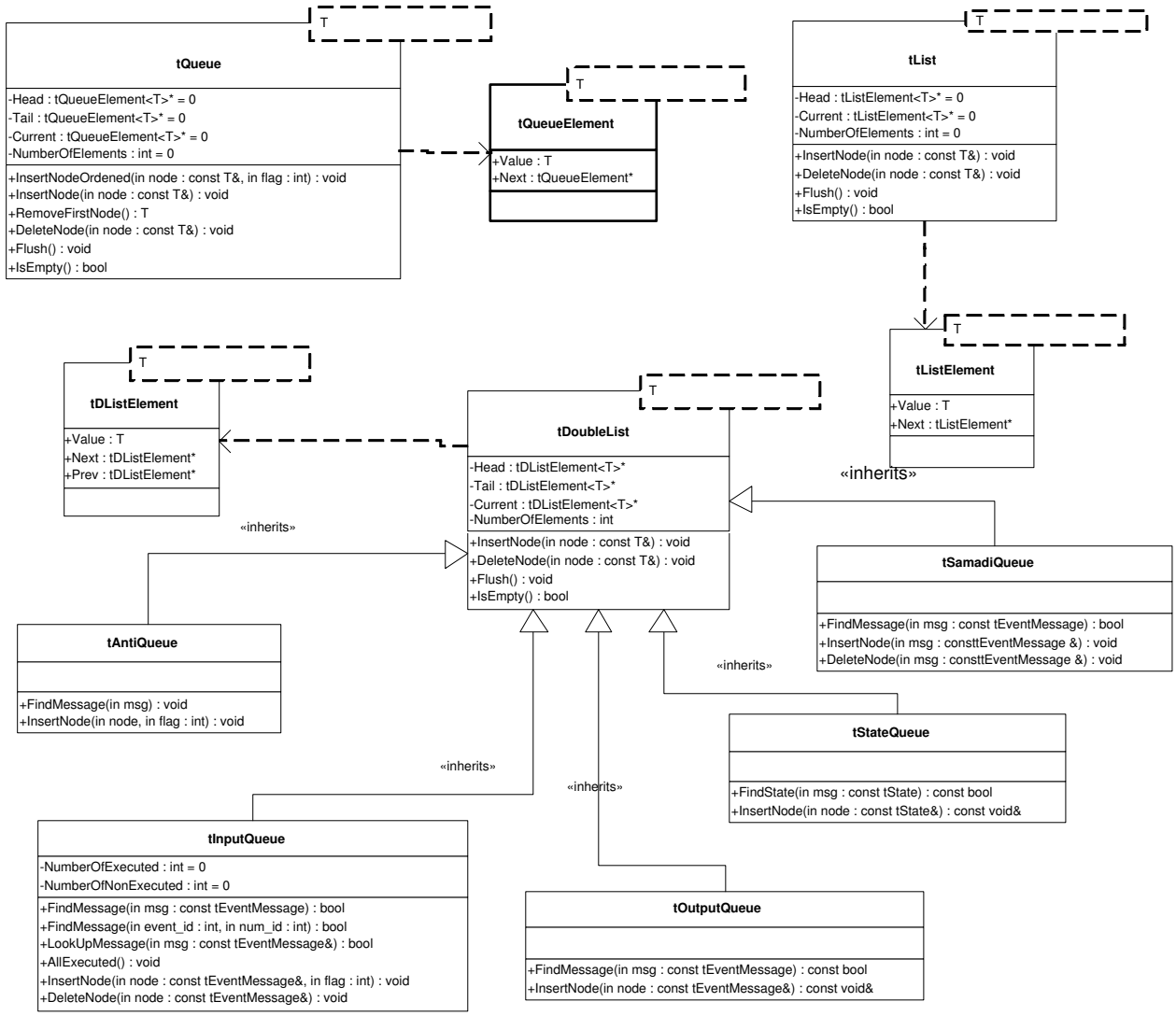


Figura 4.11: Diagrama de Classes *Templates*.

4.4.1 Classe *Template* tList<>

A classe tList<> é uma lista linear simplesmente encadeada e cada elemento dessa lista é um objeto da classe tListElement<>. Ambas as classes estão localizadas no arquivo list.h. O atributo *InputBuffer* da classe tLogicalProcess é do tipo tList<T>.

A classe tListElement<> possui apenas membros públicos. Um construtor *default*, um construtor que inicia o atributo *Value* com um valor passado como parâmetro, um método, chamado GetValue que retorna o atributo *Value*, o próprio atributo *Value*, o qual representa o conteúdo (valor) de um elemento *template* e um ponteiro para um tListElement<>, o qual representa o ponteiro para o próximo elemento da lista. A Figura 4.12 ilustra essa lista e seus atributos.

Atributos

Essa classe possui três atributos e ambos são protegidos, como visto na Tabela 4.9. O primeiro atributo é um ponteiro do tipo tListElement<> para o primeiro elemento da lista, isto é, a cabeça da lista. O atributo *Current* é um ponteiro do tipo tListElement<> no qual representa o elemento corrente da lista simplesmente encadeada e o atributo *NumberOfElements* representa o número de elementos na lista.

Tabela 4.9: Atributos de uma tList<>.

Nome	Descrição
Head	Apontador para o início da lista
Current	Apontador para o elemento corrente da lista
NumberOfElements	Número de elementos na lista

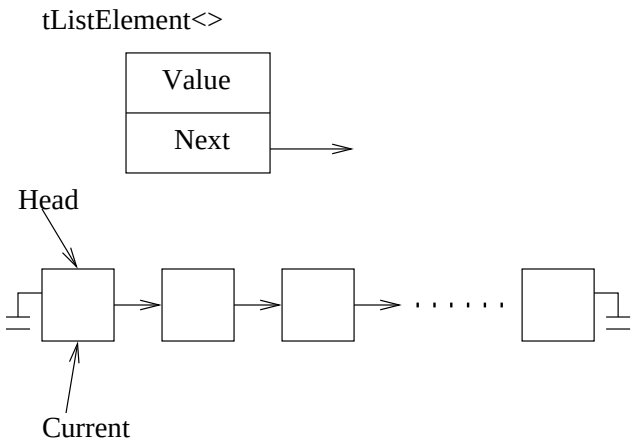


Figura 4.12: Lista linear simplesmente encadeada no ETW.

Métodos

Todos os métodos da classe tList<> são públicos. Essa classe possui um construtor *default* que inicia *Head* e *Current* apontando para zero (*NULL*) e NumberOfElements com o valor

zero. Além disso, a classe possui também um construtor de cópia que atribui ao objeto *this* o objeto passado como parâmetro. O destrutor aciona o método Flush, o qual tem a finalidade de percorrer toda a lista e desalocar a memória utilizada por todos os elementos que ainda estão contidos nela no momento em que o destrutor é chamado. Os principais métodos da classe são apresentados na Tabela 4.10.

Tabela 4.10: Métodos de tList.

Nome	Descrição
InsertNode	Insere um elemento na lista simplesmente encadeada.
DeleteNode	Passa como parâmetro o elemento a ser removido. Se o elemento for encontrado, é removido.
IsEmpty	Verifica se a lista encadeada está vazia.
Flush	Percorre uma fila liberando os espaços na memória ocupados por ela.
operator =	A lista encadeada passada como parâmetro é percorrida de modo a inserir cada um de seus elementos em outra lista encadeada.

4.4.2 Classe *Template* tDoubleList<>

Esta classe representa uma lista linear duplamente encadeada na qual cada elemento é um objeto do tipo tDListElement<>. A classe tDListElement<> possui um construtor *default* e um construtor de cópia, que passa um parâmetro que é uma referência do tipo T (classe parametrizada). Esse construtor inicializa o atributo *Value* com o valor passado como parâmetro. A classe possui também um método chamado GetValue, que devolve o valor do atributo *Value*. O atributo *Value* representa o conteúdo (valor) do objeto, o atributo *Next* é um ponteiro para o próximo elemento na lista duplamente encadeada e o atributo *Prev* é um ponteiro para o elemento anterior na lista duplamente encadeada. A Figura 4.13 ilustra uma representação dessa lista e seus atributos.

Atributos

Essa classe tem apenas atributos protegidos, como apresentado na Tabela 4.11. O atributo *Head* é um ponteiro do tipo tDListElement<> para o início da lista duplamente encadeada, *Current* é um ponteiro para o elemento corrente da lista duplamente encadeada, *Tail* é um ponteiro para o final da lista duplamente encadeada e *NumberOfElements* é o número de elementos na lista.

Métodos

A classe tDoubleList<> possui um construtor *default*, que inicializa *Head*, *Tail* e *Current* com o valor *NULL* e *NumberOfElements* com o valor zero. A classe também possui um cons-

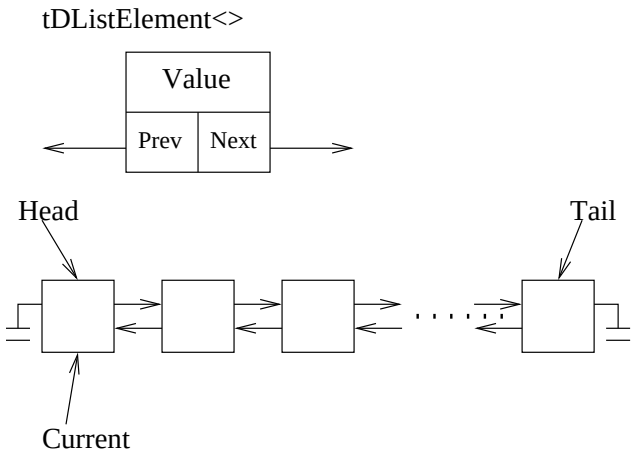


Figura 4.13: Lista linear duplamente encadeada no ETW.

Tabela 4.11: Atributos de uma tDoubleList<>.

Nome	Descrição
Head	Apontador para o início da lista
Current	Elemento corrente da lista
Tail	Apontador para o final da lista
NumberOfElements	Número de elementos na lista

trutor de cópia que atribui ao novo objeto, *this*, os valores do objeto passado como parâmetro. O destrutor da classe aciona o método Flush para liberar toda a memória utilizada pela lista. O método percorre toda a lista removendo os elementos existentes até que a lista se torne vazia. Os principais métodos da classe são apresentados na Tabela 4.12.

Tabela 4.12: Métodos de tDoubleList.

Nome	Descrição
InsertNode	Insere um novo elemento na lista duplamente encadeada.
DeleteNode	Remove da lista o elemento passado como parâmetro, se esse estiver na lista.
IsEmpty	Retorna um valor indicando se a lista esta ou não vazia.
Flush	Percorre uma fila liberando os espaços na memória ocupados por ela.
operator =	A lista passada como parâmetro é percorrida com o objetivo de copiar cada um de seus elementos para uma nova lista.

4.4.3 Classe *Template* tQueue<>

A classe é utilizada para representar as estruturas de dados em forma de filas no ETW, como ilustrado na Figura 4.14. No caso do *Time Warp*, a única estrutura de dados em forma de fila é a fila de eventos em um recurso, chamada *EventQueue* e descrita na Seção 4.3.

A seção protegida da classe possui três atributos do tipo tQueueElement<>. Essa estrutura representa um elemento pertencente à fila. A estrutura de dados tQueueElement<> possui dois atributos: *Value*, que é um tipo parametrizado, e um atributo chamado *Next*, o qual é um ponteiro para um tQueueElement<>. O atributo *Value* representa o valor do elemento correspondente na fila. Por exemplo, para uma fila de inteiros, o atributo *Value* representa o valor inteiro daquele elemento na fila. O atributo *Next* aponta para o próximo elemento da fila. Além disso, a estrutura possui um construtor *default* e um construtor que atribui o valor passado como parâmetro ao atributo Value.

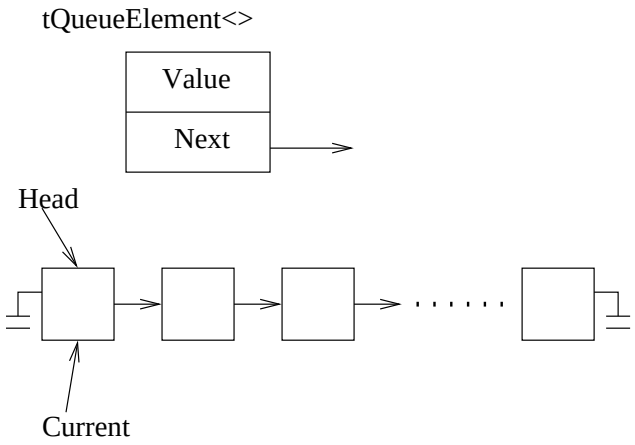


Figura 4.14: Fila no ETW.

Atributos

A classe tQueue<> é amiga da classe tState, isto é, a classe tState pode acessar todos os dados da *template* tQueue<>, tanto dados públicos quanto protegidos e privados. A Tabela 4.13 apresenta os quatro atributos desta classe.

Tabela 4.13: Atributos da classe *template* tQueue<>.

Nome	Descrição
Tail	Apontador para o final da fila.
Head	Apontador para o início da fila.
Current	Apontador para o elemento corrente da fila.
NumberOfElements	Quantidade de elementos na fila.

Métodos

Todos os métodos pertencentes à classe *template* tQueue<T> são públicos. A classe possui um construtor *default*, o qual inicia *Tail*, *Head* e *Current* apontando para *NULL*, e também inicializa *NumberOfElements* com o valor zero (a fila ainda não possui elementos) e um construtor de cópia, o qual faz uma cópia do objeto passado como parâmetro para o objeto *this*. O destrutor da classe apenas invoca um método chamado *Flush*. Os principais métodos de tQueue são apresentados na Tabela 4.14.

Tabela 4.14: Métodos de tQueue.

Nome	Descrição
InsertNodeOrdened	Insere na fila um novo elemento ordenado por *Value*.
InsertNode	Insere um elemento no final da fila, se esta existir, caso contrário, será o primeiro elemento da fila.
RemoveFirstNode	Esse método remove o primeiro elemento da fila.
DeleteNode	Percorre a fila procurando pelo elemento passado como parâmetro na fila. Se esse é encontrado, é removido.
Flush	Percorre uma fila liberando os espaços na memória ocupados por ela.
IsEmpty	Verifica se a fila está vazia.
operator =	A fila passada como parâmetro é percorrida de modo a inserir cada elemento seu no novo objeto.

4.4.4 Listas Derivadas de tDoubleList<>

Classe tAntiQueue

Esta classe é derivada da classe tDoubleList e representa uma lista ligada duplamente encadeada de antimensagens que chegam em um processo lógico antes da mensagem correspondente à mesma. Pelo fato de ser uma classe derivada de outra, um objeto desta classe pode acessar todos os membros públicos da classe base (a classe tDoubleList).

A classe tAntiQueue não possui atributos, pois um objeto dessa classe utiliza apenas os atributos disponibilizados pela classe base. Também não possui construtores e nem destrutor. Da mesma forma, utiliza os construtores e o destrutor da classe base.

Possui apenas três métodos, *FindMessage*, *InsertNode* e *GetCurrentValue*, descritos na Tabela 4.15.

Classe tInputQueue

A classe representa uma lista ligada duplamente encadeada de eventos futuros. Pode-se comparar essa lista com a lista de eventos futuros utilizada na simulação seqüencial, exceto que essa lista, além de possuir os eventos ainda não executados, guarda também os eventos

Tabela 4.15: Métodos de tAntiQueue.

Nome	Descrição
FindMessage	Procura o evento/mensagem passado como parâmetro na *AntiQueue*.
InsertNode	Insere um elemento na lista de forma que a mesma permaneça ordenada.
GetCurrentValue	Retorna o valor do elemento corrente da lista.

já executados em um processo lógico da simulação distribuída. Isso acontece para que erros futuros de causa e efeito possam ser corrigidos através do cancelamento de mensagens e restauração de estados.

A classe possui, além dos atributos adquiridos da classe base, dois atributos privados, que representam dados estatísticos. O primeiro deles, *NumberOfExecuted* representa o número de eventos executados por um processo lógico e o segundo, *NumberOfNotExecuted* representa o número de eventos ainda não executados por um processo lógico, ao final da simulação distribuída.

A classe possui um construtor *default*, o qual inicializa os dois atributos da classe com o valor zero, pois no início da simulação distribuída não existem eventos na *InputQueue*. Além de inicializar esses dois atributos, o construtor da classe aciona o construtor *default* da classe base para inicializar seus atributos. A Tabela 4.16 apresenta os principais métodos da classe.

Tabela 4.16: Métodos de tInputQueue.

Nome	Descrição
FindMessage	Procura o evento passado como parâmetro na *InputQueue*.
FindMessage	Procura o elemento com *Event_Id* e *Num_Id* iguais aos valores passados como parâmetro, percorrendo a *InputQueue*.
LookUpMessage	Procura o elemento com *Event_Id*, *Signal*, *VirtualSendTime*, *VirtualRecvTime*, *VirtualExpTime*, *Sender_Id*, *Receiver_Id*, *ExecEventMessage* e *SendEventMsg* iguais aos valores da mensagem passada como parâmetro.
AllExecuted	Varre a *InputQueue* procurando por algum evento não executado.
InsertNode	Insere um elemento na lista de forma que esta permaneça ordenada.
DeleteNode	A *InputQueue* é percorrida buscando o elemento passado como parâmetro.
operator ==	Percorre a lista *this* verificando se a mesma é igual à passada como parâmetro, elemento por elemento.

Classe tOutputQueue

É uma classe derivada de tDoubleList. Representa uma lista duplamente encadeada de antimensagens que são enviadas por um processo lógico. Quando um processo lógico envia uma mensagem remota, uma cópia negativa dessa mensagem é inserida em uma fila de saída de mensagens, *OutputQueue*. Pelo fato de ser uma classe derivada de outra, um objeto dessa classe pode acessar todos os membros públicos da classe base, no caso a classe tDoubleList.

A classe tOutputQueue, da mesma forma que um objeto da classe tAntiQueue, utiliza apenas os atributos disponibilizados pela classe base. Também não possui construtores e nem destrutor, utilizando apenas construtores e o destrutor da classe base.

Possui apenas três métodos, *FindMessage*, *InsertNode* e *GetCurrentValue*, apresentados na Tabela 4.17.

Tabela 4.17: Métodos de tOutputQueue.

Nome	Descrição
FindMessage	Procura a mensagem passada como parâmetro na *OutputQueue*.
InsertNode	Insere um elemento na lista de forma que a mesma permaneça ordenada.

Classe tStateQueue

É uma classe derivada de tDoubleList e representa uma lista duplamente encadeada de estados salvos na execução de eventos de um processo lógico. A lista de estados salvos é utilizada para efetuar futuros *rollbacks*, onde os estados salvos erroneamente são desfeitos e um estado correto é recuperado após o cancelamento de mensagens. Pelo fato de ser uma classe derivada de outra, um objeto dessa classe pode acessar todos os membros públicos da classe base, no caso a classe tDoubleList.

Essa classe, utiliza apenas os atributos disponibilizados pela classe base. Também não possui construtores e nem destrutor, utilizando apenas construtores e o destrutor da classe base.

Possui apenas três métodos, *FindState*, *InsertNode* e *GetCurrentValue*, descritos na Tabela 4.18.

Tabela 4.18: Métodos de tStateQueue.

Nome	Descrição
FindState	Procura o estado passado como parâmetro na *StateQueue*.
InsertNode	Insere um elemento na lista de forma que a mesma permaneça ordenada.
GetCurrentValue	Devolve o valor do elemento corrente da lista.

Classe tSamadiQueue

Assim como as classes anteriores, a classe tSamadiQueue também é derivada de tDoubleList. Um objeto dessa classe é uma lista duplamente encadeada de objetos do tipo tEventMessage. Essa classe possui apenas atributos e construtores derivados de sua classe base. Possui três métodos, *FindMessage*, *InsertNode* e *DeleteNode*, descritos a seguir. Essa classe foi construída para representar as listas de mensagens reconhecidas e de mensagens não reconhecidas, utilizadas para o cálculo do GVT de Samadi. A Tabela 4.19 apresenta os principais métodos da classe.

Tabela 4.19: Métodos de tSamadiQueue.

Nome	Descrição
FindMessage	Procura a mensagem/evento passada como parâmetro na lista.
InsertNode	Insere um elemento na lista de forma que a mesma permaneça ordenada.
DeleteNode	Remove o elemento passado como parâmetro da lista.

4.5 Módulo de Gerenciamento de Processos

Este módulo representa o gerenciamento de um processo `ETW`. Um processo é responsável pelo envio, recebimento e tratamento de mensagens e antimensagens e também pelo mecanismo de *rollback*, entre outros. Esta seção descreve as características desenvolvidas no núcleo `ETW` para um processo lógico.

O envio/recebimento de mensagens/antimensagens, realizado por um processo lógico, dá-se através do uso da biblioteca de troca de mensagens *LAM-MPI* [Lam04, Sni96]. Um processo utiliza basicamente as funções de envio/recebimento não bloqueante para realizar essas tarefas (`MPI_Isend` e `MPI_Irecv`). Quando um evento A gera um novo evento B para ser escalonado remotamente, o mesmo deve ser enviado a um processo lógico remoto. Tanto A quanto B são objetos do tipo tEventMessage. Para que B seja enviado, o mesmo é empacotado em um objeto do tipo tMessage. Isso acontece porque tEventMessage é uma classe, contendo atributos e métodos, enquanto tMessage é uma estrutura contendo apenas atributos, os quais são os mesmos definidos em tEventMessage. Portanto, o objetivo em utilizar um objeto tMessage ao invés de um tEventMessage é economia no envio de *bytes*.

A execução de eventos em um processo é efetuada retirando um tEventMessage da *InputQueue* com a menor marca de tempo, ou seja, é retirado dessa lista um evento ainda não executado, já que essa *guarda* tanto eventos já executados quanto os ainda não executados. As classes tEventMessage e tLogicalProcess estão relacionadas com a execução de eventos no `ETW` e a estrutura tMessage é utilizada no envio de mensagens para processos lógicos remotos.

4.5.1 Estrutura tMessage

Antes da descrição da estrutura de dados de um processo lógico, é necessário definir as estruturas de dados para tMessage e para a classe tEventMessage.

Um atributo do tipo tTypeSendEvent pode possuir dois valores, *Local* ou *Remote*. O valor *Local* indica que uma mensagem será executada localmente. Já uma mensagem que possui um valor *Remote* será executada em um processo lógico remoto.

Um atributo do tipo tSignal representa o sinal de uma mensagem no protocolo *Time Warp* e possui dois valores:

- *Positive*, que representa uma mensagem no protocolo;
- *Negative*, o qual indica uma antimensagem no protocolo.

Um objeto do tipo tMessage é necessário apenas para transmissão de dados. Seriam utilizados muitos *bytes* para empacotar um objeto da classe tEventMessage com todos seus atributos e métodos, portanto optou-se por criar a estrutura tMessage. Assim, apenas os *bytes* ocupados pelos atributos são utilizados, enviando uma quantidade menor de *bytes*. A Tabela 4.20 apresenta todos os atributos da estrutura tMessage.

Tabela 4.20: Atributos da estrutura tMessage.

Nome	**Descrição**
Event_Id	Identificador do evento.
Token_Id	Token do evento.
Signal	Sinal da mensagem.
VirtualSendTime	Marca de tempo de envio da mensagem.
VirtualRecvTime	Marca de tempo de recebimento da mensagem. É o tempo de execução da mensagem (evento).
VirtualExpTime	Marca de tempo somada com o LVT para cálculo do VirtualRecvTime da Mensagem.
Sender_Id	Processo emissor de uma mensagem.
Receiver_Id	Processo receptor de uma mensagem.
ExecEventMessage	Indica se um evento da InputQueue já foi executado.
SendEventMsg	Tipo de envio do evento criado (local ou remoto).
Resource_Id	Recurso que o evento utiliza.
Server_Id	Identificador do servidor sendo ocupado.
WaitQueueTime	Tempo de espera na fila de um recurso.
Num_Id	Número de identificação de um evento.
Parent	Representa a mensagem/evento que criou esta.

4.5.2 Classe tEventMessage

A classe tEventMessage implementa as características de uma mensagem/antimensagem (mensagem/evento) em um processo ETW. Um objeto da classe tEventMessage pode representar um evento inserido na lista de eventos futuros para execução na simulação (*InputQueue*), uma antimensagem inserida na *OutputQueue* após o envio de sua mensagem positiva, ou ainda pode representar uma antimensagem que chegou em um processo lógico antes da mensagem correspondente.

Atributos

A seção privada dessa classe possui quinze atributos, todos semelhantes aos atributos da estrutura tMessage, como mostra a Tabela 4.20.

Métodos

Essa classe possui três construtores. O primeiro é o *default*, que inicializa todos os atributos da classe. Um segundo construtor inicializa todos os atributos da classe com os valores passados como parâmetro. Já um construtor de cópia passa como parâmetro um tEventMessage& e atribui o objeto passado como parâmetro ao objeto *this*. Todos os métodos da classe tEventMessage são públicos e os principais métodos são apresentados na Tabela 4.21.

4.5.3 Classe tRand

Esta classe encapsula as funções para geração de números pseudo-aleatórios obtidas a partir da biblioteca do *SMPL* [Mac87].

Atributos

A classe tRand possui dois atributos, um público e um privado. O atributo público *In* é um vetor de sementes, com dezesseis elementos, utilizado na geração de números pseudo-aleatórios enquanto que *Strm* é um atributo privado utilizado para ajustar a semente a ser utilizada na geração de um número aleatório.

Métodos

Esta classe possui um único construtor, utilizado para inicializar seus atributos. *Strm* é inicializado com o valor um (1) e o vetor *In* é inicializado conforme descrito em [Mac87]. Os principais métodos da classe são apresentados na Tabela 4.22.

4.5.4 Classe tState

Um objeto da classe tState representa um estado no ETW. Os estados do processo lógico devem ser salvos, ou a cada execução de evento, ou usando algum mecanismo adaptativo ou de maneira incremental, pois se uma mensagem atrasada chega em um processo lógico, um

Tabela 4.21: Métodos de tEventMessage.

Nome	Descrição
operator ==	Verifica se os atributos do evento correspondente são iguais aos atributos do evento passado como parâmetro.
operator !=	Verifica se todos os atributos do evento correspondente são iguais aos do evento passado como parâmetro, exceto o sinal.
operator >	Se o sinal do evento correspondente for igual ao sinal do evento passado como parâmetro e esses sinais forem positivos, o método retorna verdadeiro se o *VirtualRecvTime* de *this* for maior que o do evento passado como parâmetro. Se o sinal do evento correspondente for igual ao do evento passado como parâmetro e os sinais desses eventos forem negativos, o método retorna verdadeiro se o *VirtualSendTime* de *this* for maior que o do evento passado como parâmetro.
operator >=	Efetua a operação comparando o *VirtualRecvTime* e o *VirtualSendTime*, da mesma forma que o método acima.
operator =	Faz uma atribuição de uma variável do tipo tMessage, passada como parâmetro, para o *this* da classe tEventMessage.
operator =	Faz uma atribuição de um objeto do tipo tEventMessage, passado como parâmetro, para um *this* da classe tEventMessage.
IsEqual	Verifica se alguns dos atributos do objeto passado como parâmetro são iguais aos mesmos do *this*.

rollback deve ser executado e um estado consistente restaurado. Esses estados são salvos na *StateQueue* de um processo lógico e os elementos dessa lista são removidos quando ocorre um *rollback* (ou quando o GVT é calculado, liberando espaço de memória utilizada). No primeiro caso, todos os estados com tempos virtuais, ($LVT's$) maiores que o tempo de execução da mensagem atrasada (*VirtualRecvTime*), devem ser removidos. No segundo caso, é realizado um *fossil collection* no processo lógico e todos os estados salvos, com tempos virtuais menores que o valor do GVT, podem ser removidos.

Atributos

Esta classe possui um único atributo público, os restantes são privados. O atributo público em questão é uma lista duplamente encadeada de recursos, que representa uma cópia da lista de recursos de um processo lógico em um instante da simulação (um estado). Os demais atributos são descritos na Tabela 4.24.

Métodos

A classe tState possui dois construtores, um construtor *default* iniciando todos os atributos da classe, menos a lista de recursos, com zero. O construtor de cópia cria um novo objeto da

Tabela 4.22: Métodos de tRand.

Nome	Descrição
InitIn	É utilizado para inicializar o vetor *In*.
Ranf	Gera números pseudo-aleatórios no intervalo entre zero e um, utilizando uma distribuição uniforme.
Stream	Este método seleciona um gerador de sementes, ou seja, ajusta qual posição do vetor *In* deve ser acessada.
Seed	Este método atribui a uma posição do vetor *In* um novo valor utilizado para semente e retorna esse valor.
Random	Este método gera um número pseudo-aleatório inteiro, dentro do intervalo determinado pelos parâmetros de entrada do método.
Uniform	Este método gera um número pseudo-aleatório utilizando uma distribuição uniforme.
Expntl	Utiliza a distribuição exponencial para gerar um número pseudo-aleatório.
Erlang	Devolve um número pseudo-aleatório utilizando a distribuição de *Erlang*.
Hyperx	Devolve um número pseudo-aleatório utilizando a distribuição *Hyperx*.
Normal	Devolve um número pseudo-aleatório utilizando uma distribuição normal.
operator ==	Verifica se um tRand passado como parâmetro é igual ou não ao *this*.

classe atribuindo ao novo objeto os atributos do objeto passado como parâmetro. Os principais métodos são apresentados na Tabela 4.24.

4.5.5 Classe tLogicalProcess

A classe tLogicalProcess implementa as características de um processo lógico no protocolo de simulação distribuída *Time Warp*. Cada processo `ETW` implementa uma simulação seqüencial. Além disso, o processo lógico é responsável pela execução de seus eventos na ordem cronológica de suas marcas de tempo e também é responsável pelo envio/recebimento de mensagens/antimensagens para outros processos lógicos. Além da classe tLogicalProcess, a biblioteca ainda possui cinco variáveis globais estáticas, as quais são apresentadas na Tabela 4.25. Todas essas variáveis apresentadas são utilizadas para colocar alguns atributos da classe em seções críticas (exclusão mútua). Isso ocorre porque esses atributos são acessados em mais de uma *thread* na simulação. Essas *threads* são apresentadas na Seção 4.2.

Antes de continuar a descrever a classe, é necessário definir algumas enumerações que a mesma utiliza. A enumeração *tStateSaveType* possui três valores, *Copy*, *Sparse* e *Incremental* e serve para indicar qual o tipo de salvamento de estados que o usuário quer utilizar na simulação. A enumeração *tRollbackType* indica qual o tipo de cancelamento que o usuário quer utilizar na simulação, agressivo ou preguiçoso. é possível a extensão, tanto do salvamento

Tabela 4.23: Atributos da Classe tState.

Nome	Descrição
Process_Id	Identifica um processo lógico.
LVT	Cópia do valor do último instante local de um processo lógico.
NumberOfPrimaryRollbacks	Número de *rollbacks* primários efetuados pelo processo lógico.
NumberOfSecondaryRollbacks	Número de *rollbacks* secundários efetuados pelo processo lógico.
NumberOfMessages	Número de mensagens no processo lógico.
NumberOfMsgSend	Número de mensagens enviadas pelo processo lógico.
NumberOfMsgRecv	Número de mensagens recebidas pelo processo lógico.
NumberOfAMsgSend	Número de antimensagens enviadas pelo processo lógico.
NumberOfAMsgRecv	Número de antimensagens recebidas pelo processo lógico.
InputQueueCurrent	Apontador para o elemento corrente da *InputQueue*.
OutputQueueCurrent	Apontador para o elemento corrente da *OutputQueue*.
AntiQueueCurrent	Apontador para o elemento corrente na *AntiQueue*.
StateQueueCurrent	Apontador para o elemento corrente da *StateQueue*.
Rand	Armazena os últimos valores utilizados para geração de números pseudo-aleatórios e sementes.
TreatMsgRate	Taxa para efetuar o tratamento de mensagens.
FossilCollectionRate	Taxa para efetuar o *fossil collection*.
Granularity	Granulosidade a ser utilizada na execução de eventos.
EventMessage	Mensagem correspondente a aquela que será executada quando o estado está sendo salvo.
RollbackType	Tipo de *rollback* efetuado pelo processo lógico.
Straggler	Evento que causou o último *rollback*.
Current	Evento corrente, sendo executado.
CalcGVT	Indica se o cálculo do GVT será efetuado ou não.
Mode	Utilizada no cálculo do GVT de Samadi.
Coordinator	Processo lógico coordenador, utilizado no cálculo do GVT.
LastGVT	Último GVT calculado.
Num_GVT	Número de $GVTs$ calculados.
NHosts	Número de processos lógicos.
NTours	Número de *tours* na simulação de um modelo fechado.

de estados quanto do cancelamento, bastando o programador apenas implementar o mecanismo e alterar a enumeração correspondente. A enumeração *tMode* indica se um processo está em modo normal (*Normal*) ou em modo de procura (*Find*). Essa última é utilizada no cálculo do GVT.

Tabela 4.24: Métodos de tState.

Nome	Descrição
operator >	Verifica se o *VirtualRecvTime* do atributo *EventMessage* do objeto *this* é maior que do objeto passado como parâmetro.
operator ==	Verifica se o atributo *EventMessage* do *this* é igual ao *EventMessage* do objeto passado como parâmetro.
operator =	Faz uma atribuição do objeto passado como parâmetro para o objeto *this* e retorna um ponteiro para este.

Tabela 4.25: Variáveis globais em lprocess.h.

Nome	Descrição
inputbuffer_mutex	Declaração de um mutex (pthread_mutex_t) para bloquear a InputBuffer, ou seja, colocá-la em uma seção crítica no processo.
unknown_msg_mutex	Declaração de um mutex (pthread_mutex_t) para bloquear a UnKnown_Msg, ou seja, colocá-la em uma seção crítica no processo.
known_msg_mutex	Declaração de um mutex (pthread_mutex_t) para bloquear a Known_Msg, ou seja, colocá-la em uma seção crítica no processo.
NewGVT_mutex	Declaração de um mutex (pthread_mutex_t) para bloquear o atributo GVT, ou seja, colocá-lo em uma seção crítica no processo.
CalcGVT_mutex	Declaração de um mutex (pthread_mutex_t) para bloquear o atributo CalcGVT, ou seja, colocá-lo em uma seção crítica no processo.

Atributos

A classe tLogicalProcess apresenta atributos públicos e privados. Alguns desses atributos representam os elementos necessários, descritos por Jefferson [Jef85], que compõem um processo lógico *Time Warp*, enquanto o restante representa dados estatísticos para controle do processo lógico. A Tabela 4.26 apresenta os atributos de domínio público da classe.

Dentre os atributos privados, existem alguns utilizados como dados estatísticos na simulação distribuída. Outros dois atributos são utilizados para somar o número de eventos que sofreram *rollbacks* primário e secundário. Esses dois atributos são utilizados para calcular o número médio de eventos que sofrem *rollbacks*. Finalmente, três atributos são utilizados para armazenar os tempos em que um processo lógico executou salvamento de estados, *rollback* primário e *rollback* secundário, como ilustrado na Tabela 4.27.

Existem atributos utilizados para determinar como o progresso da simulação no `ETW` ocorre. O atributo *TreatMsgRate* representa uma taxa utilizada para que a simulação determine se um processo lógico executará um evento ou fará tratamento de mensagens. O *FossilCollection-*

Tabela 4.26: Atributos públicos da classe tLogicalProcess.

Nome	Descrição
ResourceList	Lista de recursos pertencentes a um processo lógico.
InputQueue	Lista de eventos executados e não executados.
LogProcess	Arquivo de *log* do processo lógico.
WtimeInit	Marca de tempo real inicial.
WtimeFinal	Marca de tempo real final.
Known_Msg	Lista de mensagens enviadas e que já foram reconhecidas.
UnKnown_Msg	Lista de mensagens ainda não reconhecidas.

Tabela 4.27: Atributos públicos da classe tLogicalProcess.

Nome	Descrição
NumberOfPrimaryRollbacks	Número de *rollbacks* primários que ocorreram durante a execução da simulação distribuída.
NumberOfSecondaryRollbacks	Número de *rollbacks* secundários que ocorreram durante a execução da simulação distribuída.
NumberOfMessages	Número de mensagens no processo lógico.
NumberOfMsgSend	Número de mensagens enviadas por um processo lógico.
NumberOfMsgRecv	Número de mensagens recebidas por um processo lógico.
NumberOfAMsgSend	Número de antimensagens enviadas por um processo lógico.
NumberOfAMsgRecv	Número de antimensagens recebidas por um processo lógico.
PrimaryRollbackEventSum	Soma do número de eventos que sofreram *rollbacks* primário.
SecondaryRollbackEventSum	Soma do número de eventos que sofreram *rollbacks* secundário.
StateSavingTime	Tempo de salvamento de estados.
PrimaryRollbackTime	Tempo de *rollback* primário.
SecondaryRollbackTime	Tempo de *rollback* secundário.

Rate representa uma taxa utilizada que determina se na simulação será executado um *fossil collection* ou não, para remover as informações armazenadas com valores menores que o GVT. Ainda tem-se o atributo *Granularity*, que é utilizado para determinar a granulosidade de um evento na simulação. onde a granulosidade de um sistema paralelo corresponde ao tamanho das unidades de trabalho submetidas aos processadores.

A Tabela 4.28 apresenta os atributos que determinam a estrutura de dados utilizada por Jefferson para representar um processo lógico *Time Warp*, exceto a *InputQueue* que já foi definida [Jef85].

A *InputBuffer* representa as mensagens que são recebidas por um processo lógico *Time*

Tabela 4.28: Atributos públicos da classe tLogicalProcess.

Nome	Descrição
Process_Id	Identifica o processo lógico *Time Warp*.
LVT	Identifica o valor do *Local Virtual Time* (relógio local) de um processo lógico.
OutputQueue	Lista de antimensagens enviadas de um processo lógico para outro.
AntiQueue	Fila de antimensagens recebidas por um processo lógico, antes desse receber a mensagem correspondente.
StateQueue	Fila de estados salvos de um processo lógico.

Warp. O *RollbackType* representa o tipo de cancelamento que o processo lógico executa (agressivo ou preguiçoso), durante a simulação distribuída. O *Straggler* mantém o evento que causou o *rollback* (mensagem atrasada) e o *Current* guarda o evento corrente, que está sendo executado. *ParentOf* representa um evento que será "pai"de outro. Ainda, *Rand* é um atributo utilizado para a geração de números pseudo-aleatórios.

Os atributos utilizados no cálculo do GVT, efetuado através do algoritmo de Samadi, são apresentados na Tabela 4.29.

Tabela 4.29: Atributos para o cálculo do GVT.

Nome	Descrição
CalcGVT	Indica se o cálculo está sendo efetuado ou não.
Mode	Modo de um processo lógico.
Coordinator	Processo lógico coordenador.
Num_GVT	Número de vezes em que o GVT foi calculado.
GVT	Valor do GVT.
NHosts	Número de hosts na simulação.
NTours	O quanto a simulação executa em um modelo fechado.

Métodos

A classe tLogicalProcess contém um construtor *default* que inicializa alguns atributos da classe e um construtor que inicializa o atributo *Process_Id* com o valor passado como parâmetro. Os principais métodos da classe são apresentados na Tabela 4.30.

Tabela 4.30: Métodos de tLogicalProcess.

Nome	Descrição
Init	Inicia o valor do *LVT* do processo lógico com o valor zero, inicializa o atributo *Straggler* e inicia o tempo real da execução da simulação distribuída.
Send	É responsável pelo envio de uma mensagem/antimensagem para outro processo lógico.
Receive_EventMessage	É responsável pelo recebimento de uma mensagem/antimensagem vinda de outro processo lógico.
Receive_KnowMessage	É responsável pelo recebimento de uma mensagem de reconhecimento.
TreatMsgRecv	Retira todos as mensagens da *InputBuffer* para que essas sejam tratadas.
CalcRealTime	Retorna o tempo real final em segundos.
SetIQCurrent	Procura pelo evento com a menor marca de tempo do evento que ainda não foi executado e o ajusta para ser o elemento corrente da lista.
Rollback	Executa tanto um *rollback* primário, quanto um *rollback* secundário.
RecoveredState	Busca na lista de estados o primeiro estado que contém uma marca de tempo menor que do *straggler*.
SendingAntiMessages	O processo lógico deve transmitir antimensagens a todos os eventos na *OutputQueue* com marcas de tempo entre *VirtualRecvTime* do *straggler*/antimensagem e *VirtualSendTime* da mensagem na *OutputQueue*.
UnExecuteMessages_Primary	Tem a finalidade de desfazer todas as execuções incorretas na *Input Queue*, durante um *rollback* primário.
UnExecuteMessages_Secondary	Tem a finalidade de desfazer todas as execuções incorretas na *Input Queue*, durante um *rollback* secundário.
StateSave	Executa o *Copy State Saving*.
StateRestore	Implementa a restauração do estado recuperado.
FossilCollection	É utilizado para realizar o *fossil collection*.

4.6 Exemplo de Uso das Rotinas do ETW e Comparação com *SMPL*

Definidas as classes e estruturas de dados, essa seção apresenta um exemplo do uso do ETW. O objetivo desta seção não é mostrar resultados estatísticos com a ocorrência de *rollbacks* para esse modelo, mas apenas mostrar, com um exemplo, a construção de um modelo com o ETW. Para isso, optou-se pelo modelo da Figura 4.15, o qual representa dois recursos em série, com uma fila e um servidor cada.

A Figura 4.16 ilustra os recursos da Figura 4.15 divididos em processos lógicos. Nesse modelo, o processo lógico PL_0 representa o primeiro recurso e o processo lógico PL_1 representa o segundo. Nessa configuração dos processos lógicos podem ocorrer *rollbacks*, pois o primeiro processo lógico pode enviar alguma mensagem atrasada para o segundo processo lógico.

Para utilizar o ETW, a biblioteca `timewarp.h` deve ser incluída no arquivo que contém

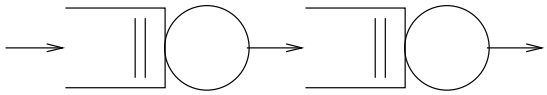


Figura 4.15: Modelo com duas filas em série.

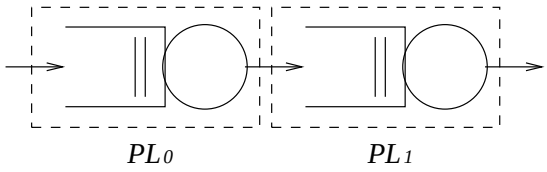


Figura 4.16: Particionamento do modelo da Figura 4.15 em processos lógicos.

o programa de simulação, e as suas primitivas devem ser utilizadas para resolver o modelo por simulação distribuída. Antes de utilizar essas primitivas, o desenvolvedor da simulação deve declarar as constantes e variáveis necessárias para a execução da mesma. Muitas dessas variáveis serão utilizadas como parâmetros de entrada do modelo a ser resolvido. A Figura 4.17 ilustra a parte inicial do código de simulação utilizado para resolver o modelo da Figura 4.16.

```
#include<stdio.h>

#include ¨timewarp.h¨

int main(int argc, char* argv[])
{
  double tc = 3.0, ts1 = 5.0, ts2 = 9.0, te = 10000.0;
  int customer = 1, token, event;
  int nts = 1;
  int CPU;
  int Disk;
  init_simulation(argc, argv, ¨Filas M/M/1 em serie ¨, 2, te, nts);
  CPU = insert_resource(¨recurso_1¨, 1, ts1, 0);
  Disk = insert_resource(¨recurso_2¨, 1, ts2, 1);
  arrival(0, 1, customer, 0.0);
```

Figura 4.17: Declaração dos parâmetros de entrada e iniciação do programa no ETW.

As duas primeiras linhas incluem a biblioteca padrão `stdio.h` e a `timewarp.h`, respectivamente. A função `main` possui dois parâmetros, os quais serão descritos posteriormente. A seguir, as variáveis utilizadas na execução do modelo são declaradas. As variáveis `tc`, `ts1`, `ts2` e `te` indicam o tempo entre chegadas, o tempo de serviço do primeiro recurso, o tempo

de serviço do segundo recurso e o tempo total de execução da simulação, respectivamente. As variáveis `customer`, `token` e `event` são utilizadas durante a execução da simulação, `nts` representa o número de ciclos que uma simulação deve efetuar em um modelo fechado. No caso desse exemplo, `nts` é iniciado com o valor 1 e não é alterado, pois o modelo executado é aberto. A declaração da variável é necessário porque `nts` é um parâmetro de *init_simulation*. `CPU` e `Disk` são variáveis utilizadas para representar os recursos no modelo. `CPU` representa o recurso em PL_0 e `Disk` representa o recurso em PL_1.

A função *init_simulation* é utilizada para iniciar o modelo em dois processos lógicos, com tempo máximo de simulação igual a 10000,0. O número de tarefas, também passado como parâmetro, é utilizado apenas em modelos fechados. Em seguida, dois recursos são inseridos, um em PL_0 e outro em PL_1 e após um primeiro evento de chegada ser escalonado em PL_0, a simulação entra em um laço para a execução de eventos.

A Figura 4.18 ilustra parte do código *SMPL* utilizado para resolver o modelo da Figura 4.16. Nota-se que apenas algumas características são alteradas, comparando o *SMPL* com o `ETW`, como por exemplo o nome da função que inicia a simulação e seus parâmetros e a função que insere recursos na simulação e o número de parâmetros utilizados.

```
#include ¨smpl.h¨

int main()
{
   real tc = 3.0, ts1 = 5.0, ts2 = 9.0, te = 10000.0;
   int customer = 1, token, event;
   int Servidor1;
   int Servidor2;

   smpl(0, ¨Filas M/M/1 em serie¨, te);
   Servidor1 = facility(¨recurso_1¨, 1);
   Servidor2 = facility(¨recurso_2¨, 1);
   schedule(1, 0.0, customer);
```

Figura 4.18: Declaração dos parâmetros de entrada e iniciação do programa no *SMPL*.

A Figura 4.19 descreve a execução da simulação do modelo da Figura 4.16. A simulação segue até que o valor do GVT alcance o tempo máximo para a execução do modelo, ou seja, 10000,0. Assim que o valor do GVT atinge esse valor, o programa de simulação abandona o laço e a função *end_simulation* é utilizada para imprimir os resultados estatísticos e finalizar a simulação. O código para a construção desse modelo é apresentado no Apêndice A.

Para entender o que cada um dos processos lógicos executa, tem-se o *case 2*, o qual requisita o recurso *CPU* para o PL_0 (último parâmetro) e o *case 4*, o qual requisita o recurso *Disk* para o PL_1. Para isso ser feito, cada uma das funções utilizadas verifica se o identificador do processo lógico executando a simulação é igual ao valor passado como parâmetro.

A Figura 4.20 descreve a execução da simulação do modelo da Figura 4.16 utilizando o *SMPL*. Nota-se que os nomes das funções são praticamente os mesmos. Uma alteração ocorre

```
while(get_gvt() < te)
{
  event = get_event(token);
  switch( event )
  {
    case 1:
      schedule(0, CPU, Req, 2, customer, 0.0);
      arrival(0, 1, ++customer, tc);
      break;

    case 2:
      if( request(CPU, event, token, 0, 0) )
        schedule(0, CPU, Rel, 3, token, ts1);
      break;

    case 3:
      release(CPU, token, 0);
      remote_schedule(0, Disk, 4, token, 0.0, 1);
      break;

    case 4:
      if( request(Disk, event, token, 0, 1) )
        schedule(1, Disk, Rel, 5, token, ts2);
      break;

    case 5:
      release(Disk, token, 1);
      break;
  }
}
```

Figura 4.19: Ciclo da execução do programa de simulação no ETW.

quando um evento é gerado para ser executado remotamente. Nesse caso, a função a ser utilizada pelo *SMPL* é *schedule* enquanto que no ETW a função é *remote_schedule*. Além disso, uma função denominada *arrival* é inserida no ETW e é utilizada para escalonar eventos de chegada. As demais funções, como por exemplo *request* e *release* são utilizadas da mesma maneira em ambos os programas de simulação. A função que busca o próximo evento a ser executado no *SMPL* é denominada *cause* enquanto que no ETW é denominada get_event, mas ambas possuem a mesma funcionalidade, ou seja, buscar o evento com a menor marca de tempo para ser executado.

A ativação do programa deve ser realizada utilizando o seguinte conjunto de comandos:

- lamboot -v lamhosts

```
while(time() <= te)
{
  cause(&event, &token);
  if( time() > te )
    break;
  switch( event )
  {
    case 1:
      schedule(2, 0.0, token);
      arrival(1, tc, ++customer);
      break;

    case 2:
      if( request(Servidor1, token, 0) = 0)
        schedule(3, ts1, token);
      break;

    case 3:
      release(Servidor1, token);
      schedule(4, 0.0, token);
      break;

    case 4:
      if( request(Servidor2, token, 0) = 0)
        schedule(5, ts2, token);
      break;

    case 5:
      release(Servidor2, token);
      break;
  }
}
```

Figura 4.20: Ciclo da execução do programa de simulação no *SMPL*.

- mpirun *N* <nome_programa> <arquivo_saida_pl0> ... <arquivo_saida_pln>
- wipe -v lamhosts

Os comandos *lamboot -v lamhosts* e *wipe -v lamhosts* são utilizados para iniciar e encerrar os *daemons* do *LAM-MPI*, respectivamente. O comando *mpirun N <nome_programa> <arquivo_saida_pl0> ... <arquivo_saida_pln>* é utilizado para executar um programa de simulação distribuída utilizando a biblioteca de troca de mensagens *LAM-MPI*, onde *N* especifica que o programa <nome_programa> será executado em todas as máquinas listadas no arquivo

lamhosts e <arquivo_saida_pl0>, ..., <arquivo_saida_pln> representam os arquivos de saída em cada processo na simulação distribuída [Lam04].

4.7 Validação do ETW

A validação do ETW foi efetuada por meio de duas abordagens. A primeira através da análise do comportamento de um processo lógico *Time Warp*, composto da identificação de suas atividades principais, na qual os resultados obtidos com a simulação são comparados com o comportamento esperado do sistema. A segunda através da comparação dos resultados obtidos com a execução de modelos descritos na literatura, utilizando-se a versão original do *SMPL*.

A proposta de tempo virtual de Jefferson [Jef85] prevê o conjunto de ações que um processo lógico deve realizar em resposta a situações durante a simulação distribuída. O ETW foi submetido a esse conjunto de situações e o comportamento observado o valida [Spo01]. A Tabela 4.31 apresenta os resultados obtidos conforme o comportamento esperado.

Pode-se observar que, em nenhum momento, foram utilizadas funções para determinar os parâmetros de entrada para o cancelamento de mensagens, salvamento de estados, para o cálculo do *GVT* e granulosidade. Quando isso ocorre, o núcleo utiliza valores padrão para cada um desses parâmetros. Esses valores indicam que o ETW utiliza como *plugins* o cancelamento agressivo, o *Copy State Saving*, o cálculo do *GVT* de Samadi e granulosidade fina, respectivamente.

Para a execução do modelo da Figura 4.15, foi utilizada uma rede de computadores disponível no Laboratório de Ensino do Mestrado em Ciência da Computação do Departamento de Computação e Estatística da Universidade Federal de Mato Grosso do Sul (DCT-UFMS), com uma taxa de transmissão de 10Mbits/seg. Os computadores utilizados executam o sistema operacional *Linux* com a distribuição *Red Hat*, versão 9.0 e *kernel* 2.4.20-20.9 com a seguinte configuração: Pentium IV 1.8GHz com 256 MB de memória *RAM* e com 30GB de espaço em disco.

Os resultados com *SMPL* e com ETW foram obtidos variando as sementes e utilizando uma distribuição exponencial para cada um dos tempos virtuais. Utilizou-se um tempo entre chegadas igual a 3.0, um tempo de serviço do primeiro recurso igual a 5.0 e um tempo de serviço do segundo recurso igual a 9.0. Observou-se resultados iguais quando comparados o *SMPL* e ETW, validando o núcleo desenvolvido.

4.8 Considerações Finais

Esse capítulo apresentou a implementação das classes utilizadas na construção do ETW e a validação do mesmo, descrevendo uma comparação com o *SMPL* e mostrando o comportamento representativo de um processo lógico *Time Warp*. Além disso, foi apresentado um exemplo de um modelo descrevendo o uso das primitivas do ETW.

A motivação para esse capítulo reside no fato de que a avaliação de desempenho, através de simulação distribuída, é uma tarefa complexa, e a disponibilidade de ferramentas otimistas

Tabela 4.31: Validação do Modelo Representativo do Comportamento de um Processo Lógico *Time Warp*.

Situação	**Comportamento Esperado**
Chegada de uma mensagem e a antimensagem correspondente está armazenada na *Anti Queue*.	Ambas são descartadas.
Chegada de uma mensagem e a antimensagem correspondente não está armazenada na *Anti Queue*.	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 *Input Queue* para posterior execução.
Marca de tempo da mensagem que chegou é menor do que o *LVT* do processo lógico.	O processo lógico imediatamente ativa o procedimento de *rollback*.
Chegada de uma mensagem.	O processo lógico verifica na *Input Queue* 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 *Anti Queue*.
Processo lógico ativa o *rollback* provocado por mensagem atrasada.	O evento correspondente à mensagem é inserido na *Input Queue* 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 anitmensagem voltam para a *Input Queue* para serem re-executados.
Input Queue vazia.	O processo lógico trata as mensagens e/ou antimensagens contidas na *Input Buffer*, se existirem.
Buffer de entrada vazio.	O processo lógico executa os eventos da *Input Queue*, se existirem.

não atinge todos os requisitos necessários à sua utilização. Com isso, o avaliador tem á sua disposição um núcleo *Time Warp*, que pode ser estendido através de adição de *plugins*, pronto para ser utilizado em um ambiente automático, como por exemplo o *ASDA*. O núcleo representado pelo *ETW* permite, além da simulação usando o protocolo otimista *Time Warp*,

a inserção de novos *plugins*. Também é contribuição deste capítulo a validação do modelo de *plugins* apresentado em [Spo01], que definiu o núcleo básico e *plugins*.

O fato de o `ETW` manter a estrutura de um programa *SMPL* torna-o de fácil entendimento e utilização. Assim, o desenvolvedor que já está acostumado com a simulação orientada a eventos necessita apenas do conhecimento sobre a simulação distribuída para fazer uso do `ETW`.

O Capítulo 5 mostra o desenvolvimento de modelos utilizando o `ETW`, comparando-o com a versão seqüencial em *SMPL*, além de análises com diferentes granulosidades e modelos.

Capítulo 5

Estudos de Caso

5.1 Considerações Iniciais

O ETW foi utilizado em estudos de caso, para analisar o desempenho da simulação distribuída comparada à simulação seqüencial e também para verificar o comportamento de diversas métricas da simulação distribuída em relação a diferentes situações do modelo e arquitetura. Com isso, pode-se mostrar a potencialidade do uso do ETW na avaliação de sistemas de filas.

As análises foram efetuadas com diferentes situações da granulosidade de eventos: fina, média e grossa. Com isso, pode-se verificar como o desempenho da simulação distribuída depende também da carga de trabalho atribuída aos processos lógicos (execução de eventos). No estudo comparativo da simulação seqüencial e da simulação distribuída a análise foi efetuada através da comparação dos tempos de execução obtidos com o ETW e a correspondente execução seqüencial, isto é, foram executadas versões seqüenciais, utilizando o *SMPL* para análise dos tempos de simulação. As métricas utilizadas para avaliação de desempenho da simulação distribuída foram:

- tempo de simulação seqüencial: tempo (em segundos) de execução do programa de simulação em um único elemento de processamento, utilizando o *SMPL*;
- tempo de simulação distribuída: tempo (em segundos) de execução do programa de simulação no ETW em uma máquina paralela ou sistema distribuído com n elementos de processamento.

Os resultados obtidos nos testes forneceram dados para uma análise referente à utilização do ETW. Os valores das variáveis estatísticas obtidas com as execuções dos modelos hipotéticos, utilizando diferentes granulosidades, fornecem informações que podem indicar como está o comportamento do ETW. Dentre essas variáveis, destacam-se:

- comprimento médio de *rollbacks*;
- tempo efetuando *rollbacks* primário e secundário;

- tempo executando salvamento de estados.

A Figura 5.1 ilustra a classificação das plataformas computacionais em função da comunicação e processamento, apresentada por Morselli [Mor00]. A categoria **1** representa plataformas com um alto desempenho na comunicação e um baixo desempenho de processamento. A categoria **2** representa plataformas com alto desempenho na comunicação e alto desempenho de processamento. Na categoria **3** estão representadas plataformas com comunicação e processamento de baixo desempenho. As plataformas com baixo desempenho na comunicação e alto desempenho de processamento são representativas da categoria **4**. É nessa categoria que enquadra-se a rede fisicamente instalada no Laboratório de Mestrado em Ciência da Computação da Universidade Federal de Mato Grosso do Sul, utilizada neste trabalho, embora o ideal para a realizao dos testes com o `ETW` sejam as categorias **2** e **3**.

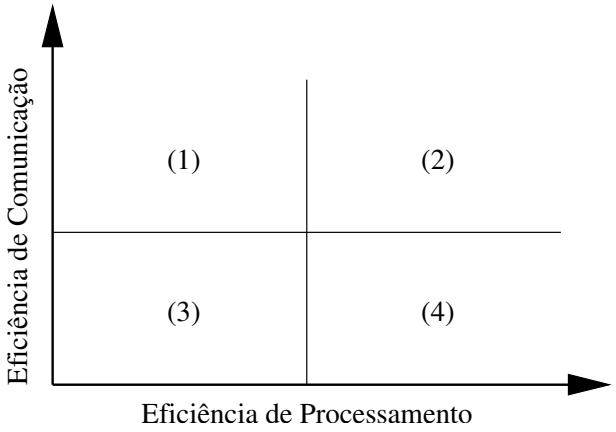


Figura 5.1: Diagrama Comunicação x Processamento.

Além disso, com o intuito de garantir a precisão dos resultados obtidos, as simulações foram executadas utilizando uma rede isolada, de forma que outros usuários não estivessem competindo pela rede e processadores. A Figura 5.2 ilustra um exemplo de configuração da plataforma de processamento e comunicação utilizada em todos os modelos. As estações de trabalho (*workstations*) utilizadas para a execução da simulação têm processadores *Pentium IV* com 1.8 *GHz* de *clock*, 256 *MB* de memória *RAM* e disco com capacidade de armazenamento de 30 *GB*, com sistema operacional *Linux* distribuição *Red Hat* versão 9.0, *kernel* $2.4.20-20.9$ e *LAM-MPI* versão 7.0.4 para efetuar troca de mensagens. A comunicação é efetuada através de uma rede *Ethernet* de 10 *Mbps*. Essa plataforma é representativa da categoria **4**, com baixa capacidade de comunicação e alta capacidade de processamento, conforme definido por Morselli [Mor00].

A Seção 5.2 descreve os modelos utilizados para a execução das simulações, necessárias para a obtenção dos dados para a análise. A Seção 5.3 descreve a análise do desempenho da simulação distribuída, quando comparada à simulação seqüencial. A Seção 5.4 descreve as análises das métricas efetuadas com a simulação distribuída `ETW`. A Seção 5.5 apresenta as considerações finais do capítulo.

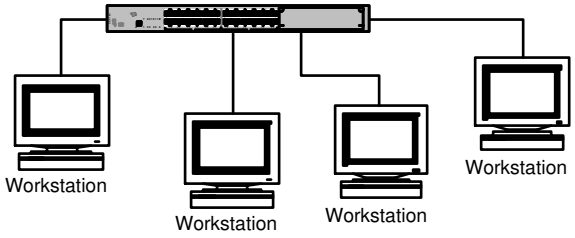


Figura 5.2: Rede de interconexão.

5.2 Descrição dos Modelos

Foram utilizados três modelos para os estudos de caso com o `ETW` e para efetuar as comparações dos resultados obtidos com as execuções com diferentes granulosidades e partições dos processos lógicos. O modelo da Figura 5.3 (Modelo Hipotético 1) possui dois recursos em série, denominados *recurso_1* e *recurso_2*, e é utilizado para a comparação dos tempos de simulação seqüencial e distribuída. Para esse modelo, são utilizadas duas variações, alterando os parâmetros de entrada do mesmo (todos expressos em uma unidade de tempo *ut* que, para simplificar a notação, será omitida no restante do texto). Na variação 1, são utilizados como parâmetros de entrada um tempo entre chegadas igual a 3,0, um tempo de serviço do *recurso_1* igual a 5,0 e um tempo de serviço do *recurso_2* igual a 9,0. Na variação 2 utiliza-se como parâmetros de entrada um tempo entre chegadas igual a 5,0, um tempo de serviço do $recurso_1$ igual a 3,0, um tempo de serviço do $recurso_2$ igual a 12,0 e um tempo de execução igual a 10000,0. Esses valores são fictícios e são utilizados apenas para efeitos de teste devido ao fato de que todos os modelos já haviam sido executados utilizando o *SMPL*. O modelo foi executado com dois processos lógicos, como apresentado na Figura 5.4.

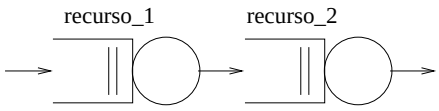


Figura 5.3: Modelo Hipotético 1.

Os clientes (tarefas) são escalonados para a utilização do *recurso_1* de acordo com a taxa de chegada. Se esse recurso estiver ocupado, o cliente ocupa a fila do mesmo, mas caso esteja disponível, retira uma requisição da fila e a executa. Após o *recurso_1* terminar a execução da tarefa, essa é escalonada para a utilização do *recurso_2* e após a execução, essa deixa o sistema.

O modelo da Figura 5.5, Modelo Hipotético 2, é composto por dois recursos, denominados *recurso_1* e *recurso_2*. Também foi executado utilizando simulação seqüencial e distribuída com diferentes granulosidades e processamentos. Esse modelo possui um tempo entre chegadas igual a 2,5, um tempo de serviço do *recurso_1* igual a 1,5, um tempo de serviço do *recurso_2*

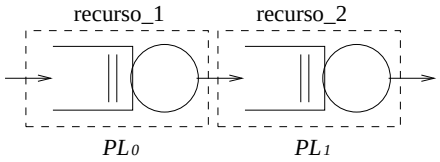


Figura 5.4: Particionamento do modelo da Figura 5.3.

igual a 4,5 e um tempo de execução igual a 1000,0. Além disso, utilizou-se 50% de probabilidade de um evento ser liberado entre os dois recursos ou seguir para o *recurso_2*. A Figura 5.6 apresenta o particionamento do modelo.

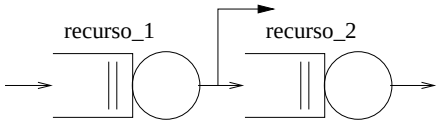


Figura 5.5: Modelo Hipotético 2.

Os clientes (tarefas) são escalonados para a utilização do *recurso_1* de acordo com a taxa de chegada. Se esse recurso estiver ocupado, o cliente ocupa a fila do mesmo, mas caso esteja disponível, retira uma requisição da fila e a executa. Após o *recurso_1* terminar a execução da tarefa, essa pode deixar o sistema ou ser solicitado para o *recurso_2*. Nesse caso, uma tarefa é escalonada para a utilização do *recurso_2* e após a execução, essa sai do sistema.

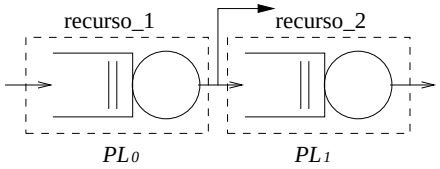


Figura 5.6: Particionamento do modelo da Figura 5.5.

O modelo da Figura 5.7 (Modelo Hipotético 3) possui três recursos, denominados *recurso_1*, *recurso_2* e *recurso_3*. Esse modelo possui um tempo entre chegadas igual a 5,0, um tempo de serviço do *recurso_1* igual a 3,0, um tempo de serviço do *recurso_2* igual a 9,0, um tempo de serviço do *recurso_3* igual a 10,0 e um tempo de execução igual a 10000,0. Além disso, utilizou-se 50% de probabilidade de um evento requisitar os recursos 2 ou 3. O modelo foi executado com três processos lógicos, como apresentado na Figura 5.8.

Os clientes (tarefas) são escalonados para a utilização do *recurso_1* de acordo com a taxa de chegada. Se esse recurso estiver ocupado, o cliente ocupa a fila do mesmo, mas caso esteja disponível, retira uma requisição da fila e a executa. Após o *recurso_1* terminar a execução da tarefa, essa pode ser encaminhada para *recurso_2* ou para *recurso_3*, dependendo do valor

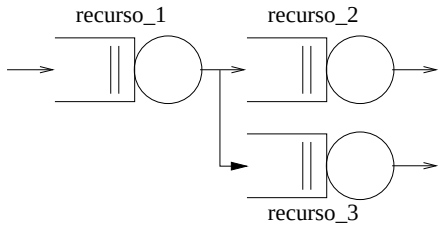


Figura 5.7: Modelo Hipotético 3.

gerado utilizando uma probabilidade de 50%. Nesse caso, uma tarefa é escalonada para a utilização do *recurso_2* ou do *recurso_3* e após a execução, a tarefa deixa o sistema.

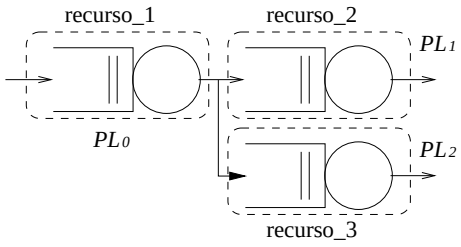


Figura 5.8: Particionamento do modelo da Figura 5.7.

Para os estudos com os modelos hipotéticos 1, 2 e 3, foram realizadas execuções alternando entre granulosidade de eventos fina, média e grossa, para uma plataforma com um processamento denominado rápido, utilizando máquinas *Pentium IV* da categoria **4**, com taxa de transmissão a $10Mbps$. Como não havia uma plataforma de outra categoria disponível, foi inserido, apenas para efeitos de teste, um atraso na execução dos eventos da simulação distribuída e essa situação foi denominada processamento lento.

5.3 Análise de Desempenho da Simulação Distribuída

As análises foram realizadas por meio da comparação dos resultados obtidos na simulação distribuída com a correspondente simulação seqüencial, sendo necessária a implementação de versões seqüenciais do programa de simulação. Os programas seqüenciais foram implementados utilizando a extensão funcional *SMPL*.

Para garantir que os dados obtidos estivessem estatisticamente corretos, foram realizadas 15 amostras (quinze execuções) com distribuições exponenciais para o tempo entre chegadas e tempos de serviço dos recursos nos modelos.

Os valores médios obtidos com a execução do Modelo Hipotético 1 são apresentados na Tabela 5.1. Esses dados indicam *speedup*, tempo de execução da simulação seqüencial (**TSeq**) e tempo de execução da simulação distribuída (**TDistr**) com granulosidades fina, média e grossa utilizando a plataforma *Pentium IV* descrita na Seção 5.1.

Tabela 5.1: *Speedup* - Modelo 1 - Variação 1 - Processamento Rápido.

Granulosidade	***Speedup***	**TSeq**(s)	**TDistr**(s)
Fina	$8,5419 \times 10^{-05}$	0,1559	1825,7015
Média	0,0020	3,8667	1931,1212
Grossa	0,0453	214,4000	4734,6229

A execução seqüencial demonstra ser mais rápida que a distribuída, pois esse modelo é muito simples e o desempenho é baixo devido à troca de mensagens, tanto no envio de mensagens para a execução remota quanto para o cálculo do *GVT*. Portanto, não há ganho de desempenho em paralelizar um modelo tão simples, como era esperado.

A Figura 5.9 ilustra o relacionamento entre as médias dos resultados obtidos para o modelo da Figura 5.3 com processamento rápido e granulosidades fina, média e grossa. Observa-se que o desempenho da simulação distribuída tende a melhorar conforme a granulosidade aumenta, devido a menor sobrecarga da troca de mensagens em relação à execução de eventos.

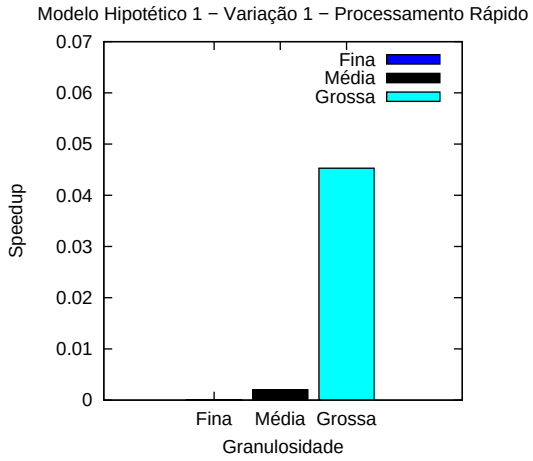


Figura 5.9: Relação *Speedup* x Granulosidades.

A Tabela 5.2 apresenta execuções do mesmo modelo mas com a inserção de um atraso no processamento de eventos, no valor de 10 micro-segundos. Observa-se, novamente, que o modelo, por ser simples, apresenta um tempo real de simulação seqüencial com um desempenho melhor que a simulação distribuída. Os dados representam *speedup*, tempo de execução da simulação seqüencial (TSeq) e tempo de execução na simulação distribuída (TDistr).

A Figura 5.10 ilustra o relacionamento entre o *speedup* obtido e as granulosidades fina, média e grossa. Comparando o gráfico da Figura 5.9 com o da Figura 5.10, nota-se tendência de melhora no desempenho do `ETW`, isto é, diminuindo a capacidade de processamento e aumentando a granulosidade, o desempenho da simulação distribuída melhora, mas para esse modelo simples o ganho não é suficiente para obter um bom *speedup*.

Tabela 5.2: *Speedup* - Modelo 1 - Variação 1 - Processamento Lento.

Granulosidade	***Speedup***	**TSeq**(s)	**TDistr**(s)
Fina	0,0938	297,0000	3166,8026
Média	0,09139	297,0667	3250,6555
Grossa	0,1193	594,0667	4980,5734

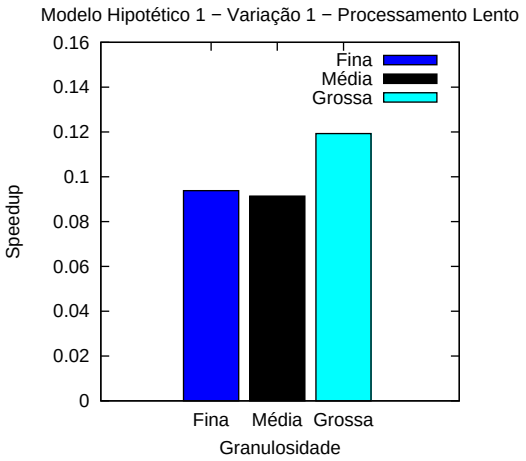


Figura 5.10: Relação *Speedup* x Granulosidades.

No Apêndice C são analisados estatisticamente os dados obtidos para o modelo hipotético 1, variação 1, como executado nas arquiteturas com processamentos rápido e lento.

Para o mesmo modelo, foram utilizadas duas variações nos parâmetros de entrada. A Tabela 5.3 apresenta os resultados obtidos na execução do modelo 1 utilizando uma variação em seus parâmetros de entrada, como descrito anteriormente. De acordo com essa tabela, mesmo alterando os parâmetros a simulação seqüencial apresenta um desempenho melhor, comparado ao desempenho da simulação distribuída. Esse desempenho novamente é prejudicado pela execução de eventos utilizando um processamento rápido e pela baixa taxa de transmissão de mensagens na rede de comunicação. Além disso, o modelo é simples e conta apenas com dois processos lógicos.

Tabela 5.3: *Speedup* - Modelo 1 - Variação 2 - Processamento Rápido.

Granulosidade	***Speedup***	**TSeq**(s)	**TDistr**(s)
Fina	0,0004	0,1239	316,3330
Média	0,01238	4,3333	350,0072
Grossa	0,0746	242,06667	3243,0864

A Figura 5.11 apresenta o valor do *speedup* obtido a partir da Tabela 5.3. Da mesma forma que no caso anterior, a maior granulosidade de eventos permitiu a obtenção de melhor resultado, com relação às outras granulosidades.

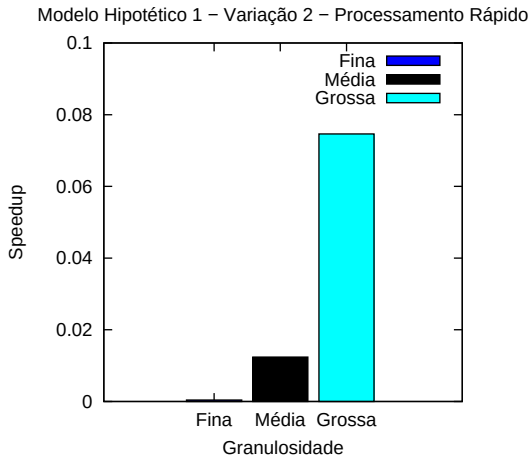


Figura 5.11: Relação *Speedup* x Granulosidades.

A Tabela 5.4 apresenta a execução dessa segunda variação do modelo com a inserção de atraso no processamento lento. Mesmo não obtendo um valor considerável comparando o *speedup*, nota-se uma melhora no desempenho quando os dados obtidos são comparados com os dados da Tabela 5.3.

Tabela 5.4: *Speedup* - Modelo 1 - Variação 2 - Processamento Lento.

Granulosidade	***Speedup***	**TSeq**(s)	**TDistr**(s)
Fina	0,0799	216,7333	2713,0497
Média	0,0799	216,7333	2711,9180
Grossa	0,1211	433,5333	3580,0683

Um conjunto significativo de simulações executadas apresentou *speedups* com valores muito inferiores a 1,0. Esse comportamento mostra que a sobrecarga gerada pelo sincronismo da simulação distribuída dificulta a utilização do paralelismo. Mesmo com esses resultados obtidos com o *speedup*, eles são importantes no auxílio para caracterizar a eficiência no sistema de comunicação utilizado e os diferentes processamentos (rápido e lento). A Figura 5.12 ilustra um gráfico comparando o *speedup* obtido com as diferentes granulosidades nas execuções do modelo. Nota-se que, para esse modelo simples, a utilização de um processamento mais lento e uma granulosidade grossa implicam em um aumento no *speedup* (pois há um balanceamento entre o custo de comunicação e de processamento).

A Tabela 5.5 apresenta os resultados obtidos com a execução do modelo hipotético 2, com um processamento rápido. Os resultados mostram que a execução com uma granulosidade

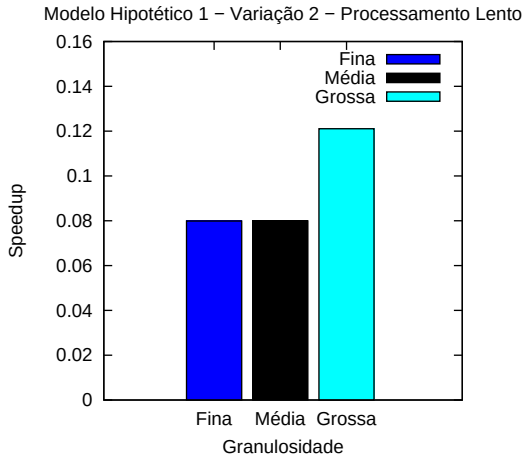


Figura 5.12: Relação *Speedup* x Granulosidades.

mais grossa permite que a simulação distribuída alcance um melhor desempenho, quando comparada com as demais granulosidades, para esse modelo.

Tabela 5.5: *Speedup* - Modelo Hipotético 2 - Processamento Rápido.

Granulosidade	***Speedup***	**TSeq**(s)	**TDistr**(s)
Fina	$2,3644 \times 10^{-05}$	0,0014	60,2345
Média	0,0105	0,9536	90,5210
Grossa	0,0602	44,4000	737,3454

Para o modelo hipotético 2, a Figura 5.13 ilustra os valores de *speedup* obtidos executando com um processamento rápido e com granulosidades fina, média e grossa. Verifica-se aumento no *speedup* observado, devido à menor sobrecarga de comunicação imposta pelo modelo com granulosidade grossa.

A Tabela 5.6 apresenta os resultados obtidos com a execução do modelo hipotético 2, com um processamento lento. Os valores obtidos com o *speedup* não são os ideais para que a simulação distribuída seja mais eficiente que a simulação seqüencial e devido ao fato que o modelo analisado é pequeno.

Os testes efetuados com o modelo hipotético 2 também, como esperado, não indicaram a simulação distribuída como mais rápida em relação à simulação seqüencial devido ao fato que esse modelo é simples. A Figura 5.14 ilustra os dados obtidos executando o modelo com um processamento lento e com granulosidades fina, média e grossa.

A Tabela 5.7 apresenta os resultados obtidos com a execução do modelo hipotético 3, com um processamento rápido. Novamente os resultados utilizando granulosidade grossa mostram-se com um melhor desempenho, quando comparado com outras granulosidades (o excesso de otimismo dos processos lógicos é limitado). Apesar disso, a simulação distribuída apresenta

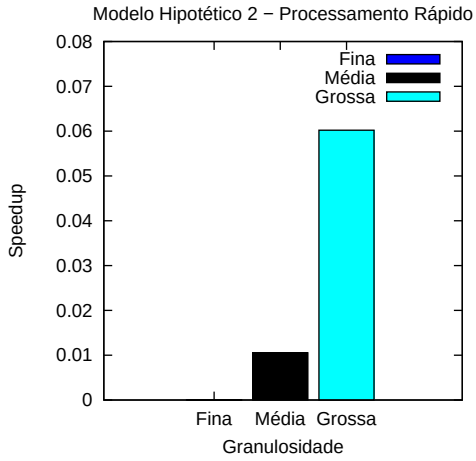


Figura 5.13: Relação *Speedup* x Granulosidades.

Tabela 5.6: *Speedup* - Modelo Hipotético 2 - Processamento Lento.

Granulosidade	***Speedup***	**TSeq**(s)	**TDistr**(s)
Fina	0,0703	39,6000	563,1141
Média	0,0615	39,6000	643,7238
Grossa	0,0981	79,2667	807,6411

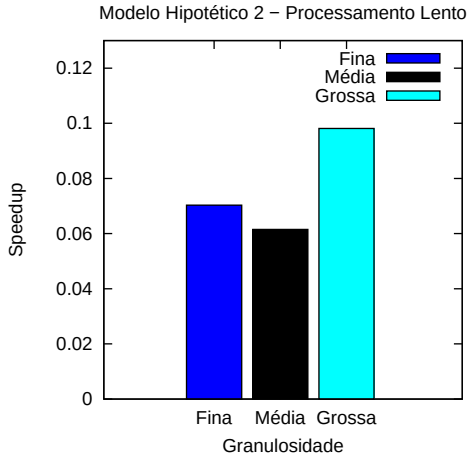


Figura 5.14: Relação *Speedup* x Granulosidades.

um desempenho muito pequeno em relação à simulação seqüencial. Isso deve-se ao fato de que o modelo estudado ainda é pequeno e que os custos com comunicação e sincronização são dominantes, em relação ao processamento de eventos.

Tabela 5.7: *Speedup* - Modelo Hipotético 3 - Processamento Rápido.

Granulosidade	***Speedup***	**TSeq**(s)	**TDistr**(s)
Fina	$2,3286 \times 10^{-05}$	0,0056	240,9161
Média	0,0132	4,4004	333,5411
Grossa	0,0532	269,4000	5061,9906

A Figura 5.15 ilustra o relacionamento entre o *speedup* obtido e as granulosidades fina, média e grossa. Os valores de *speedup* obtidos mostram resultados parecidos quando esses são comparados aos modelos anteriores.

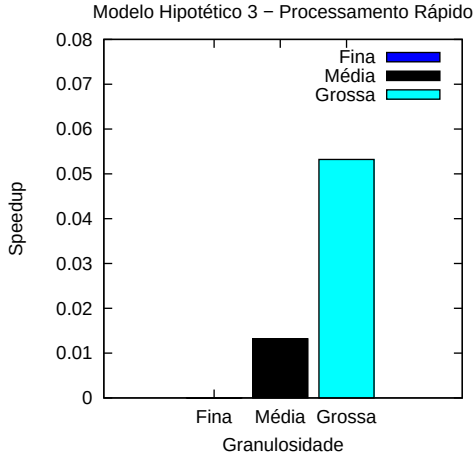


Figura 5.15: Relação *Speedup* x Granulosidades.

A Tabela 5.8 apresenta os resultados obtidos com a execução do modelo hipotético 3, com um processamento lento. Comparando esses resultados com os resultados apresentados na Tabela 5.7, o processamento lento mostrou-se mais eficiente que o processamento rápido nas granulosidades fina, média e grossa, pois há uma menor sobrecarga no número de mensagens trocadas entre os processos lógicos.

Tabela 5.8: *Speedup* - Modelo Hipotético 3 - Processamento Lento.

Granulosidade	***Speedup***	**TSeq**(s)	**TDistr**(s)
Fina	0,0420	235,6667	5608,0554
Média	0,0496	236,9333	4775,6317
Grossa	0,0814	471,3333	5788,5385

A Figura 5.16 ilustra o relacionamento entre o *speedup* obtido e as granulosidades fina, média e grossa. Comparando o gráfico da Figura 5.15 com o da Figura 5.16, nota-se uma

melhora no desempenho do ETW, isto é, a tendência é de que com a diminuição da capacidade de processamento e o aumento da granulosidade, o desempenho da simulação distribuída melhora (considerando a mesma rede de comunicação), pois assim reduz-se o impacto do custo da comunicação no desempenho (já que o excesso de otimismo dos processos lógicos é limitado).

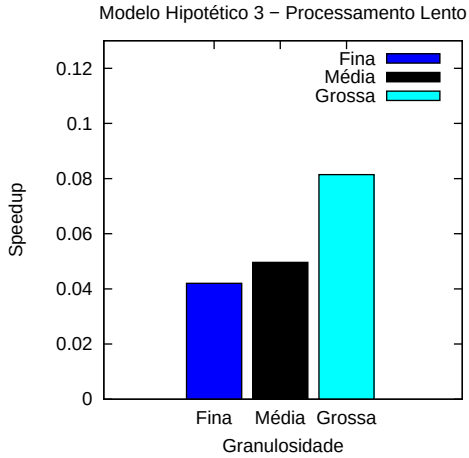


Figura 5.16: Relação *Speedup* x Granulosidades.

Na análise do *speedup*, os valores obtidos eram esperados, visto que foram analisados modelos pequenos, nos quais a sobrecarga de comunicação (mensagens e antimensagens, além do cálculo do GVT) é agravada. Além disso, observa-se, nos resultados obtidos com as execuções desses modelos hipotéticos, uma leve melhora nos valores do *speedup*, tanto no aumento da granulosidade quanto na utilização do processamento lento. Isso ocorre por que o tempo de execução desses modelos na simulação seqüencial tende a aumentar mais, proporcionalmente, em relação ao tempo de execução desses modelos na simulação distribuída.

Os testes efetuados por Morselli [Mor00] e Ulson [Uls99] nesse tipo de plataforma, com sincronização conservativa e modelos com pequeno número de processos lógicos, obtiveram resultados parecidos. Em outras plataformas ou com modelos maiores foi obtido *speedup*.

5.4 Análise de Métricas da Simulação Distribuída

Esta seção é utilizada para analisar o comportamento do ETW com as métricas apresentadas na Seção 5.1, com processamentos e granulosidades diferentes em 4 situações, com 3 modelos hipotéticos. A importância de analisar cada uma dessas métricas é que essas podem indicar características fundamentais no comportamento do ETW e auxiliar no mecanismo de troca de protocolos sugerido por Morselli [Mor00], o qual permite a mudança de protocolo de sincronização (*CMB* ou *Time Warp*) em tempo de execução.

5.4.1 Modelo Hipotético 1

É realizada uma análise das métricas descritas na Seção 5.1, para o modelo da Figura 5.3, com a variação 1, utilizando processamentos rápido e lento e granulosidades fina, média e grossa. Como descrito, esse modelo possui dois recursos, com uma fila e um servidor cada. Além disso, para a execução desse modelo, o particionamento do mesmo foi feito em dois processos lógicos, PL_0 e PL_1.

Comprimento Médio de *Rollbacks* Primários

O comprimento médio de *rollbacks* primários é resultado da divisão da soma de eventos que sofrem *rollbacks* primários pelo número total de *rollbacks* primários efetuados durante uma simulação. A Figura 5.17 ilustra o comprimento médio de *rollback* primário durante a execução da simulação distribuída para esse modelo utilizando duas variações, com processamento rápido e granulosidades fina, média e grossa. Observa-se, em ambas as variações, que o comprimento médio de *rollbacks* primários tende a diminuir com o aumento da granulosidade e que em ambas as granulosidades a variação 1 possui valores menores para essa métrica. Esses resultados são influenciados pelos parâmetros de entrada utilizados nessas variações.

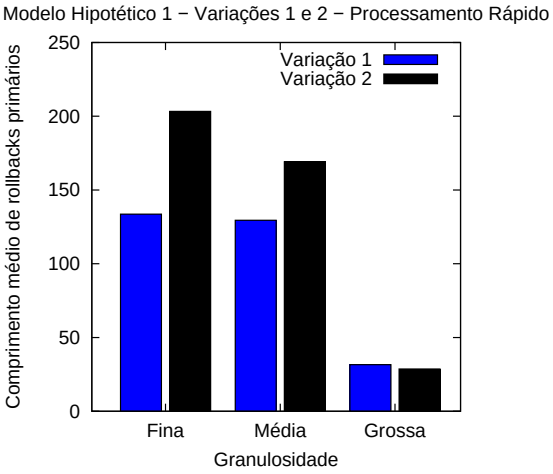


Figura 5.17: Modelo hipotético 1, processamento rápido: Comprimento médio de *rollbacks* primários X Granulosidade.

A Tabela 5.9 apresenta os valores obtidos com o comprimento médio de *rollbacks* primários, com processamento rápido. Observa-se que o comprimento médio de *rollbacks* primários tende a diminuir com o aumento da granulosidade em ambas as variações.

A Figura 5.18 ilustra o comprimento médio de *rollback* primário durante a execução da simulação distribuída para o modelo, utilizando duas variações, com processamento lento e granulosidades fina, média e grossa. Os resultados mostram que o comprimento médio de *rollbacks* primários tende a diminuir conforme a granulosidade aumenta e o processamento torna-se mais lento. Os resultados mostram uma grande quantidade de eventos que sofreram

Tabela 5.9: Comprimento de *rollbacks* primários: variações 1 e 2, processamento rápido.

Granulosidade	**Variação 1**	**Variação 2**
Fina	133,6061	203,3078
Média	129,4975	169,0757
Grossa	31,5388	28,6448

rollback primário em um processamento rápido. Enquanto que, para um processamento lento, um número menor de eventos sofrem *rollback* primário. Essas diferenças ocorrem por que há um maior equilíbrio entre comunicação e processamento de eventos, aumentando o número de *rollbacks* primários efetuados. Desse modo, ocorre uma diminuição no excesso de otimismo na execução da simulação distribuída no processo lógico, ocasionando uma diminuição no comprimento médio de *rollbacks* primários. Isso traz como resultado um número maior de *rollbacks* primários sendo executados, mas um número menor de eventos sendo desfeitos, melhorando o *speedup*.

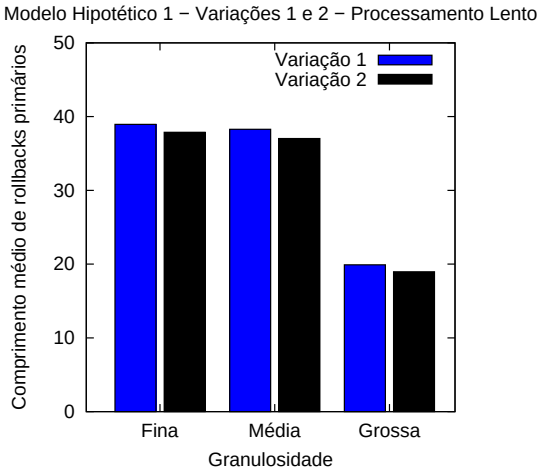


Figura 5.18: Modelo hipotético 1, processamento lento: Comprimento médio de *rollbacks* primários X Granulosidade.

A Tabela 5.10 apresenta os valores obtidos com o comprimento médio de *rollbacks* primários utilizando um processamento lento. Observa-se que o comprimento médio de *rollbacks* primários tende a diminuir com o aumento da granulosidade em ambas as variações.

Tempo Executando *Rollback* Primário

O tempo utilizado para a execução de *rollbacks* primários indica o tempo real em que um programa de simulação utiliza para cancelar mensagens erradas, restaurar um estado consistente e inserir a mensagem atrasada na *Input Queue*. Esse tempo pode influenciar de

Tabela 5.10: Comprimento de *rollbacks* primários: variações 1 e 2, processamento lento.

Granulosidade	**Variação 1**	**Variação 2**
Fina	38,9433	37,8802
Média	38,2885	37,0267
Grossa	19,9038	18,9678

modo significativo o desempenho de um programa de simulação distribuída. O aumento no tempo em que um programa executa *rollbacks* primários pode aumentar o tempo de execução de um modelo na simulação distribuída.

A Figura 5.19 ilustra o tempo em que a simulação distribuída executa *rollbacks* primários no modelo, para as variações 1 e 2, com processamento rápido e granulosidades fina, média e grossa. Para a variação 1, os resultados mostram um tempo menor executando *rollbacks* primários para uma granulosidade mais grossa. Isso indica que, conforme a granulosidade aumenta, o tempo em que a simulação executa *rollbacks* primários tende a diminuir, pois o aumento da granulosidade retarda a execução dos eventos, limitando o otimismo durante a execução da simulação distribuída. Para a variação 2, as granulosidades fina e média mostraram baixos tempos com a simulação executando *rollbacks* primários, enquanto que a granulosidade grossa apresenta um tempo maior de execução de *rollbacks* primários. Esses resultados mostram que, conforme a granulosidade aumenta, o tempo de execução de *rollbacks* primários tende a crescer também. Isso deve-se ao fato de que, embora o comprimento médio de *rollbacks* primários diminua com o aumento na granulosidade, o número de *rollbacks* primários tende a crescer muito, aumentando o tempo de execução desses.

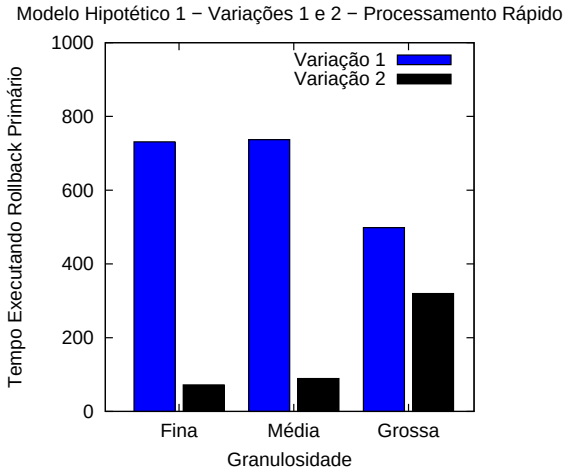


Figura 5.19: Modelo hipotético 1, processamento rápido: Tempo executando *rollback* primário X Granulosidade.

A Tabela 5.11 apresenta os valores obtidos com a execução do modelo hipotético 1 com as

duas variações. Com esses valores observa-se as mesmas conclusões obtidas com a análise da Figura 5.19.

Tabela 5.11: Tempo executando *rollbacks* primários: variações 1 e 2, processamento rápido.

Granulosidade	**Variação 1**	**Variação 2**
Fina	731,0826	72,0371
Média	736,9171	89,0648
Grossa	498,3512	319,7797

A Figura 5.20 ilustra o tempo em que a simulação distribuída executa *rollbacks* primários no modelo utilizando duas variações, com processamento lento e granulosidades fina, média e grossa. Para a variação 1, os resultados obtidos com processamento lento mostram que o tempo em que a simulação distribuída executa *rollbacks* primários tende a diminuir conforme a granulosidade aumenta e com a utilização do processamento lento. Os valores dos parâmetros de entrada, descritos na Seção 5.2, influenciam nesses resultados, pois a chegada de muitos clientes em PL_1 resulta na ocorrência de muitos *rollbacks* primários.

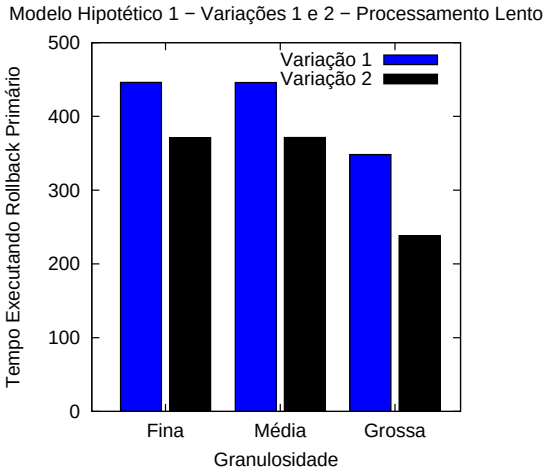


Figura 5.20: Modelo hipotético 1, processamento lento: Tempo executando *rollback* primário X Granulosidade.

Para a variação 2, os resultados indicam que o tempo executando *rollbacks* primários também tende a diminuir com o aumento da granulosidade. Além disso, comparando os processamentos rápido e lento, observa-se também que essa métrica diminui com granulosidade grossa. As características dos parâmetros desse modelo indicam que não chegam tantos eventos para serem executados em PL_1 como na variação 1. No processamento rápido, a granulosidade grossa não foi suficiente para diminuir o impacto causado pela chegada de clientes ao sistema (dependente dos parâmetros da aplicação). Mas como o processamento é lento, o tempo gasto

em *rollbacks* primários na granulosidade grossa é menor do que nas outras granulosidades, pois a simulação distribuída atinge um equilíbrio entre comunicação e processamento de eventos.

A Tabela 5.12 apresenta os valores obtidos com a execução do modelo hipotético 1 com as duas variações, para um processamento lento. Observa-se as mesmas conclusões feitas com a análise da Figura 5.20.

Tabela 5.12: Tempo executando *rollbacks* primários: variações 1 e 2, processamento lento.

Granulosidade	**Variação 1**	**Variação 2**
Fina	446,0528	371,1955
Média	445,9269	371,4488
Grossa	348,3431	238,3245

Tempo Executando Salvamento de Estados

O salvamento de estados na simulação distribuída otimista é necessário para desfazer as computações erradas caso uma mensagem atrasada seja recebida por um processo lógico. Isso significa que qualquer programa de simulação distribuída otimista gasta uma porção de tempo executando salvamento de estados. A Figura 5.21 ilustra o tempo em que um programa de simulação distribuída executa salvamento de estados no modelo com processamento rápido, granulosidades fina, média e grossa e com duas variações, 1 e 2, no PL_0. Para a variação 1, os resultados obtidos demonstram que o tempo em que a simulação distribuída executa o salvamento de estados tende a diminuir quando utiliza-se uma granulosidade mais grossa.

Para a variação 2, os resultados obtidos mostraram que o tempo de salvamento de estados também tende a diminuir com o aumento da granulosidade. Além disso, esses resultados mostraram que, na execução da variação 1, o tempo de salvamento de estados é maior, em PL_0. A diferença ocorre devido aos valores dos parâmetros de entrada utilizados, os quais influenciam nos resultados.

A Figura 5.22 ilustra o tempo em que um programa de simulação distribuída executa salvamento de estados no modelo com processamento rápido, granulosidades fina, média e grossa e com duas variações, 1 e 2, no PL_1. Na variação 1, os resultados obtidos novamente mostraram que o tempo em que a simulação distribuída executa o salvamento de estados tende a diminuir quando utiliza-se uma granulosidade mais grossa. Enquanto que, na variação 2, os resultados obtidos mostraram que o tempo de salvamento de estados tende a crescer com o aumento da granulosidade. Observa-se que PL_1, nessa variação 2, executa o salvamento de estados em um tempo maior, pois PL_1 executa *rollbacks* primários decorrentes de execuções incorretas de eventos e essas execuções não são descartadas no cálculo do tempo de salvamento de estados em PL_1, ou seja, para o tempo de salvamento de estados, PL_1 acumula estados salvos inutilmente.

A Tabela 5.13 apresenta os valores obtidos com a execução do modelo hipotético 1 com as duas variações, para um processamento rápido, representando as Figuras 5.21 e 5.22.

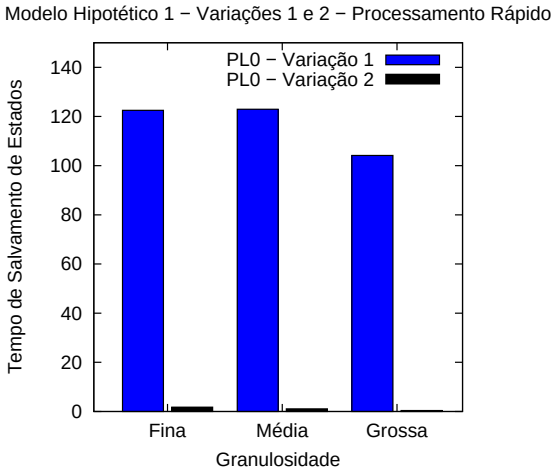


Figura 5.21: Modelo hipotético 1, processamento rápido: Tempo executando salvamento de estados X Granulosidade.

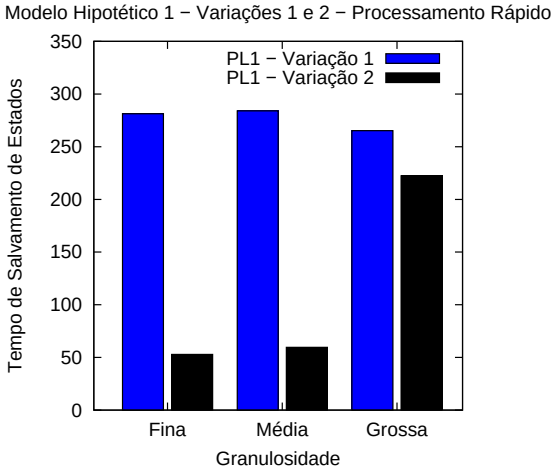


Figura 5.22: Modelo hipotético 1, processamento rápido: Tempo executando salvamento de estados X Granulosidade.

A Figura 5.23 ilustra o tempo em que o modelo com processamento lento e granulosidades fina, média e grossa, utilizando variações 1 e 2 para o PL_1. Com o processamento lento, o tempo em que a simulação distribuída executa o salvamento de estados na variação 1 tende a crescer com o aumento da granulosidade, pois os parâmetros utilizados na execução desse modelo influenciam os resultados. Enquanto que na variação 2 essa métrica tende a diminuir com o aumento da granulosidade. Além disso, observa-se que PL_0, na variação 1, executa o salvamento de estados em um tempo maior comparado à variação 2, isto é, PL_0 praticamente não executou salvamento de estados na variação 2. Esses resultados também são influenciados

Tabela 5.13: Tempo executando salvamentos de estados: variações 1 e 2, processamento rápido.

Granulosidade	**Variação 1**	**Variação 2**
	PL_0	PL_0
Fina	122,5228	1,7546
Média	122,9639	1,0833
Grossa	104,1671	0,4029
	PL_1	PL_1
Fina	281,2904	52,8451
Média	283,9855	59,5440
Grossa	265,2850	222,4212

pelos diferentes parâmetros utilizados em ambas as variações.

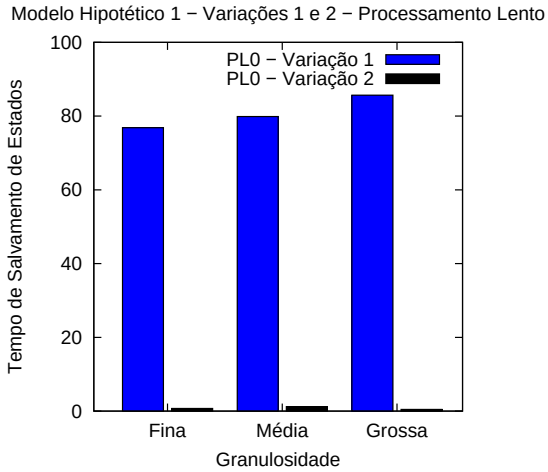


Figura 5.23: Modelo hipotético 1, processamento lento: Tempo executando salvamento de estados X Granulosidade.

A Figura 5.24 ilustra os resultados obtidos do tempo de salvamento de estados com esse modelo, utilizando um processamento lento, granulosidades fina, média e grossa, com variações 1 e 2 para PL_1. Em ambas as variações observa-se que essa métrica tende a diminuir com o aumento da granulosidade e no processamento lento, PL_1 gastou mais tempo salvando estados em relação ao processamento rápido, com granulosidades fina e média, mas passou menos tempo executando salvamento de estados com a granulosidade grossa.

O motivo para que PL_1 passe mais tempo executando salvamento de estados que PL_0 está no fato que os eventos salvos com execuções incorretas também são considerados em PL_1. Além disso, PL_1, passa a executar menos salvamentos de estados, o comprimento de *rollbacks* primários diminui e o número de *rollbacks* primários aumenta.

A Tabela 5.14 apresenta os valores obtidos com a execução do modelo hipotético 1 com as duas variações, para um processamento rápido, representando as Figuras 5.23 e 5.24.

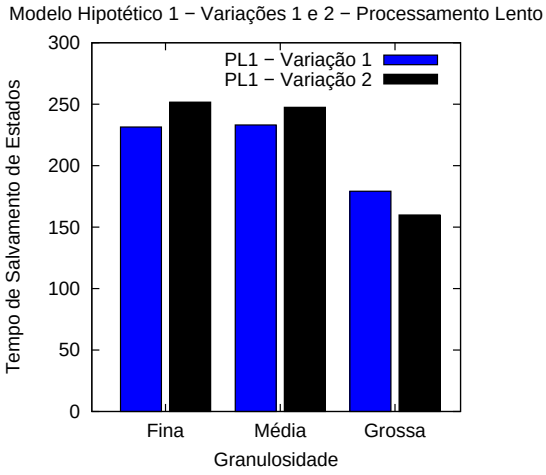


Figura 5.24: Modelo hipotético 1, processamento lento: Tempo executando salvamento de estados X Granulosidade.

Tabela 5.14: Tempo executando salvamentos de estados: variações 1 e 2, processamento lento.

Granulosidade	**Variação 1**	**Variação 2**
	PL_0	PL_0
Fina	76,8423	0,7098
Média	79,8687	1,1992
Grossa	85,6717	0,4670
	PL_1	PL_1
Fina	231,4515	251,6921
Média	233,0839	247,4959
Grossa	179,2056	159,8198

Visão Geral do Tempo Gasto em Salvamento de Estados e Execução de *Rollbacks* Primários

A Figura 5.25 ilustra uma visão geral do tempo gasto com salvamento de estados e *rollbacks* primários com granulosidades fina, média e grossa para a variação 1. Os resultados apresentados executando o modelo, com uma granulosidade fina e processamentos rápido e lento indicam uma redução no tempo de salvamento de estados e no tempo de execução de *rollbacks* primários quando um processamento lento é utilizado. Além disso, esses resultados mostram que PL_1 salva estados durante um maior tempo em relação a PL_0, pois esse processo lógico também salvou estados incorretos.

Para a granulosidade média, observa-se que esses resultados são próximos quando comparados com os obtidos com a granulosidade fina. Os resultados mostram um PL_0 efetuando menos salvamentos de estados que PL_1, pois PL_1 acumula também o tempo gasto em salvamento de estados referentes a eventos que são desfeitos na execução de *rollbacks* primários.

Também mostram que PL_1 executa *rollbacks* primários em um tempo maior no processamento rápido.

Com granulosidade grossa, os resultados indicam uma redução, tanto no tempo de salvamento de estados quanto no tempo de execução de *rollbacks* primários no processamento lento, principalmente em PL_1. PL_1 continua a efetuar salvamento de estados em um tempo maior que PL_0, também por que PL_1 acumula o tempo de salvamento de estados feito inutilmente.

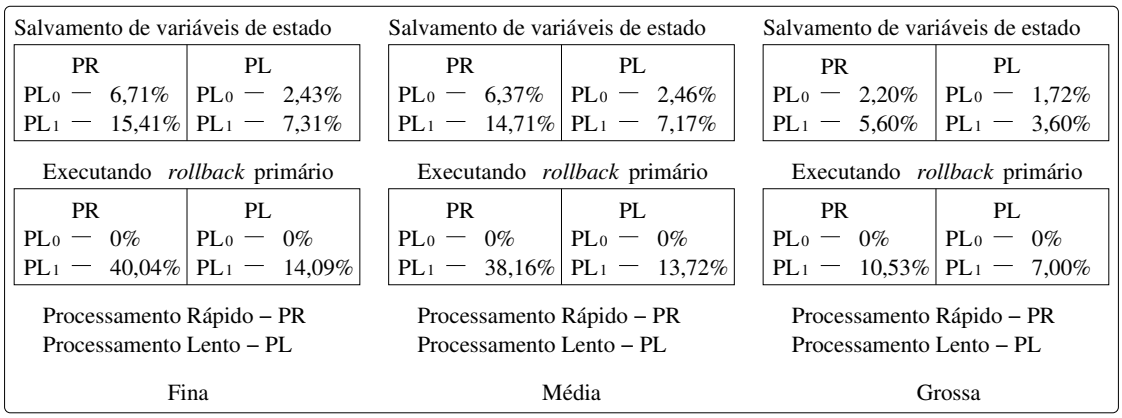


Figura 5.25: Modelo hipotético 1, variação 1: Tempo do ciclo de simulação.

Foram realizadas as coletas dos tempos de cada passo na execução da simulação distribuída, com processamentos rápido e lento, na variação 2 e com granulosidades fina, média e grossa para entender onde está sendo gasto mais tempo na execução do programa. A Figura 5.26 ilustra os dados obtidos com a execução do modelo, com uma granulosidade fina e processamentos rápido e lento. Observa-se que PL_0 passou pouco tempo da execução da simulação distribuída efetuando salvamento de estados, tanto em um processamento rápido quanto em um lento, enquanto que PL_1 efetuou salvamento de estados em um tempo maior, pois os estados salvos decorridos de execuções incorretas também são considerados. PL_1 também efetuou *rollbacks* primários, em um processamento lento, em um tempo menor.

Com a granulosidade média, esses resultados indicam que PL_0 efetuou salvamento de estados, mas com um tempo menor em relação a PL_1, novamente por PL_1 não descartar estados salvos incorretamente. PL_1 executou *rollbacks* primários praticamente em um tempo igual utilizando uma granulosidade fina.

Na granulosidade grossa, observa-se uma redução no tempo gasto com salvamento de estados em ambos os processos lógicos e uma redução na execução de *rollbacks* primários em PL_1. Além disso, como PL_1 considera os estados salvos incorretamente, o tempo de salvamento de estados nesse processo lógico é maior que em PL_0.

5.4.2 Modelo Hipotético 2

Para o modelo da Figura 5.5, utilizando processamentos rápido e lento e granulosidades fina, média e grossa, é realizada uma análise das métricas descritas na Seção 5.1. Esse modelo

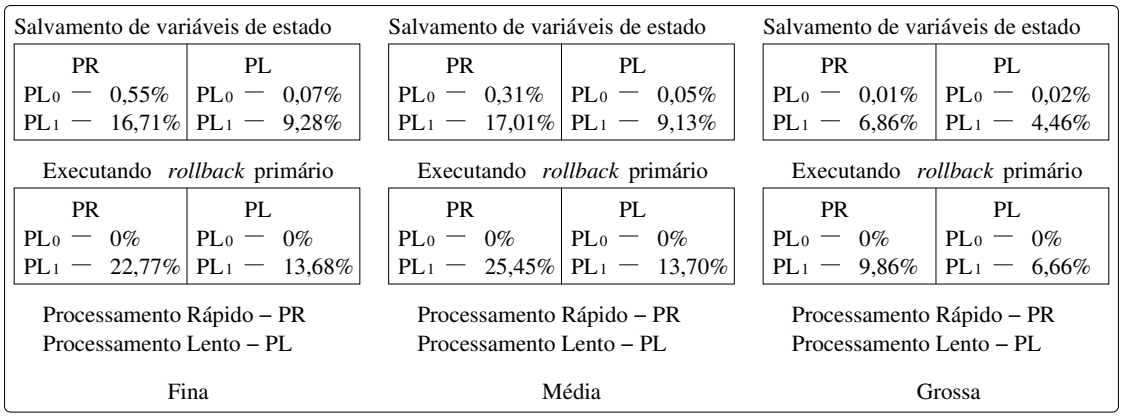


Figura 5.26: Modelo hipotético 1, variação 2: Tempo do ciclo de simulação.

possui dois recursos, cada um com uma fila e um servidor. O particionamento foi efetuado com dois processos lógicos, PL_0 e PL_1.

Comprimento Médio de *Rollbacks* Primários

A Figura 5.27 ilustra o comprimento médio de *rollback* primário durante a execução da simulação distribuída para esse modelo, com processamentos rápido e lento e granulosidades fina, média e grossa. Os resultados obtidos indicam que o comprimento médio de *rollback* primário tende a diminuir com o aumento da granulosidade, em ambos os processamentos. Além disso, a execução do modelo com processamento lento tem como resultados um comprimento médio de *rollback* primário menor em relação ao processamento rápido, para ambas as granulosidades. Como no modelo anterior, essas situações ocorrem por que há um equilíbrio entre comunicação e processamento de eventos, aumentando o número de *rollbacks* primários e diminuindo o número médio de eventos que sofrem *rollbacks*.

A Tabela 5.15 apresenta os valores obtidos para o comprimento médio de *rollbacks* primários com a execução do modelo hipotético 2, com processamentos rápido e lento.

Tabela 5.15: Comprimento médio de *rollbacks* primários.

Granulosidade	**Processamento Rápido**	**Processamento Lento**
Fina	231,1655	118,7047
Média	175,3601	116,5349
Grossa	104,9291	71,5613

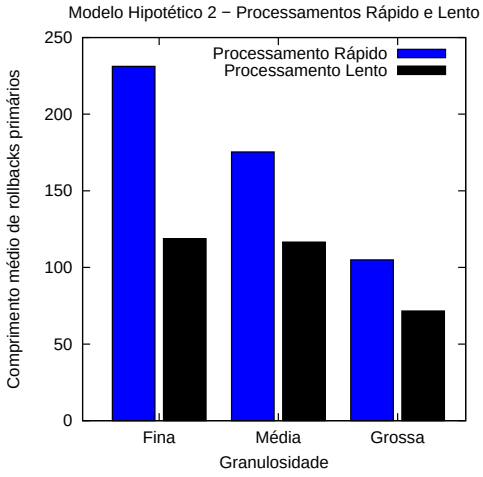


Figura 5.27: Modelo hipotético 2, processamento rápido: Comprimento médio de *rollbacks* primários X Granulosidade.

Tempo Executando *Rollback* Primário

A Figura 5.28 ilustra o tempo em que a simulação distribuída executa *rollbacks* primários no modelo, com processamentos rápido e lento e granulosidades fina, média e grossa. Para o processamento rápido, os resultados mostram um tempo maior executando *rollbacks* primários para uma granulosidade mais grossa, indicando que essa métrica tende a crescer com o aumento da granulosidade, pois o aumento na granulosidade não permite que a simulação distribuída alcance um equilíbrio entre comunicação e processamento de eventos.

No processamento lento, os resultados obtidos indicam que, nas granulosidades fina e média, o tempo em que a simulaçãao distribuída tende a aumentar, mas com a granulosidade grossa, o tempo em que a simulação distribuída executa *rollbacks* primários diminui, pois a execução atinge um equilíbrio entre comunicação e processamento de eventos, limitando o otimismo durante a execução da simulação distribuída. Para as granulosidades fina e média, executando em um processamento lento, essa métrica é maior em relação ao processamento rápido, mas para a granulosidade grossa, o valor dessa métrica diminui. O aumento no tempo de execução de *rollbacks* primários com processamento rápido e granulosidade grossa ocorre devido ao grande aumento no número de *rollbacks* primários executados, mesmo com a diminuição no comprimento médio de *rollbacks* primários.

A Tabela 5.16 apresenta os valores obtidos do tempo de execução de *rollbacks* primários, com a execução do modelo hipotético 2, com processamentos rápido e lento.

Tempo Executando Salvamento de Estados

A Figura 5.29 ilustra o tempo em que um programa de simulação distribuída executa salvamento de estados no modelo com processamentos rápido e lento, em PL_0 e granulosidades fina, média e grossa. Percebe-se que o tempo de salvamento de estados cresce entre as granu-

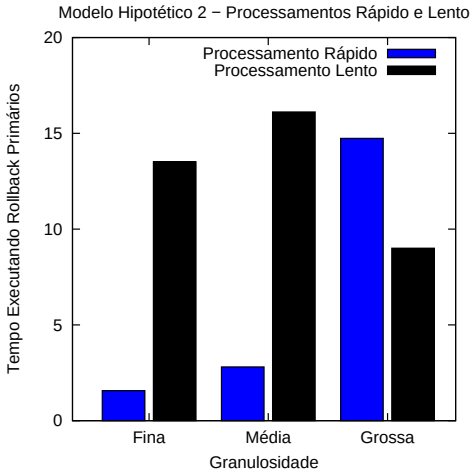


Figura 5.28: Modelo hipotético 2, processamento rápido: Tempo executando *rollback* primário X Granulosidade.

Tabela 5.16: Tempo de execução de *rollbacks* primários.

Granulosidade	**Processamento Rápido**	**Processamento Lento**
Fina	1,5669	13,5137
Média	2,8041	16,1135
Grossa	14,7385	9,0069

losidades fina e média em um processamento rápido e diminui com uma granulosidade grossa pois as características dos parâmetros de entrada do modelo e a própria aplicação não permitem que a simulação distribuída alcance um equilíbrio entre comunicação e processamento de eventos nesse modelo. No processamento lento, o tempo de salvamento de estados tende a diminuir entre as granulosidades fina e mdia, mas volta a crescer com uma granulosidade grossa.

A Tabela 5.17 apresenta os valores obtidos do tempo de salvamento de estados em PL_0, com a execução do modelo hipotético 2 e processamentos rápido e lento.

Tabela 5.17: Tempo de salvamento de estados em PL_0.

Granulosidade	**Processamento Rápido**	**Processamento Lento**
Fina	0,2669	0,1073
Média	0,6092	0,0671
Grossa	0,0757	0,0946

O PL_0 gasta menos tempo salvando estados em relação ao PL_1, pois PL_1 considera os

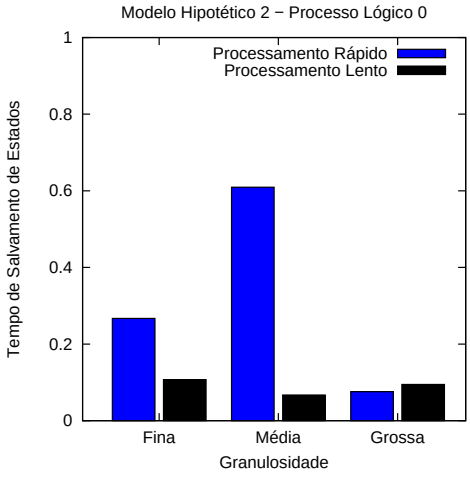


Figura 5.29: Modelo hipotético 2, processamento rápido: Tempo executando salvamento de estados X Granulosidade.

estados salvos incorretamente. Com processamentos rápido e lento e granulosidades fina, média e grossa, PL_1 tem um grande aumento no tempo de salvamento de estados, principalmente nas granulosidades fina e média, como ilustrado na Figura 5.30. No processamento rápido, o tempo de salvamento de estados tende a diminuir em PL_1 conforme a granulosidade aumenta, enquanto que no processamento lento, o tempo de salvamento de estados tende a crescer entre as granulosidades fina e média, mas depois volta a diminuir com a granulosidade grossa, pois novamente a execução da simulação atinge um equilíbrio entre comunicação e processamento.

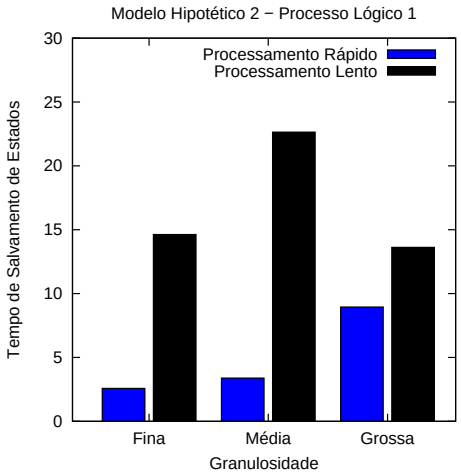


Figura 5.30: Modelo hipotético 2, processamento lento: Tempo executando salvamento de estados X Granulosidade.

A Tabela 5.18 apresenta os valores obtidos do tempo de salvamento de estados em PL_1,

com a execução do modelo hipotético 2 e processamentos rápido e lento.

Tabela 5.18: Tempo de salvamento de estados em PL_1.

Granulosidade	**Processamento Rápido**	**Processamento Lento**
Fina	2,5753	14,6237
Média	3,3823	22,6398
Grossa	8,9431	13,6204

Visão Geral do Tempo Gasto em Salvamento de Estados e Execução de *Rollbacks* Primários

A Figura 5.31 ilustra uma visão geral das porcentagens de tempos gastos efetuando salvamento de estados e *rollbacks* primários com granulosidades fina, média e grossa. Os resultados obtidos com a execução do modelo, com uma granulosidade fina e processamentos rápido e lento, indicam que os processos lógicos passaram pequena parte do tempo de simulação distribuída efetuando salvamento de estados, PL_1 efetuou salvamentos de estados em um tempo maior em relação a PL_0. PL_1 também gasta tempo salvando os estados incorretos e não descarta esses tempos. Além disso, PL_1 também passou pouco tempo efetuando *rollbacks* primários, em relação ao tempo total de simulação distribuída.

Com granulosidade média, observa-se que PL_1 efetua salvamentos de estados em um tempo maior em relação a PL_0, também por que PL_1 não descarta os estados salvos incorretamente. Comparando com a granulosidade fina, PL_1 efetuou *rollbacks* primários em uma porcentagem de tempo maior.

Utilizando granulosidade grossa, percebe-se uma redução na porcentagem de tempo em que a simulação distribuída efetua salvamento de estados e *rollbacks* primários em relação às granulosidades fina e média, indicando que, para uma granulosidade mais grossa, essas porcentagens de tempos tendem a diminuir. Além disso, como PL_1 acumula o tempo de salvamento de estados, mesmo para os estados salvos incorretamente, o tempo de salvamento de estados em PL_1 mostra-se maior que em PL_0.

5.4.3 Modelo Hipotético 3

A análise das métricas descritas na Seção 5.1 é realizada para o modelo da Figura 5.7, utilizando processamentos rápido e lento e granulosidades fina, média e grossa. Esse modelo possui três recursos, com uma fila e um servidor. Cada recurso é inserido em um processo lógico, isto é, o particionamento do modelo foi feito em três processos lógicos, PL_0, PL_1 e PL_2.

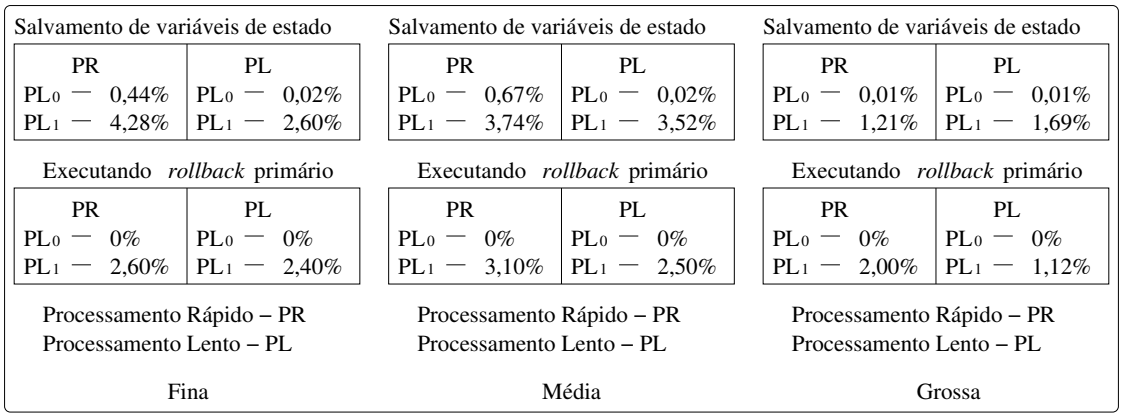


Figura 5.31: Modelo hipotético 2: Tempo do ciclo de simulação.

Comprimento Médio de *Rollbacks* Primários

Observa-se, com o comprimento médio de *rollback* primário desse modelo, que em PL_1 essa métrica tende a diminuir com o aumento da granulosidade, tanto utilizando processamento rápido, quanto lento. A Figura 5.32 ilustra essa situação, em PL_1, com processamentos rápido e lento e granulosidades fina, média e grossa. Também percebe-se que o comprimento médio de *rollbacks* primários é maior utilizando um processamento lento. Com a utilização de granulosidade grossa e processamento lento, a simulação distribuída atinge um equilíbrio entre comunicação e processamento de eventos, trazendo como resultado um número maior de *rollbacks* primários sendo executados, mas menos eventos sendo desfeitos, melhorando o *speedup* (como mostrado na Seção 5.3).

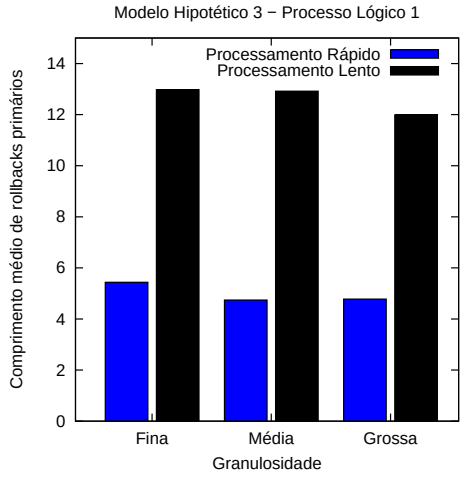


Figura 5.32: Modelo hipotético 3, processamento rápido: Comprimento médio de *rollbacks* primários X Granulosidade.

A Figura 5.33 ilustra uma situação, em PL_2, com processamentos rápido e lento e granulosidades fina, média e grossa para o comprimento médio de *rollbacks* primários. Os resultados obtidos também mostram que o comprimento médio de *rollbacks* primários também tende a diminuir com o aumento da granulosidade em PL_2. Além disso, observa-se que essa métrica possui valores maiores em PL_2, quando comparados aos valores obtidos em PL_1. Esse fato ocorre devido às características dos parâmetros utilizados e também pela função de geração de números pseudo-aleatórios utilizados.

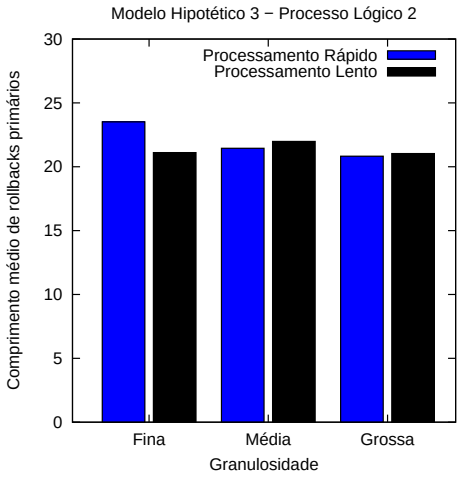


Figura 5.33: Modelo hipotético 3, processamento lento: Comprimento médio de *rollbacks* primários X Granulosidade.

A Tabela 5.19 apresenta os valores obtidos com o comprimento médio de *rollbacks* primários no modelo hipotético 3, em PL_1 e em PL_2 e processamentos rápido e lento.

Tabela 5.19: Comprimento médio de *rollbacks* primários em PL_1 e PL_2.

Granulosidade	**Processamento Rápido**	**Processamento Lento**
	PL_1	PL_1
Fina	5,4376	12,9797
Média	4,7427	12,9172
Grossa	4,7772	11,9889
	PL_2	PL_2
Fina	23,5159	21,1122
Média	21,4433	21,9810
Grossa	20,8255	21,0334

Tempo Executando *Rollback* Primário

A Figura 5.34 ilustra o tempo em que o modelo executou *rollbacks* primários, durante a simulação distribuída, com processamentos rápido e lento e granulosidades fina, média e grossa, em PL_1. Percebe-se que PL_1 executou *rollbacks* primários durante um tempo maior no processamento lento. Além disso, no processamento lento, com as granulosidades fina e média, o tempo executando *rollbacks* primários tende a crescer com o aumento da granulosidade, mas com a granulosidade grossa, esse valor diminui levemente. Enquanto que, com o processamento rápido, o tempo de execução de *rollbacks* primários tende a crescer com o aumento da granulosidade. Como o aumento da granulosidade retarda o processamento dos eventos, a simulação distribuída atinge um equilíbrio entre comunicação e processamento de eventos, limitando o otimismo durante a execução da simulação distribuída.

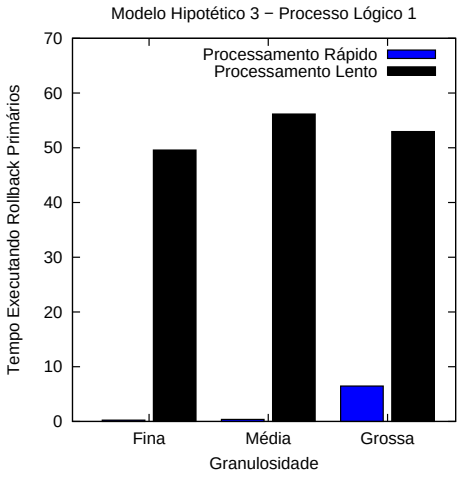


Figura 5.34: Modelo hipotético 3, processamento rápido: Tempo executando *rollback* primário X Granulosidade.

Os resultados obtidos com o tempo executando *rollbacks* primários em PL_2, com processamentos rápido e lento e granulosidades fina, média e grossa para o modelo são ilustrados na Figura 5.35 mostram uma situação parecida com PL_1, pois novamente os resultados demonstram um crescimento no tempo de execução de *rollbacks* primários com um processamento rápido e um aumento nessa métrica entre as granulosidades fina e média e depois uma diminuição na métrica com a granulosidade grossa.

Esses resultados são influenciados pelos parâmetros de entrada do modelo, pois PL_2 avança seu relógio da simulação distribuída de forma mais rápida que PL_1. O tempo executando *rollbacks* primários tende a crescer com o aumento da granulosidade. Nesse caso, o comprimento médio de *rollbacks* primários tende a diminuir com o aumento da granulosidade, mas o número de *rollbacks* primários executados tende a crescer muito, aumentando o tempo de execução de *rollbacks* primários. Além disso, os resultados apresentam diferenças nos dois processos lógicos, com o PL_1 utilizando um tempo menor para executar *rollbacks* primários em relação ao PL_2, novamente devido às características do modelo e de seus parâmetros de entrada.

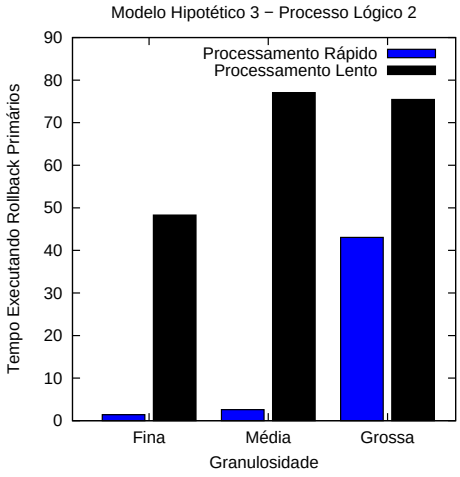


Figura 5.35: Modelo hipotético 3, processamento lento: Tempo executando *rollback* primário X Granulosidade.

A Tabela 5.20 apresenta os valores obtidos com o tempo de execução de *rollbacks* primários no modelo hipotético 3, em PL_1 e em PL_2, com processamentos rápido e lento.

Tabela 5.20: Tempo executando *rollbacks* primários em PL_1 e PL_2.

Granulosidade	**Processamento Rápido**	**Processamento Lento**
	PL_1	PL_1
Fina	0,2312	49,5897
Média	0,3801	56,1579
Grossa	6,4684	52,9530
	PL_2	PL_2
Fina	1,3972	48,2856
Média	2,5891	75,4613
Grossa	43,0448	77,0700

Tempo Executando Salvamento de Estados

O salvamento de estados é efetuado em todos os processos lógicos envolvidos na simulação distribuída. A Figura 5.36 ilustra o tempo em que esses processos lógicos executaram salvamento de estados durante a simulação distribuída para o modelo em PL_0, com processamentos rápido e lento e granulosidades fina, média e grossa. Observa-se que o tempo de salvamento de estados em um processamento rápido cresce entre as granulosidades fina e média, mas diminui com uma granulosidade mais grossa, pois a simulação distribuída atinge um equilíbrio entre comunicação e processamento. Enquanto que, com um processamento mais lento, o tempo de

salvamento de estados em PL_0 tende a crescer com o aumento da granulosidade, o que não acontece com os demais processos lógicos.

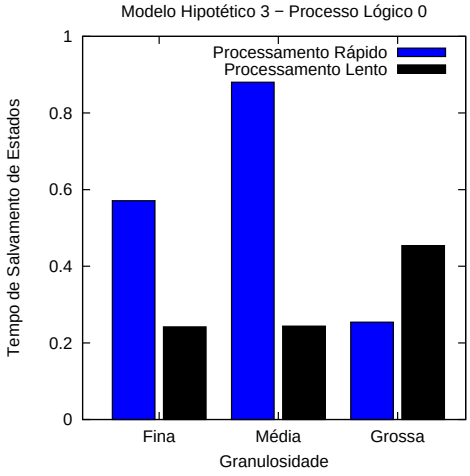


Figura 5.36: Modelo hipotético 3, processamento rápido: Tempo executando salvamento de estados X Granulosidade.

Em PL_1, o tempo de salvamento de estados tende a diminuir com o aumento da granulosidade para um processamento lento, enquanto que, para um processamento rápido, o tempo de salvamento de estados tende a crescer. Observa-se, também, que no PL_0, com granulosidades fina e média, o tempo de salvamento de estados foi menor no processamento lento, quando comparado com o processamento rápido. O mesmo não aconteceu nos demais processos lógicos e com a granulosidade grossa em PL_0, devido às características dos parâmetros de entrada. A Figura 5.37 ilustra esses resultados obtidos com o tempo de salvamento de estados em PL_1, com processamentos rápido e lento e granulosidades fina, média e grossa.

A Figura 5.38 apresenta os resultados obtidos com o salvamento de estados em PL_2, com processamentos rápido e lento e granulosidades fina, média e grossa. Para um processamento rápido, a simulação distribuída ainda não atingiu um equilíbrio, ocorrendo um crescimento no tempo de salvamento de estados quando a granulosidade aumenta. No processamento lento, para as granulosidades fina e média o tempo de salvamento de estados cresce, mas quando é utilizada granulosidade grossa, o tempo de salvamento de estados diminui, devido às características da aplicação. Essa execução atinge um equilíbrio entre comunicação e processamento, melhorando o *speedup*.

Esse modelo possui três recursos, divididos em três processos lógicos. O fato de que o comprimento médio de *rollbacks* primários é maior em PL_2 em relação a PL_1 explica por que esse processo lógico salva mais estados.

A Tabela 5.21 apresenta os valores obtidos com o tempo executando salvamento de estados no modelo hipotético 3, em PL_0, PL_1 e PL_2, com processamentos rápido e lento.

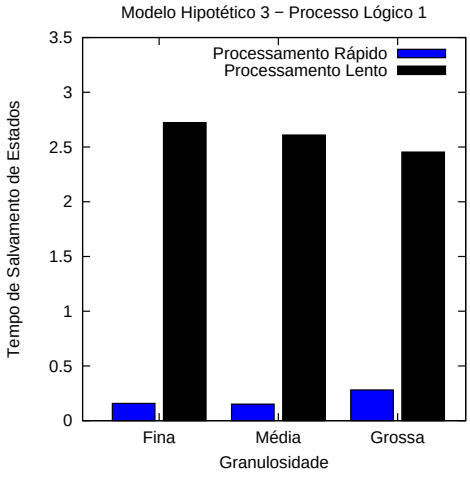


Figura 5.37: Modelo hipotético 3, processamento lento: Tempo executando salvamento de estados X Granulosidade.

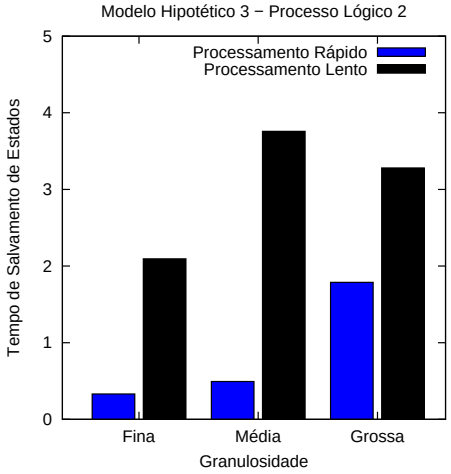


Figura 5.38: Modelo hipotético 3, processamento lento: Tempo executando salvamento de estados X Granulosidade.

Visão Geral do Tempo Gasto em Salvamento de Estados e Execução de *Rollbacks* Primários

A Figura 5.39 ilustra uma visão geral das porcentagens dos tempos gastos efetuando salvamento de estados e *rollbacks* primários para o modelo, com granulosidades fina, média e grossa. Os resultados obtidos com a execução do modelo utilizando granulosidade fina e processamentos rápido e lento indicam que os processos lógicos passaram pouco tempo na simulação distribuída efetuando salvamento de estados. Além disso, comparando com os outros modelos executados, os processos lógicos também ficaram uma menor porcentagem de

Tabela 5.21: Tempo executando salvamento de estados em PL_0, PL_1 e PL_2.

Granulosidade	**Processamento Rápido**	**Processamento Lento**
	PL_0	PL_0
Fina	0,5710	0,2418
Média	0,8803	0,2436
Grossa	0,2540	0,4538
	PL_1	PL_1
Fina	0,1575	2,7225
Média	0,1511	2,6100
Grossa	0,2810	2,4542
	PL_2	PL_2
Fina	0,3307	2,0939
Média	0,4928	3,7577
Grossa	1,7879	3,2803

tempo efetuando *rollbacks* primários.

Utilizando uma granulosidade média, observa-se, novamente, que ambos os processos lógicos ficaram menos de 1% (um por cento) do tempo de execução da simulação distribuída efetuando salvamentos de estados. O tempo gasto com execução de *rollbacks* primários também foi pequeno. Apenas no processamento lento os processos lógicos ficaram pouco mais de 1% (um por cento) do tempo de execuçã da simulação distribuída efetuando *rollbacks* primários e uma menor porcentagem em um processamento rápido.

Com granulosidade grossa, percebe-se uma pequena redução na porcentagem de tempo em que a simulação distribuída efetua salvamento de estados e *rollbacks* em relação à granulosidade média, indicando que esses tempos tendem a diminuir com o aumento da granulosidade, mas aumentam com a utilização do processamento lento.

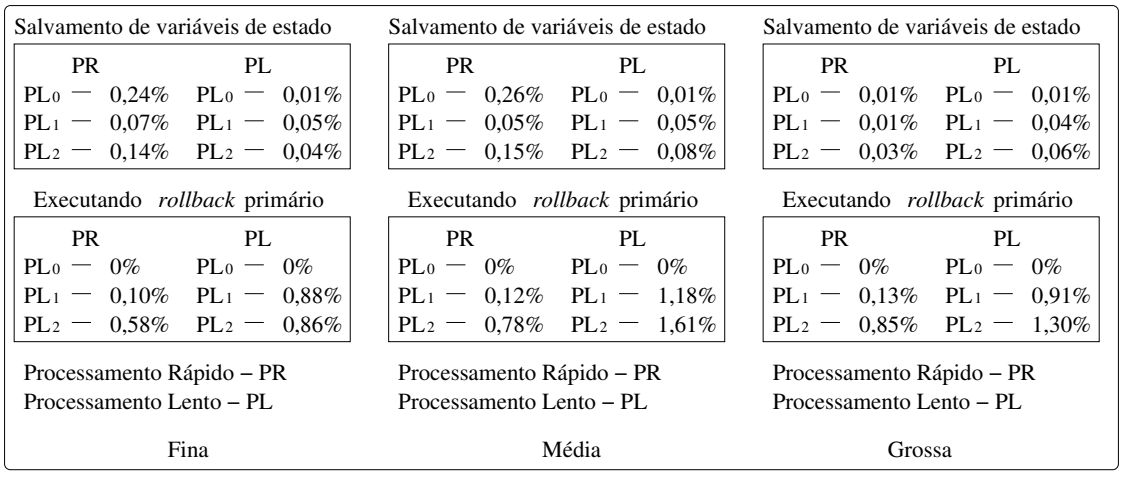


Figura 5.39: Modelo hipotético 3: Tempo do ciclo de simulação.

5.5 Considerações Finais

Este capítulo apresentou os resultados da análise de desempenho da simulação distribuída usando o ETW e também análise de diversos aspectos da simulação distribuída, tendo como estudos de caso modelos de redes de filas. As execuções dos modelos hipotéticos, usando a simulação distribuída, mostraram um baixo desempenho do ETW em relação à simulação seqüencial, principalmente pelo fato desses modelos serem simples, sendo necessário poucos processos lógicos para efetuar a simulação. Porém, mostraram a viabilidade do uso do ETW para efetuar simulação distribuída. Como o ETW é executado apenas com modelos hipotéticos muito simples, procurou-se mostrar as potencialidades do mesmo, isto é, o que se pode extrair da simulação distribuída e quais as métricas que podem ser utilizadas.

Nos modelos observados, o comprimento médio de *rollbacks* primários influencia no tempo de execução, pois para um processamento rápido, essa estatística mostra um número de eventos que sofrem *rollbacks* primários bem maior em relação ao processamento lento. Conforme a granulosidade torna-se mais grossa, o número de eventos que sofrem *rollbacks* primários também tende a diminuir. Foram observados também o comportamento do tempo gasto na execução de *rollbacks* primários e no salvamento de estados. Essas métricas são dependentes das características do modelo e dos parâmetros da aplicação, nos modelos de filas estudados.

Os resultados obtidos com as execuções dos modelos hipotéticos foram parecidos com os resultados da meta-simulação realizada por Spolon [Spo01]. Ou seja, isso mostra que o ETW está no caminho certo para ser usado na avaliação de sistemas através da solução de modelos de filas.

Além do uso do ETW para efetuar simulação distribuída otimista e de sua aplicação na análise da simulação distribuída, este capítulo apresenta como contribuição a identificação de uma possível métrica para o protocolo de trocas de Morselli [Mor00], que é o comprimento médio de *rollbacks* primários.

O capítulo 6 apresenta as conclusões gerais deste trabalho, as contribuições e sugestões para continuidade.

Capítulo 6

Conclusões, Contribuições e Propostas para Trabalhos Futuros

6.1 Conclusões Gerais

Esta dissertação mostrou a implementação de um núcleo para simulação distribuída otimista utilizando o protocolo *Time Warp*. Esse núcleo, denominado `ETW`, foi desenvolvido conforme proposta apresentada por Spolon [Spo01], na qual o *Time Warp* é visto como um conjunto de processos lógicos que efetuam operações básicas de execução e reexecução de eventos, além de uma série de *plugins* que, quando adicionados ao núcleo, permitem completar o funcionamento do *Time Warp* ou estender/alterar seu comportamento, para representar um protocolo otimista variante.

No decorrer do trabalho foi apresentada uma revisão sobre simulação seqüencial, seu mecanismo de operação e descrição da extensão funcional *SMPL*, que serviu de base para o desenvolvimento do engenho de simulação do `ETW`. O uso do *SMPL* como base facilita o trabalho do programador e/ou modelador que já utiliza essa extensão. Além disso, foi apresentada uma revisão sobre simulação distribuída detalhando o protocolo *Time Warp* básico, o qual também serviu de base para o desenvolvimento do `ETW`. Foram apresentados seus mecanismos de controle local e global e alguns algoritmos que alteram o seu comportamento.

A implementação do `ETW` mostrou a viabilidade de utilizar um ambiente de troca de mensagens para implementação da simulação distribuída otimista. Os estudos de desempenho do Capítulo 5, embora não tendo apresentado tempos de execução da simulação distribuída menores do que os da simulação seqüencial, indicam que o `ETW` pode ter um melhor desempenho se for utilizado em modelos mais complexos. Os modelos executados no ambiente de rede do laboratório do mestrado em Ciência da Computação da Universidade Federal de Mato Grosso do Sul não demonstraram um bom desempenho, pois eram simples e foram executados utilizando um processamento rápido e uma comunicação lenta. Estudos apresentados por Ulson [Uls99] mostraram melhores resultados em plataformas com processamento lento. Entretanto, serviram para mostrar o uso do `ETW` como protocolo de sincronização em simulação distribuída.

Neste trabalho, as maiores dificuldades encontradas ficaram concentradas no desenvolvi-

mento das rotinas relacionadas com o gerenciamento dos módulos, principalmente *threads*, no ambiente de passagem de mensagens *LAM-MPI*, implementação do mecanismo de *rollback* e implementação do *Copy Sate Saving*.

6.2 Contribuições

A contribuição fundamental dessa dissertação é a disponibilização do `ETW`, que poderá ser imediatamente incorporado ao ASDA. Com a inserção do `ETW` ao ASDA, é possível efetuar comparações entre o *Time Warp* e outros protocolos de simulação distribuída, como por exemplo o *CMB*, e comparações entre protocolos de simulação distribuída e programas de simulação seqüencial. A inserção de protocolos variantes do *Time Warp* básico, com o uso de *plugins*, permitirá efetuar comparações entre esses e o `ETW`.

Além disso, a revisão de simulação seqüencial e distribuída fornece um entendimento básico sobre ambas, que são importantes para a avaliação de desempenho. Os testes efetuados com modelos de filas hipotéticos permitiram a identificação de uma possível métrica para o protocolo de trocas de Morselli [Mor00]. A análise dos resultados obtidos mostrou quais métricas são potencialidades para esse mecanismo de trocas.

A estrutura do `ETW` permite que novos trabalhos possam implementar outros *plugins*, como por exemplo para salvamento de estados, cancelamento, cálculo de GVT e escalonamento de eventos. Alguns desses *plugins* foram implementados no `ETW`, como o *Copy State Saving*, o cancelamento agressivo, o cálculo do GVT de Samadi [Sam85] e o mecanismo de *fossil collection*.

6.3 Sugestões para Continuidade do Trabalho

Como sugestões para trabalhos futuros, os seguintes tópicos podem ser abordados:

- Integrar o `ETW` ao ASDA;
- Alterar o diagrama de classes do `ETW` de forma a permitir vários processos lógicos em um mesmo processador e, assim, estudar e implementar algoritmos de escalonamento de processos lógicos em processadores [Qua02];
- Estudar algoritmos adaptativos para execução do cálculo do GVT, verificando a influência desse na simulação distribuída e para execução do *fossil collection*;
- Inserir outros *plugins* do *Time Warp*, como por exemplo, o salvamento de estados incremental, cálculo do GVT de Mattern, algoritmos de escalonamento de eventos, entre outros;
- Remodelar o núcleo *Time Warp* transformando-o em um programa totalmente orientado a objetos, utilizando linguagem de programação Java (com *sockets*) para implementar a comunicação entre processos;

- Integrar o *batch means* para efetuar análise de saída na simulação distribuída;
- Efetuar testes para verificar a influência das características da plataforma na simulação distribuída otimista, como executado por Ulson [Uls99] com simulação conservativa;
- Identificar e estudar outras possíveis métricas, no *Time Warp*, que servirão de base para a troca de protocolos proposta por Morselli [Mor00];
- Estudar classes de modelos fechados utilizando a simulação distribuída, de forma a identificar quais métricas são potenciais para esses modelos;
- Estudar classes de aplicações que melhor se adaptam à simulação distribuída otimista;
- Estudar a influência dos parâmetros do modelo na simulação distribuída.

Referências Bibliográficas

[Ale01] Alexopoulos, C; Seila, A. F. Output Data Analysis for Simulations. *Proceedings of the 33rd Winter Simulation Conference*. vol.1 p.115-122, 2001.

[Aya92] Ayani, R; Rajaei, H. Parallel Simulation Using Conservative Time Windows. *Proceedings of the 24th Winter Simulation Conference*. p.709-717, 1992.

[Bag94] Bagrodia, R. L.; Jha, V.; Waldorf, J. The Maisie Environment for Parallel Simulation. *Proceedings of the 27th Annual Simulation Symposium*. p.4-12, 1994.

[Bag98a] Bagrodia, R. Language Support for Parallel Discrete Event Simulation. *IEEE Computer*. vol.31 p.77-85, 1998.

[Bag98b] Bagrodia, R.; Meyer, R.; Takai, M.; Chen, Y.; Zeng, X.; Martin, J.; Song, H. PARSEC: A Parallel Simulation Environment for Complex Systems. *IEEE Computer*. Vol.3, n.10, p.77-85, 1998.

[Ban01] Banks, J.; Carson, J. S.; Nicol, D. M.; Nelson B. L. *Discrete-Event System Simulation*. Third Edition. Prentice-Hall International Series In Industrial and Systems Engineering, 2001.

[Bru97] S. M. Bruschi. *Extensão do ASiA para Simulação de Arquitetura de Computadores*. São Carlos, Setembro de 1997. Dissertação (Mestrado) – Instituto de Ciências Matemáticas de São Carlos – Universidade de São Paulo - USP, 1997.

[Bru00] Bruschi, S. M. ASDA - An Automatic Distributed Simulation Environment. *Proceedings of 2000 Summer Computer Simulation Conference (SCSC'2000)*. Vancouver, Canada, 2000.

[Bru03] Bruschi S. M. *ASDA - Um Ambiente de Simulação Distribuído Automático*. São Carlos, Fevereiro de 2003. Tese (Doutorado) – Instituto de Ciências Matemáticas e de Computação de São Paulo – Universidade de São Paulo - USP.

[Bry77] Bryant, R. E. Simulation of Packet Communications Architeture Computer Systems. *MIT-LCSTR-188*. 1977.

[Car00] Carothers, C. D.; Fujimoto, R. Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms. *IEEE Transactions on Parallel and Distributed Systems*. vol.11, pag.299 - 317, March, 2000.

[Car02] Carson II, J. S. Model Verification and Validation. *Proceedings of the 34th Winter Simulation Conference*. vol.1 p.52-58, 2002.

[Cha79] Chandy, K. M.; Misra, J. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*. vol.SE-5 p.440-452, 1979.

[Cha85] Chandy, K. M.; Lamport, L. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems (TOCS)*. vol.3 p.63-75, 1985.

[Cra99] Crain, R. C.;Henriksen, J. O. Simulation Using GPSS/H. *Proceedings of the 31th Winter Simulation Conference - WSC'99*. vol.2, p.182-187, 1999.

[Dic91] Dickens, P.; Reynolds, P. A Performance Model for Parallel Simulation. *Proceedings of the 1991 Winter Simulation Conference*. p.618-626, 1991.

[Ewi99] Ewing, G. C.; Pawlikowski, K.; Mcnickle, D. Akaroa-2: Exploiting Network Computing by Distributing Stochastic Simulation. *Proceeding of European Simulation Multiconference (ESM99)*. Warson - Poland, p.175-181, 1999.

[Fel03] Felix, C. E. R.; Uyehara, D. U.; Takebe, K. M.; Nuha, V. M. *Cálculo do Global Virtual Time em Simulação Distribuída Otimista*. Campo Grande, 2003. Projeto Final – Departamento de Computação e Estatística – Universidade Federal de Mato Grosso do Sul - DCT/UFMS.

[Fer94] Ferscha, A.; Chiola, G. Self-Adaptative Logical Processes: the Probabilistic Distributed Simulation Protocol. *Proceedings of the 27th Annual Simulation Symposium*. p.78-88, IEEE Computer Society Press, April 1994.

[Fer95] Ferscha, A. Probabilistic Adaptative Direct Optimism Control in Time Warp. *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*. p.120-129, IEEE Computer Society Press, 1995.

[Fis92] Fishwick, P. A. SimPack: Getting Started With Simulation Programming in C and C++. *Proceedings of the 24th Winter Simulation Conference*. vol. p., 1992.

[Fis96] Fishwick, P.A. Extending Object-Oriented Design for Physical Modeling. *ACM Transactions on Modeling and Computer Simulation.* , Submitted July 1996.

[Fle95] Fleischmann, J; Wilsey, P. A. Comparative Analysis of Periodic State Saving Thechniques in Time Warp Simulators. *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*. p. 50-58, 1995.

[Fre99] Freitas Filho, P. J. *Introdução à Modelagem e Simulação de Sistemas*. Visual Books, 1999.

[Fuj90] Fujimoto, R. M. Parallel Discrete-Event Simulation. *Communications of the ACM*. vol.33 p.33-53, 1990.

[Fuj00] Fujimoto, R. M. *Parallel and Distributed Simulation Systems*. Wiley-Interscience. Series Editor, 2000.

[Gho96] Ghosh, S. On the Proof of Correctness of "Yet Another Asynchronous Distributed Discrete Event Simulation Algorithm (YADDES)". *IEEE Transactions on Systems, Man, and Cibernetics - Part A: Systems and Humans*. vol.26, n.1, p.68-80, 1996.

[Gob91] Goble, J. Introduction to SIMFACTORY II.5 *Proceedings of the 23th Winter Simulation Conference*. p.77-80, 1991.

[Gom95] Gomes, F.; Cleary, J.; Covington, A.; Franks S.; Unger, B.; Ziao, Z. SimKit: A High Performance Logical Process Simulation Class Library in C++. *Proceedings of the 27th Winter Simulation Conference*. vol.1 p.706-713, 1995.

[Gom96] Gomes, A. F. B.*Optimizing Incremental State Saving and Restoration*. Tese (Doutorado). Departament of Computer Science – The University of Calgary. April, 1996.

[Har03] Harrel, C. R.; Price, R. N. Simulation Modeling Using ProModel Technology. *Proceedings of the 35th Winter Simulation Conference*. vol.1 p.175-181, 2003.

[Hla02] Hlavicka, J.; Racek, S. C-Sim - The C Language Enhancement For Discrete-Time Simulation. *Proceedings of the International Conference on Dependable Systems and Networks*. p.539, 2002.

[Hu01] Hu, A.; San, Y.; Wang, Z. Verifying and Validating: A Simulation Model. *Proceedings of the 33rd Winter Simulation Conference*. vol.1, p.595-599, 2001.

[Hus04] Hussain, A.; Kapoor, A.; Heidemann, J. The Effect of Detail on Ethernet Simulation. *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, 2004.

[Ing02] Ingalls, R. G. Introduction to Simulation. *Proceedings of the 34th Winter Simulation Conference*. vol.1 p.7-16, 2002.

[Isk04] Iskra, K. A.; van Albada, G. D.; Sloot, P. M. A. Towards Grid-aware Time Warp. *Proceedings of the 18th Workshop on Parallel and Distributed Simulation (PADS'04)*.

[Jef85] Jefferson, D. R. Virtual time. *ACM Transaction on Programming Languages and Systems*. vol.7 p.404-425, 1985.

[Kal97] Kalantery, N. Tentative Time Warp. *In: Proceedings of the 3th International Euro-Par Conference. In Parallel and Distributed Algorithms, Lecture Notes in Computer Science*. vol. 1300, p. 458-467, 1997.

[Kal04] Kalantery, N. Time Warp - Connection Oriented. *Proceedings of the 18th Workshop on Parallel and Distributed Simulation (PADS'04).*

[Kaw99] Kawabata, C. L. O. *Extensão do Ambiente de Simulação Automático (ASiA) Para Simulação de Redes de Computadores.* São Carlos, Novembro de 1999. Dissertação (Mestrado). Instituto de Ciências Matemáticas e de Computação de São Carlos – Universidade de São Paulo - USP, 1999.

[Kob78] Kobayashi, H. *Modelling and Analysis. An Introduction to System Performance Evaluation.* Addison-Wesley Publishing Company, 1978.

[Kre97] Kreutzer, W.; Hopkins, J.; van Mierlo, M. SimJAVA - A Framework for Modeling Queueing Networks in Java. *Proceedings of the 29th Winter Simulation Conference.* vol. p.483-488, 1997.

[Lam04] The LAM/MPI Team. LAM/MPI Users Guide, Version 7.0.4. *Open Systems Lab.* http://www.lam-mpi.org, January, 2004.

[Li04] Li, L; Tropper, C. Event Reconstruction in Time Warp. *Proceedings of the 18th Workshop on Parallel and Distributed Simulation (PADS'04).*

[Lin91] Lin, Y.-B.; Lazowska, E. D. A Study of Time Warp Rollback Mechanisms. ACM Transactions on Modeling and Computer Simulations. vol. 1, n. 1, p.51-72, January 1991.

[Lin93] Lin, Y.-B.; Preiss, B. R.; Loucks, W. M.; Lasowska, E. D. Selecting the Checkpoint Interval In Time Warp Simulation. *Proceedings of the 7th Workshop on Parallel and Distributed Simulation.* p. 127-134, 1993.

[Lit93] Little, M. C.; McCue, D. L. Construction and Use of a Simulation Package in C++. *Computing Science Technical Report, University of Newcastle – http://cxxsim.ncl.ac.uk/.* n.437, 1993.

[Low99] Low, Y.; Lim, C.; Cai, W.; Huang, S.; Hsu, W.; Jain, S.; Turner, S. J. Survey of Languages and Runtime Libraries for Parallel Discrete-Event Simulation. *Simulation.* Vol.72 p.170-186, 1999.

[Lyn96] Lynch, N. A. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[Mac87] MacDougall, M. H. *Simulation Computer Systems - Tecniques and Tools.* The MIT Press, 1987.

[Mat93] Mattern, F. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. *Journal of Parallel and Distributed Computing.* vol.18 p., 1993.

[Mis86] Misra, J. Distributed Discrete-Event Simulation. *ACM Computing Surveys.* vol.65 p.18-39, 1986.

[Mor00] Morselli Jr., J. C. de M. *Um Mecanismo Para Troca de Protocolos de Sincronização de Simulação Distribuída em Tempo de Execução*. Tese (Doutorado). Instituto de Física de São Carlos – Universidade de São Paulo - USP, 2000.

[Orl95] Orlandi, R. C. G. S. *Ferramentas para Análise de Desempenho de Sistemas Computacionais Distribuídos*. Dissertação (Mestrado). Instituto de Ciências Matemáticas e de Computação - ICMC, Universidade de São Paulo, São Carlos - USP. Abril, 1995.

[Oza01] Ozaki, A.; Furuichi, M.; Takahashi, K.; Matsukawa, H. Design and Implementation of Parallel and Distributed Wargame Simulation System. *IEICE/IEEE Joint Special Issue on Autonomous Decentralized Systems and Systems Assurance*. vol.E84-D, no.10, p.1376-1384, 2001.

[Pan97] Panda, D.K.; Basak, D.; Donglai Dai; Kesavan, R.; Sivaram, R.; Banikazemi, M.; Moorthy, V. Simulation Of Modern Parallel Systems: A CSIM-based Approach. *Proceedings of the 29th Winter Simulation Conference*.vol.2, p.1013-1020, 1997.

[Qua98] Quaglia, F. Event History Based Sparse State Saving in Time Warp. *Proceedings of the Twelfth Workshop on Parallel and Distributed Simulation*. p.72-79, 1998.

[Qua02] Quaglia, F.; Cortellessa, V. On the Processor Scheduling Problem in Time Warp Synchronization. *ACM Transactions on Modeling and Computer Simulation*. Vol.12, Number 3, pag.143-175, 2002.

[Raj93] Rajaei, H.; Ayani, R.; Thorelli, L. The Local Time Warp Approach to Parallel Simulations. *In: Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS'93)*. vol.23, p.119-126, 1993.

[Raj95] Rajan, R.; Wilsey, P. A.; Dynamically Switching Between Lazy and Agressive Cancellation in a Time Warp Parallel Simulator. Proceedings of the 28th Annual Simulation Symposium. p. 22-30, 1995.

[Rat02] Rathmell, J.; Sturrock, D. T. The Arena Product Family: Enterprise Modeling Solutions. *Proceedings of the 34th Winter Simulation Conference*. vol.1 p.165-172, 2002.

[Rei90a] Reiher, P.; Fujimoto, R.; Bellenot, S.; Jefferson, D.; Cancellation Strategies in Optimistic Execution Systems. SCS - The Society for Computer Simulation. vol.22, Number 1, p.112-121, 1990.

[Rei90b] Reiher, P. Parallel Simulation Using The Time Warp Operating System. *Proceedings of the 23th Winter Simulation Conference*. vol.1 p.38-45, 1990.

[Ric91] Rich, D. O.; Michelsen, R. E. An Assessment of the ModSim/TWOS Parallel Simulation Environment. *Proceedings of the 23th Winter Simulation Conference*. vol.1, p.509-518, 1991.

[Ril03] Riley, P. F.; Riley, G. F.; SPaDes - A Distributed Agent Simulation Environment With Software-In-The-Loop Execution. *Proceedings of the 35th Winter Simulation Conference*. vol. 2 p.817-825, 2003.

[Roh02] Rohrer, M. W.; McGregor, I. W. Simulating Reality Using AutoMod. *Proceedings of the 34th Winter Simulation Conference*. vool.1, p.173-181, 2002.

[Rön94] Rönngren, R; Ayani, R. Adaptative Checkpointing In Time Warp. *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*. p. 110-117, 1994.

[Rön96] Rönngren, R; Liljenstam, M.; Ayani, R.; Montagnat, J. A Comparative Study of State Saving Mechanisms for Time Warp Synchronized Parallel Discrete Event Simulation. *Proceedings of the 29th Annual Simulation Symposium*. p.5-14, 1996.

[Sad99] Sadowski, D.; Bapat, V. The Arena Product Family: Enterprise Modeling Solutions. *Proceedings of the 31st Winter Simulation Conference*. vol.1, p.159-166, 1999.

[Sam85] Samadi, B. *Distributed Simulation, Algorithms and Performance Analysis*. Los Angeles, 1985. Tese (Doutorado) – University of California.

[San89] Santana, M. J. *An Advanced Filestore Architecture for a Multiple-LAN Distributed Computing System*. Tese (Doutorado). University of Southampton, 1989.

[San94] Santana, R. H. C.; Santana, M. J.; Spolon, R. *Técnicas para Avaliação de Desempenho de Sistemas Computacionais*. Notas Didáticas – Instituto de Ciências Matemáticas e de Computação de São Carlos (ICMC) – Universidade de São Paulo - USP, Setembro de 1994.

[Sch02] Schriber, T. J.; Brunner, D. T. Inside Discrete-Event Simulation Software: How it Works and Why it Matters. *Proceedings of the 34th Winter Simulation Conference*. vol.1 p.97-107, 2002.

[Sha88] Sharma, R; Rose, L. I. Modular Design for Simulation. *Software, Practice and Experience*. vol.18(10), 1988.

[Sil00] Da Silva, A. R. F. *Modelos de Rede de Filas para Sistemas Computacionais Distribuídos Simulação X Métodos Analíticos*. São Carlos, 2000. Dissertação (Mestrado). Instituto de Ciências Matemáticas de São Carlos – Universidade de São Paulo - USP, Abril de 2000.

[Sko96] Sköld, S.; Rönngren, R. Event Sensitive State Saving In Time Warp Parallel Discrete Event Simulations. *Proceedings of the 28th Winter Simulation Conference*. vol.1 p.653-660, 1996.

[Sni96] Snir, M.; Otto, S.; Huss-Lederman, S.; Walker, D.; Dongarra, J. MPI: The Complete Reference. *The MIT Press*. 1996.

[Soa92] Soares, L. F. G. *Modelagem e Simulação Discreta de Sistemas*. Editora Campus - LTDA., 1992.

[Spo91] Spolon, R. *Extensão Funcional de Modula2 para Simulação de Sistemas Discretos*. Relatório de Iniciação Científica, ICMSC-USP, 1991.

[Spo94] Spolon, R. *Um Editor Gráfico para um Ambiente de Simulação Automático*. São Carlos, 1994. Dissertação (Mestrado) – IFSC – Universidade de São Paulo - USP.

[Spo01] Spolon, R. *Um Método para Avaliação de Desempenho de Protocolos de Sincronização Otimistas para Simulação Distribuída*. São Carlos, 2001. Tese (Doutorado) – IFSC – Universidade de São Paulo - USP.

[Sta01] Stanley, B. The AutoMod Product Suite Tutorial. *Proceedings of the 33rd Winter Simulation Conference*. vol.1 p.209-216, 2001.

[Ste91] Steinman, J. S. SPEEDES: Synchronous Parallel Environment for Emulation and Discrete Event Simulation. *Proceedings of SCS Multiconference on Advances on Parallel and Distributed Simulation*. p.95-103, 1991.

[Ste92] Steinman, J. S. SPEEDES: A Unified Approach to Parallel Simulation. *Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS'92)*. p.75-84, 1992.

[Tan94] Tanenbaum, A. S. *Redes de Computadores*. Segunda Edição. Editora Campus, 1994.

[Uls99] Ulson, R. S. *Simulação Distribuída em Plataformas de Portabilidade: Viabilidade de Uso e Comportamento do Protocolo CMB*. Tese (Doutorado). Instituto de Física de São Carlos – Universidade de São Paulo - USP, 1999.

[Wei02] Weicker, R. Benchmarking. *Lecture Notes in Computer Science*. vol.2459, p.179-207, 2002.

[Wil96] Wilsey, P. A.; Martin, D. E.; McBrayer, T. J. WARPED: A Time Warp Simulation Kernel for Analysis and Application Development. *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*. vol. p.383-386, 1996.

[Wil97] Wilsey, P. A. Feedback Control in Time Warp Synchronized Parallel Simulations. *Proceedings of the First International Workshop on Distributed Interactive Simulation and Real Time Applications*. vol. p.31-38, 1997.

[Wil03] Wilsey, P. A.; Martin, D. E.; Hoekstra, R. J.; Keiter, E. R.; Hutchinson, S. A. Redesigning the WARPED Simulation Kernel for Analysis and Application Development. *Proceedings of 36th Annual Simulation Symposium (ANSS'03)*. vol. p., 2003.

[Won96] Wonnacott, P.; Bruce, D. The APOSTLE Simulation Language: Granularity Control and Performance Data. *Proceedings of the Tenth Workshop on Parallel and Distributed Simulation*. p.114-123, 1996.

[Zen04] Zeng, Y.; Cai, W.; Turner, S. J. Batch Based Cancellation: A Rollback Optimal Cancellation Scheme in Time Warp Simulations. *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, 2004.

[Zho04] Zhou, S; Turner, S. J.; Cai, W.; Zhao, H.; Pang, X. A Utility Model for Timely State Update in Distributed Wargame Simulations. *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, 2004.

Apêndice A - Implementações dos Modelos Utilizados

A.1 Modelo Hipotético 1

O código do modelo ilustrado na Figura A.1 e implementado utilizando o `ETW` é apresentado a seguir:

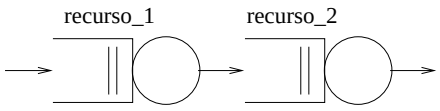


Figura A.1: Modelo Hipotético 1.

```
#include <stdio.h>

#include "timewarp.h"

#define Taxa 50

int main(int argc, char *argv[]) {
  double tc  = 3.0;
  double ts1 = 5.0;
  double ts2 = 9.0;
  double te  = 10000.0;
  int    customer = 1,
         token;
  int    event;

  int    prob;
  int    nts = 1;
```

```
int CPU;
int Disco;

init_simulation(argc, argv, "Modelo Hipotetico 1", 2,
                te, nts);

CPU = insert_resource("recurso_1", 1, ts1, 0);
Disco = insert_resource("recurso_2", 1, ts2, 1);

arrival(0, 1, customer, 0.0);

while(get_gvt() < te)
{
  event = get_event(token);

  switch( event )
  {
    case 1:
      schedule(0, CPU, Req, 2, customer, 0.0);
      stream(1);
      arrival(0, 1, ++customer, expntl(tc));
      break;

    case 2:
      if( request(CPU, event, token, 0, 0) )
      {
        stream(2);
        schedule(0, CPU, Rel, 3, token, expntl(ts1));
      }
      break;

    case 3:
      release(CPU, token, 0);
      remote_schedule(0, Disco, 4, token, 0.0, 1);
      break;

    case 4:
      if( request(Disco, event, token, 0, 1) )
      {
        stream(3);
        schedule(1, Disco, Rel, 5, token, expntl(ts2));
      }
      break;
```

```
      case 5:
    release(Disco, token, 1);
    break;
    }
  }
  end_simulation();
  return 0;
}
```

O código implementado em *SMPL* do modelo da Figura A.1 é apresentado a seguir:

```
#include "smpl.h"

int main() {
  real tc  = 3.0,
       ts1 = 5.0,
       ts2 = 9.0,
       te  = 10000.0;
  int  Cliente = 1,
       Evento,
       Token,
       Servidor1,
       Servidor2;

  int prob;

  // Inicia o sistema.
  smpl(0, "Modelo Hipotetico 1", te);

  // Cria e nomeia um descritor para cada recurso.
  Servidor1 = facility("recurso_1", 1);
  Servidor2 = facility("recurso_2", 1);

  // Escalona um evento.
  schedule(1, 0.0, Cliente);

  while( time() <= te )
  {
    // Remove o evento do inicio da fila e avan\c{c}a o relogio.
    cause(&Evento, &Token);

    if( time() > te )
      break;
```

```
    switch(Evento)
    {
      case 1:  /* Chegada de cliente */
        schedule(2, 0.0, Token);
        stream(1);
        schedule(1, expntl(tc), ++Cliente);
        break;

      case 2:  /* Requisicao do servidor */
        if(request(Servidor1, Token, 0) == 0)
        {
          stream(2);
          schedule(3, expntl(ts1), Token);
        }
        break;

      case 3:  /* Fim de servico */
        release(Servidor1, Token);
        schedule(4, 0.0, Token);
        break;

      case 4:
        if(request(Servidor2, Token, 0) == 0)
        {
          stream(3);
          schedule(5, expntl(ts2), Token);
        }
        break;

      case 5:
        release(Servidor2, Token);
        break;
    }
  }
  // Gera um relatorio de resultados.
  report();
  return 0;
}
```

A variação 2 do modelo hipotético 1 apresenta apenas alterações em alguns parâmetros de entrada. O parâmetro *tc* é utilizado com o valor igual a 5,0, o parâmetro *ts1* com o valor 3,0 e *ts2* com o valor igual a 12,0.

A.2 Modelo Hipotético 2

O código do modelo ilustrado na Figura A.2 e implementado utilizando o ETW é apresentado a seguir:

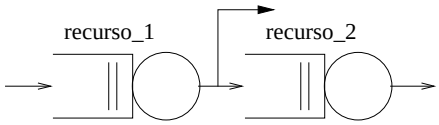


Figura A.2: Modelo Hipotético 2.

```
#include <stdio.h>

#include "timewarp.h"

#define Taxa 50

int main(int argc, char *argv[]) {
  double tc  = 2.5;
  double ts1 = 1.5;
  double ts2 = 4.5;
  double te  = 1000.0;
  int    customer = 1,
         token;
  int    event;

  time_t t;

  int    prob;
  int    nts = 1;

  int CPU;
  int Disco1;

  init_simulation(argc, argv, "Modelo Hipotetico 2", 2, te, nts);

  CPU = insert_resource("recurso_1", 1, ts1, 0);
  Disco1 = insert_resource("recurso_2", 1, ts2, 1);

  arrival(0, 1, customer, 0.0);

  while(get_gvt() < te)
```

```
{
  event = get_event(token);

  switch( event )
  {
    case 1:
      schedule(0, CPU, Req, 2, customer, 0.0);
      stream(1);
      arrival(0, 1, ++customer, expntl(tc));
      break;

    case 2:
      if( request(CPU, event, token, 0, 0) )
      {
        stream(2);
        schedule(0, CPU, Rel, 3, token, expntl(ts1));
      }
      break;

    case 3:
      stream(1);
      prob = random(1, 100);

      if( prob <= Taxa )
        release(CPU, token, 0);
      else
        release(CPU, token, 0);
        remote_schedule(0, Disco1, 4, token, 0.0, 1);
      break;

    case 4:
      if( request(Disco1, event, token, 0, 1) )
      {
        stream(3);
        schedule(1, Disco1, Rel, 5, token, expntl(ts2));
      }
      break;

    case 5:
      release(Disco1, token, 1);
      break;
  }
}
```

```
  end_simulation();
  return 0;
}
```

O código implementado em *SMPL* do modelo da Figura A.2 é apresentado a seguir:

```
#include "smpl.h"

#define Taxa 50

int main(int argc, char* argv[]) {
  real tc  = 2.5,
       ts1 = 1.5,
       ts2 = 4.5,
       te  = 1000.0;
  int  Cliente = 1,
       Evento,
       Token,
       Servidor1,
       Servidor2;
  int  prob;

  time_t t;
  // Inicia o sistema.
  smpl(0, "Modelo Hipotetico 2", te);

  // Cria e nomeia um descritor para cada recurso.
  Servidor1 = facility("recurso_1", 1);
  Servidor2 = facility("recurso_2", 1);

  // Escalona um evento.
  schedule(1, 0.0, Cliente);

  while( time() <= te )
  {
    // Remove o evento do inicio da fila e avan\c{c}a o relogio.
    cause(&Evento, &Token);

    if( time() > te )
      break;

    switch(Evento)
    {
      case 1:  /* Chegada de cliente */
```

```
        schedule(2, 0.0, Token);
        stream(1);
        schedule(1, expntl(tc), ++Cliente);
        break;

      case 2:  /* Requisicao do servidor */
       if(request(Servidor1, Token, 0) == 0)
       {
         stream(2);
         schedule(3, expntl(ts1), Token);
       }
       break;

      case 3:  /* Fim de servico */
        stream(1);
        prob = random(1, 100);

        if( prob <= Taxa )
          release(Servidor1, Token);
        else
        {
          release(Servidor1, Token);
          schedule(4, 0.0, Token);
        }
        break;

      case 4:
        if(request(Servidor2, Token, 0) == 0)
        {
          stream(3);
          schedule(5, expntl(ts2), Token);
        }
        break;

      case 5:
        release(Servidor2, Token);
        break;
    }
  }
  // Gera um relatorio de resultados.
  report();
  return 0;
}
```

A.3 Modelo Hipotético 3

O código do modelo ilustrado na Figura A.3 e implementado utilizando o ETW é apresentado a seguir:

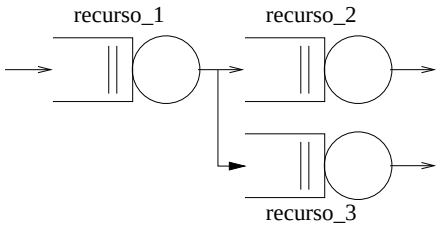


Figura A.3: Modelo Hipotético 3.

```
#include <stdio.h>

#include "timewarp.h"

#define Taxa 50

int main(int argc, char *argv[]) {
  double tc  = 5.0;
  double ts1 = 3.0;
  double ts2 = 9.0;
  double ts3 = 10.0;
  double te  = 10000.0;
  int    customer = 1,
         token;
  int    event;

  time_t t;

  int    prob;
  int    nts = 1;

  int CPU;
  int Disco1;
  int Disco2;

  init_simulation(argc, argv, "Modelo Hipotetico 3", 2,
                  te, nts);
```

```
CPU = insert_resource("recurso_1", 1, ts1, 0);
Disco1 = insert_resource("recurso_2", 1, ts2, 1);
Disco2 = insert_resource("recurso_3", 1, ts3, 1);

arrival(0, 1, customer, 0.0);

while(get_gvt() < te)
{
  event = get_event(token);

  switch( event )
  {
    case 1:
      schedule(0, CPU, Req, 2, customer, 0.0);
      //stream(2);
      arrival(0, 1, ++customer, tc);
      break;

    case 2:
      if( request(CPU, event, token, 0, 0) )
      {
        //stream(2);
        schedule(0, CPU, Rel, 3, token, ts1);
      }
      break;

    case 3:
      release(CPU, token, 0);
      stream(1);
      prob = random(1, 100);

      if( prob <= Taxa )
        remote_schedule(0, Disco1, 4, token, 0.0, 1);
      else
        remote_schedule(0, Disco2, 5, token, 0.0, 1);
      break;

    case 4:
      if( request(Disco1, event, token, 0, 1) )
      {
        //stream(2);
        schedule(1, Disco1, Rel, 6, token, ts2);
      }
```

```
        break;

      case 5:
        if( request(Disco2, event, token, 0, 1) )
        {
          //stream(2);
          schedule(1, Disco2, Rel, 7, token, ts3);
        }
        break;

      case 6:
        release(Disco1, token, 1);
        break;

      case 7:
        release(Disco2, token, 1);
        break;
    }
  }
  end_simulation();
  return 0;
}
```

O código implementado em *SMPL* do modelo da Figura A.3 é apresentado a seguir:

```
#include "smpl.h"

#define Taxa 50

int main(int argc, char* argv[]) {
  real tc  = 5.0,
       ts1 = 3.0,
       ts2 = 9.0,
       ts3 = 10.0,
       te  = 10000.0;
  int  Cliente = 1,
       Evento,
       Token,
       Servidor1,
       Servidor2,
       Servidor3;

  int prob;
```

```
time_t t;
// Inicia o sistema.
smpl(0, "Modelo Hipotetico 3", te);

// Cria e nomeia um descritor para cada recurso.
Servidor1 = facility("recurso_1", 1);
Servidor2 = facility("recurso_2", 1);
Servidor3 = facility("recurso_3", 1);

// Escalona um evento.
schedule(1, 0.0, Cliente);

while( time() <= te )
{
  // Remove o evento do inicio da fila e avan\c{c}a o relogio.
  cause(&Evento, &Token);

  if( time() > te )
    break;

  switch(Evento)
  {
    case 1:  /* Chegada de cliente */
      schedule(2, 0.0, Token);
      //stream(2);
      schedule(1, tc, ++Cliente);
      break;

    case 2:  /* Requisicao do servidor */
      if(request(Servidor1, Token, 0) == 0)
      {
        //stream(2);
        schedule(3, ts1, Token);
      }
      break;

    case 3:  /* Fim de servico */
      release(Servidor1, Token);
      stream(1);
      prob = random(1, 100);

      if( prob <= Taxa )
        schedule(4, 0.0, Token);
```

```
        else
          schedule(5, 0.0, Token);
        break;

      case 4:
        if(request(Servidor2, Token, 0) == 0)
        {
          //stream(2);
          schedule(6, ts2, Token);
        }
        break;

      case 5:
        if(request(Servidor3, Token, 0) == 0)
        {
          //stream(2);
          schedule(7, ts3, Token);
        }
        break;

      case 6:
        release(Servidor2, Token);
        break;

      case 7:
        release(Servidor3, Token);
        break;
    }
  }
  // Gera um relatorio de resultados.
  report();
  return 0;
}
```

Apêndice B - *LAM-MPI*

B.1 Introdução

O *MPI* (*Message Passing Interface*) é uma biblioteca de definições e funções que podem ser usadas em programas *C, C++* ou *Fortran*. O *LAM-MPI* é um *daemon* baseado na implementação do *MPI*. Inicialmente, o `lamboot` distribui os *daemons* do *LAM* baseado em uma lista de máquinas fornecidas pelo usuário. Esses *daemons* permanecem inativos na lista de máquinas remotas até que eles recebam uma mensagem para carregar o arquivo executável do *MPI* para iniciar a execução. Para desativar os *daemons*, basta executar o programa `lamhalt`. Como exemplo, considere o arquivo apresentado na Tabela B.1, chamado `lamhosts`, o qual contém uma lista de máquinas onde devem ser executados os *daemons* do *MPI* [Lam04].

Tabela B.1: Arquivo *lamhosts*.

work150.dct.ufms.br	rpss
work151.dct.ufms.br	rpss
work152.dct.ufms.br	rpss

Para iniciar o *daemons* nessas máquinas, basta executar o comando `lamboot -v lamhosts`. De maneira análoga, para desativar os *daemons*, o comando `lamhalt` ou o comando `wipe -v lamhosts` são executados.

B.2 Envio e Recebimento de Mensagens

Em ambientes de troca de mensagens, as tarefas coordenam suas atividades através do envio e do recebimento de mensagens. Uma mensagem é um vetor de elementos de um determinado tipo de dados. Uma mensagem é enviada para uma tarefa específica através da função *MPI_Send*. A função *MPI_Recv* recebe uma mensagem de uma determinada tarefa. As duas funções descritas são utilizadas para envio e recebimento bloqueantes. As funções *MPI_Isend* e *MPI_Irecv* são utilizadas para envio e recebimento não bloqueante de mensagens, respectivamente. O protótipo dessas funções é apresentado na Tabela B.2.

O conteúdo da mensagem está armazenado em *message*. Os parâmetros *count* e *datatype* permitem que o sistema determine quanto espaço é necessário para armazenar a mensagem

Tabela B.2: Funções para envio/recebimento de mensagens.

Descrição	**Protótipo**
Envio bloqueante	MPI_Send(message, count, datatype, destination, tag, communicator);
Recebimento bloqueante	MPI_Recv(message, count, datatype, source, tag, communicator, status);
Envio não bloqueante	MPI_Isend(message, count, datatype, destination, tag, communicator, request);
Recebimento não bloqueante	MPI_Irecv(message, count, datatype, source, tag, communicator, request);

e qual o tipo de dado sendo enviado/recebido. Os tipos de dados definidos pelos *LAM-MPI* estão representados na Tabela B.3. O parâmetro *destination* é o *rank* da tarefa que irá receber a mensagem, e o parâmetro *source* é o *rank* da tarefa que está enviando a mensagem. O parâmetro *tag* é um inteiro que pode ser utilizado para distinguir diferentes tipos de mensagens que uma tarefa pode enviar ou receber.

Tabela B.3: Tipos de dados pré-definidos pelo *LAM-MPI*.

LAM-MPI	**C/C++**
MPI_CHAR	char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	bype

B.3 Comunicação Coletiva

A comunicação coletiva envolve todos os processos de um comunicador. O *LAM-MPI* possui várias funções para efetuar esse tipo de comunicação, as quais são descritas a seguir:

- *Broadcast*: é uma operação de comunicação coletiva onde uma única tarefa envia o mesmo dado para todas as tarefas do comunicador. Esta operação é implementada no *LAM-MPI* pela função *MPI_Bcast*;
- *Reduce*: permite que todas as tarefas de um comunicador contribuam com um dado que é combinado usando uma operação binária. A função *MPI_Reduce* implementa essa operação;

- *Allreduce*: é usada da mesma maneira que a função *MPI_Reduce*. A única diferença é que o resultado da operação é armazenado em todos os processos, ao invés de ser armazenado apenas na tarefa raiz;

- *Gather*: é uma operação de comunicação coletiva onde a tarefa raiz recebe dados de todos os outros processos do comunicador. É implementada pelo *LAM-MPI* através da função *MPI_Gather*;

- *Allgather*: é usada da mesma maneira que *MPI_Gather*. A diferença é que o conteúdo de envio de cada tarefa é armazenado no conteúdo de recebimento de cada tarefa, ao invés de ser armazenado apenas no conteúdo de recebimento da tarefa raiz;

- *Scatter*: é uma operação de comunicação coletiva onde a tarefa raiz envia um conjunto de dados distintos para cada processo do comunicador. É implementada pelo através da função *MPI_scatter*;

- *Alltoall*: a operação troca total (*total exchange*) é uma operação de comunicação coletiva onde cada tarefa envia um conjunto de dados distintos para todas as outras tarefas do comunicador. Existem duas formas de realizar a troca total no *LAM-MPI*, *MPI_Alltoall* e *MPI_Alltoallv*. A primeira forma é utilizada quando desejamos enviar a mesma quantidade de dados para todos os processos;

- *Alltoallv*: é usada da mesma maneira que *MPI_Alltoall*. A diferença é que cada tarefa envia/recebe quantidades diferentes de dados para/de cada tarefa.

Apêndice C - Análises Estatísticas

Foram realizados testes de hipótese sobre os valores médios obtidos para o tempo de execução da simulação seqüencial e o tempo de execução da simulação distribuída.

Os testes realizados têm por objetivo verificar se as diferenças observadas para os tempos de execução dos modelos de simulação analisados são estatisticamente significativas. Nesse apêndice é exemplificado o teste realizado para a execução do modelo hipotético 1, variação 1, com processamento rápido e granulosidades fina, média e grossa.

H_0 é denominada a hipótese de nulidade, a ser testada, e H_1 a hipótese alternativa, considerada hipótese complementar a H_0. O teste leva à aceitação ou rejeição da hipótese H_0, o que corresponde à negação ou afirmação de H_1, respectivamente. Para manter a uniformidade, deve-se provar a hipótese H_0. A Tabela C.1 apresenta os possíveis testes de hipóteses que podem ser utilizados para dois valores médios amostrais quando σ^2 é desconhecido.

Tabela C.1: Roteiro para os testes de hipóteses.

Teste de Hipótese	**Hipótese Nula**	**Hipótese Alternativa**	**Região de Rejeição**
Bilateral	H_0: μ_1 - $\mu_2 = 0$	H_1: μ_1 - $\mu_2 \neq 0$	$\|t\| > t_{\alpha/2}$
Unilateral à direita	H_0: μ_1 - $\mu_2 = 0$	H_1: μ_1 - $\mu_2 > 0$	$t > t_\alpha$
Unilateral à esquerda	H_0: μ_1 - $\mu_2 = 0$	H_1: μ_1 - $\mu_2 < 0$	$t < -t_\alpha$

O teste de hipótese utilizado neste trabalho é o unilateral à direita, o qual tem-se como hipótese nula H_0: $TDistr$ - $TSeq = 0$, como hipótese alternativa H_1: $TDistr$ - $TSeq > 0$ e a região de rejeição utilizada é $t > t_\alpha$.

Para os testes de hipóteses realizados usou-se os resultados obtidos do modelo hipotético 1, com variação 1 e processamento rápido, com granulosidades. Fez-se uso do teste de hipóteses sobre a diferença de duas médias populacionais, utilizando a distribuição t de *Student*, para identificar qual o teste adequado para as médias amostras. Para efetuar esse teste, primeiro deve-se verificar o tamanho das amostras e a igualdade das duas variâncias através da distribuição F. Para o modelo em estudo, mostrou que ambas as variâncias são distintas. As Equações C.1 e C.2, são utilizadas para efetuar os testes de hipóteses utilizando a distribuição t de *Student* com variâncias distintas.

$$\delta^2_{TDistr} \neq \delta^2_{TSeq}, n_{TDistr} \neq n_{TSeq} \tag{C.1}$$

$$t = \frac{\overline{X_{Distr}} - \overline{X_{TSeq}}}{\sqrt{\frac{S^2_{TDistr}}{n_{TDistr}} + \frac{S^2_{TSeq}}{n_{TSeq}}}} \tag{C.2}$$

t_α é valor obtido a partir dos valores amostrais, $\overline{X_{TDistr}}$ e $\overline{X_{TSeq}}$ são as médias amostrais, S^2_{TDistr} e S^2_{TSeq} indicam as variâncias amostrais, n_{TDistr} e n_{TSeq} indicam os tamanhos das amostras, $TDistr$ indica amostras de execução da simulação distribuída e $TSeq$ indica amostras de execução da simulação seqüencial.

Para efetuar o teste de hipótese, deve-se fixar o valor de α, calcular t_α na tabela de *Student*, com graus de liberdade $v = n_{TDistr} + n_{TSeq} - 2$ e calcular t, de acordo com a Equação C.2. Se $t > t_\alpha$, rejeita-se H_0, caso contrário, aceita-se H_0.

O nível de significância α de um teste é a probabilidade da hipótese nula ser rejeitada, quando verdadeira. Nesse caso, é considerado um α igual a $0,05$. Conseqüentemente, tem-se $t_\alpha = 2{,}048$. A Tabela C.2 apresenta os resultados obtidos com o teste t de *Student* realizado. Em todos os casos, rejeita-se H_0, pois $t > t_\alpha$.

Tabela C.2: Análise Estatística - Modelo Hipotético 1 - Variação 1.

Granulosidade	**Hipótese**	t	**Resultado**
Fina	H_0: $TDistr - TSeq = 0$	$61,91$	$t > 2,048$ (rejeita-se H_0)
Média	H_0: $TDistr - TSeq = 0$	$51,20$	$t > 2,048$ (rejeita-se H_0)
Grossa	H_0: $TDistr - TSeq = 0$	$103,19$	$t > 2,048$ (rejeita-se H_0)

Bibliografia

Mendehall, W.; Sincich, T *Statistics for Engineering and the Sciences.* Dellen Publishing Company, 1992.

Apêndice D - Orientação de Projetos Relacionados ao ETW

Neste apêndice são apresentados os trabalhos realizados relacionados com o ETW.

- Félix, C. E. R.; Uehara, D. U.; Nuha, V. M. e Takebe, K. M. *Cálculo do Global Virtual Time em Simulação Distribuída Otimista*. Projeto Final – Bacharelado em Análise de Sistemas – Departamento de Computação e Estatística – Universidade Federal de Mato Grosso do Sul - DCT/UFMS. Campo Grande, 2003.
- Sotoma, E. L. K.; da Silva, E. L. G. e Fortini, E.M. *Algoritmos de Salvamento de Estados Incremental State Saving para o Time Warp*. Projeto Final – Bacharelado em Análise de Sistemas – Departamento de Computação e Estatística – Universidade Federal de Mato Grosso do Sul - DCT/UFMS. Campo Grande, 2004.