Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e Computação
Departamento de Engenharia de Computação e Automação Industrial
DCA/FEEC/UNICAMP

Simulador Distribuído para Auxílio ao Projeto de Sistemas Embutidos:
desenvolvimento e exemplos de aplicação

Marcelo Tilli
Orientadora: **Profa. Dra. Alice Maria B. H. Tokarnia**

Dissertação de Mestrado
Data da defesa: 21/12/2001

Comissão Julgadora:

Profa. Dra. Ana Cristina Viera de Melo IME/USP
Prof. Dr. Eleri Cardozo – FEEC/UNICAMP
Pofa. Dra. Wu Shin Ting – FEEC/UNICAMP
Campinas, SP – Brasil
Dezembro de 2001

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

Tilli, Marcelo
T465s Simulador distribuído para auxílio ao projeto de sistemas embutidos: desenvolvimento e exemplos de aplicação / Marcelo Tilli.--Campinas, SP: [s.n.], 2001.

Orientador: Alice Maria B.H. Tokarnia.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Elétrica e Computação.

1. Métodos de simulação. 2. Sistemas embutidos de computador. I. Tokarnia, Alice Maria B. H. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e Computação. III. Título.

Resumo

Tilli, M. Simulador Distribuído para Auxílio ao Projeto de Sistemas Embutidos: *desenvolvimento e exemplos de aplicação*. Campinas, DCA/ FEEC/UNICAMP, 2001. Dissertação de Mestrado

Esse trabalho apresenta um simulador orientado a eventos, distribuído e implementado seguindo a metodologia de simulação distribuída conservadora. Apresentamos o modelo adotado para a representação do sistema, o funcionamento interno do simulador e sugerimos algumas formas de explorar os resultados fornecidos pelo simulador durante o projeto de sistemas embutidos. O funcionamento desse simulador distribuído é ilustrado por dois exemplos. O primeiro consiste num sistema desenvolvido para cálculo, pelo Método de Barnes-Hut, da trajetória de corpos sob a ação de forças gravitacionais. Esse exemplo, devido a carga computacional, é adequado para avaliação de desempenho de sistemas computacionais distribuídos. O segundo exemplo, ilustra o uso do simulador durante o projeto de uma pequena rede ATM. Esse exemplo ilustra como o simulador é utilizado para auxiliar na tomada de decisões de projeto. Durante a execução dos exemplos, verificamos que o simulador distribuído desenvolvido chega a ser 3,28 vezes mais rápido do que o simulador seqüencial.

Palavras-Chaves: Simulação distribuída conservadora, Java, simulador orientado a eventos, sistemas embutidos.

Abstract

Tilli, M. Distributed Simulator to Assist in Embedded Systems Design. Campinas, DCA/ FEEC/UNICAMP, 2001. Master's Thesis

This work presents a distributed event-driven simulator, following the conservative distributed simulation methodology. We present the model adopted for embedded system representation and the internal functioning of the simulator. The use of the simulator is illustrated by two case studies. The first case study is a system developed to perform the calculation of body movement under the action of gravitational forces, using the Barnes-Hut method. Due to the great computational overhead caused by force calculation, this problem is suitable for benchmarking distributed systems. The second case study uses the simulator to assist in the design of a small ATM network. This case study shows how the simulator can be used to assist in the decision-making process during the design. The distributed simulator achieved a speedup of 3.28 as compared to an event-driven sequential simulator.

Keywords: Distributed conservative simulation, Java, event-driven simulator, embedded systems.

Para Noemi

"Alguns nascem para mudar o mundo - viver suas fantasias.
mas a maioria de nós apenas sonha sobre coisas que gostaríamos de ser"
Neil Peart

Agradecimentos

Em primeiro lugar, agradeço a minha mãe Noemi, pelo apoio, dedicação, carinho e incentivo. Ao meu pai Alfredo e ao meu irmão Fabiano.

Aos amigos Adelice, Kyn, Rodrigo e William, pela amizade e pela convivência.

Aos amigos Antônio, Ernesto e Maurício, pela amizade, criticas e sugestões que tanto me ajudaram na conclusão desse trabalho.

A minha orientadora Alice, pela oportunidade, confiança, paciência e orientação desse trabalho.

Agradeço a professora Ana Cristina Viera de Melo, ao professor Eleri Cardozo e a professora Wu Shin Ting, por aceitarem participar da banca julgadora.

Ao CNPq, pela bolsa de mestrado.

Sumário

Lista de Figuras

Lista de Tabelas

Abreviaturas

As abreviaturas empregadas nessa tese correspondem a acrônimos para os seguintes termos técnicos em inglês:

AN – *Allocation Network*

ASIC – *Application Specific Integrated Circuit*

ATM - *Asynchronous Transfer Mode*

CDFG - *Control Data-Flow Graph*

CMB – *Chandy/Misra/Bryant*

DDE - *Distributed Discrete Event*

DFG - *Data-Flow Graph*

FIFO – *First In, First Out*

IN - *Inter-connection Network*

IPC - *Input Port Controllers*

LP – *Logical Process*

LVT – *Local Virtual Time*

LVTH – *Local Virtual Time Horizon*

OPC - *Output Port Controllers*

PC - *Port Controllers*

RMI - *Remote Method Invocation*

RN - *Routing Network*

ROM – *Read-Only Memory*

SN – *Sorting Network*

VC – *Virtual Channel*

VCC – *Virtual Channel Connection*

VCI – *Virtual Channel Identifier*

VCL – *Virtual Channel Link*

VP – *Virtual Path*

VPC – *Virtual Path Connection*

VPL – *Virtual Path Link*

VPI – *Virtual Path Identifier*

1. Introdução

1.1 Motivação e Objetivos

Sistemas embutidos são geralmente sistemas computadorizados especializados e podem fazer parte de um sistema ou máquina maior [3]. Um sistema embutido tanto pode possuir um único microprocessador com os programas armazenados em uma memória ROM e um ASIC como pode ser um mini-PC controlado por sistemas operacionais especiais. Sistemas embutidos estão presentes em relógios, carros, fornos de microondas, televisores, telefones celulares, aparelhos de som, etc. A memória e o tamanho físico dos sistemas embutidos são limitados, o que demanda que o hardware e o software sejam pequenos.

Tecnologias como a computação distribuída, sistemas inteligentes, computação móvel e a Internet aumentaram a capacidade e a abrangência dos sistemas embutidos. Com isso, vários sistemas embutidos hoje utilizam processadores de 32 e 64 bits, possuem interfaces gráficas, como PDAs e telefones celulares, e alguns são programados com linguagens de alto nível como Java, C e C++[4].

O software também constitui parte importante de um sistema embutido, o que torna a escolha da linguagem de programação e plataformas tão significativas para o sucesso do projeto. Muitos sistemas embutidos utilizam algum sistema operacional, como QNX, LynxOS, WindowsCE e VxWorks. Cada tipo de sistema operacional oferece estrutura para a execução em diferentes processadores, com capacidade de funcionamento em tempo real, suporte a redes e outras funções específicas.

A simulação é uma parte importante do co-projeto de hardware/software de um sistema embutido pois oferece a possibilidade de verificar o funcionamento antes da implementação do sistema. Este trabalho apresenta um simulador distribuído orientado a eventos [1], desenvolvido em linguagem de programação Java de acordo com a metodologia conservadora de simulação distribuída [2]. Pretendemos com esse trabalho fornecer uma ferramenta para auxiliar e acelerar o processo de

desenvolvimento de sistemas embutidos, que vem crescendo em complexidade nos últimos anos, demandando cada vez mais tempo para simulação.

Com um simulador distribuído, a execução dos módulos do sistema pode ser repartida entre diversos computadores, sendo possível, para muitos sistemas, aproveitar o paralelismo de funções do sistema para reduzir o tempo da simulação. Como o simulador pode ser utilizado tanto em rede local como através da Internet, projetistas em localidades distantes podem realizar a simulação de um projeto com módulos armazenados nos seus computadores de trabalho. Nesse caso, a vantagem oferecida pelo simulador distribuído está na facilidade de uso, pois os projetistas não precisam transportar seus módulos para uma única rede local. O tempo requerido para a simulação, entretanto, tende a crescer devido aos atrasos da comunicação via rede.

O simulador apresentado pode ser utilizado para validar um projeto a partir da fase de especificação. O sistema a ser simulado deve ser descrito por grafos de fluxo de dados, cujos nós representam funções executadas quando as entrada representadas nas arestas do grafo estão disponíveis. As funções devem ser em código Java. A seção 1.2 aborda algumas razões para o uso de Java tanto como linguagem de descrição das funções do sistema e como linguagem de desenvolvimento do simulador.

Esse simulador foi desenvolvido para atender às características particulares do projeto de sistemas embutidos. Para tanto, o modelo adotado é adequado para descrição tanto de funções de controle como de processamento de sinais, que são comumente encontradas nesses sistemas. Como muitos sistemas embutidos precisam satisfazer especificações de tempo real, o simulador permite a observação do comportamento temporal através dos tempos e taxas de execução de tarefas. Além disso, o simulador calcula o tamanho dos buffers de entrada dos módulos e a utilização dos elementos de processamento e canais de comunicação. Essas informações possibilitam estimativas preliminares das dimensões físicas, do consumo de energia e indicam os módulos do projeto para os quais devem ser direcionadas as aplicações de recursos humanas e financeiras do projeto.

Para uso de simulador, os módulos do sistema devem ser codificados em Java. As primitivas da interface do simulador com esses módulos são simples, requerendo pouquíssimas alterações no código escrito pelo projetista. A seção 2.4 descreve detalhadamente essa interface.

Esse trabalho incluí descrições detalhadas dos exemplos desenvolvidos para ilustrar o desempenho e a aplicação do simulador em projetos. O objetivo é facilitar comparações com outros trabalhos sobre simuladores.

1.2 Java

Nessa seção, apresentamos os fatores que levaram a escolha de Java tanto para descrição das funções do sistema a ser simulado.

1.2.1 Java como Linguagem de Especificação de Sistemas Embutidos

A escolha da linguagem de especificação do sistema é um ponto crítico numa metodologia de co-projeto de hardware/software, pois a partir dessa especificação serão feitas a validação funcional e o particionamento [10][11] do sistema. A validação funcional pode ser feita pela simulação de uma especificação executável do sistema, sendo assim, a linguagem de especificação deve permitir uma simulação eficiente dessa especificação.

Após o particionamento funcional, para a descrição das funções do sistema implementadas em hardware, é necessária uma linguagem capaz de expressar concorrência e eventos com restrição de tempo como Verilog ou VHDL. Para a descrição das funções dos módulos implementados por software, linguagens de programação como C++ ou Java possuem todas as características necessárias para descrever o comportamento das funções de software do sistema [10].

Essa é a situação tradicional, que leva a um projeto que inclui em várias linguagens. Como os projetistas nem sempre dominam todas as linguagens, a comunicação entre os grupos de projeto e a manutenção de documentos se torna difícil e propensa a erros.

A linguagem Java, por outro lado, possui algumas características [12] que facilitam a especificação dos módulos a serem implementados tanto hardware como em software como, por exemplo, suporte para a programação concorrente e modelo de memória com comportamento previsível.

No trabalho realizado em [10], os autores mostram como utilizar Java como linguagem única na especificação de sistemas embutidos. No método proposto pelos autores, a análise da especificação Java do sistema gera um CDFG (*Control Data-Flow Graph*) em duas fases. Na primeira fase, é feita uma análise das dependências de dados para detectar *loops* no fluxo de controle do programa. Isso é feito para determinar o fluxo global de controle e de dados do programa. Na segunda fase, são feitas a identificação e extração da concorrência. Os *loops* do programa são analisados para determinar se existem dependências entre as iterações do *loop*. A análise também identifica o uso de *threads* no programa. Com isso, consegue-se extrair a concorrência implícita (*loops*) e explícita (*threads*) da especificação do sistema. A possibilidade de extrair a concorrência implícita é importante durante a síntese automática dos módulos implementados em hardware.

No trabalho desenvolvido em [11], os autores propõem uma metodologia, chamada de *Refinamento Formal Sucessivo*, para o uso de Java na especificação de sistemas embutidos reativos[1]. A metodologia proposta permite que sistemas sejam projetados em linguagens de propósito geral e então refinados numa forma restrita consistente com o modelo a ser descrito. No caso desse trabalho, são modelados sistemas síncronos e reativos. O refinamento gera um programa que é determinístico, utiliza memória restrita e executa em tempo limitado. Um programa com essas propriedades é adequado para ser utilizado como especificação de sistemas embutidos reativos.

Em [14], Java é utilizada como linguagem de especificação para programação em tempo real. Os autores criaram uma especificação para fornecer uma plataforma para o uso de Java para o desenvolvimento de software em tempo real, permitindo aos programadores analisar corretamente o comportamento temporal do software.

1.2.2 Java como Linguagem de Desenvolvimento do Simulador

Outras características que justificam a escolha de Java para a implementação do simulador são listadas a seguir:

[1] Sistemas reativos são aqueles que interagem com o ambiente, geralmente com algum tipo de restrição de tempo

- Java é orientada a objetos. Como nas linguagens orientadas a objeto, os objetos interagem apenas através do envio e recepção de mensagens [15], essa metodologia de programação acelera o desenvolvimento de novos programas, o que facilita a manutenção e a reutilização do software. Podemos incluir novos métodos e variáveis em um objeto sem causar alterações na interação desse objeto com outros objetos do programa, facilitando assim a manutenção do sistema. Além disso, os objetos podem ser reaproveitados em diversos projetos. A reutilização de objetos diminui a quantidade de código novo que precisa ser escrito e diminui a quantidade de erros do programa, uma vez que o objeto reaproveitado já foi depurado [16].
- A linguagem Java possui extensas bibliotecas de rotinas para facilitar o uso de protocolos TCP/IP como o HTTP e o FTP. As aplicações feitas em Java podem acessar objetos através de redes com o uso de URLs com a mesma facilidade com que seriam feitos acessos aos arquivos em um disco local. Essa característica facilita o desenvolvimento de aplicações distribuídas, eliminando do programa construções específicas para lidar com acesso a rede. Facilita também a construção dos módulos do projetista, sendo que o mesmo não precisa alterar o código do módulo para que seja executado no simulador distribuído ou no seqüencial.
- Além das verificações em tempo de compilação para detectar possíveis problemas, também são feitas verificações dinâmicas em tempo de execução para eliminar situações de erro. Essa característica da linguagem diminui a incidência de erros em tempo de execução, o que muito facilitou a depuração do código do simulador.
- O código compilado de um programa Java pode ser executado em qualquer plataforma sem a necessidade de alterações. Java não tem nenhum aspecto de sua especificação padrão que seja dependente da plataforma. As funções e comandos da linguagem podem estar disponíveis em diferentes versões do compilador. Por exemplo, o tipo *int* em Java é sempre um inteiro de 32 bits, em C++ pode ser um inteiro de 32 ou 16 bits, dependendo da implementação do compilador da linguagem. Isso facilita a migração do simulador entre diferentes plataformas computacionais. O simulador foi utilizado nas plataformas *Windows* e *Solaris* sem que nenhuma alteração fosse realizada [13].

- Java é interpretado, mas o desempenho revelou-se adequado para o funcionamento do simulador, conforme pode ser comprovado através dos exemplos no Capítulo V. A linguagem C++ pode gerar códigos até 10 vezes mais rápidos que o Java interpretado. Para aplicações onde o desempenho é crítico, os compiladores *JIT(Just in Time Compilers)* executam programas Java com um desempenho semelhante aos desenvolvidos na linguagem C++ [17][18].
- A linguagem Java implementa monitores [19][20]. Monitores encapsulam dados que só podem ser observados e modificados pelos procedimentos de acesso ao monitor. Em qualquer instante, apenas um procedimento de acesso pode ser executado, garantindo acesso mutuamente exclusivo aos dados encapsulados no monitor e evitando erros resultantes no compartilhamento de informações.

1.3 Trabalhos Anteriores

Nessa seção comentamos alguns dos trabalhos dedicados ao desenvolvimento de simuladores distribuídos com características semelhantes ao apresentado nessa tese.

Em [5], os autores apresentam a ferramenta *Ptolemy 2* que auxilia a construção e simulação de modelos heterogêneos voltados para sistemas embutidos. A ferramenta foi desenvolvida em Java e opera em diversos domínios, que caracterizam a forma de modelar o sistema: domínio de tempo contínuo, de eventos discretos, de tempo discreto, de máquinas de estado finito, etc. O domínio DDE (*Distributed Discrete Event*) permite simulações com um modelo semelhante ao usado nesse trabalho de tese. O domínio DDE implementa um simulador distribuído orientado a eventos seguindo a metodologia conservadora, exposta em [2]. A ênfase desse domínio do *Ptolemy* não é aumentar a velocidade de execução, mas sim em estudar a utilidade da simulação distribuída conservadora para modelar e projetar sistemas distribuídos com restrições de tempo de execução. A aplicação deve ser descrita exclusivamente a partir de blocos existentes na biblioteca dessa ferrramenta. Assim a interface do projetista é completamente diferente da proposta nesse trabalho. Aliás, a necessidade de reescrever

os exemplos impediu a comparação do nosso trabalho com esse simulador. Além disso, não foram fornecidos pelos autores dados relativos ao desempenho dessa ferramenta.

Em [6], os autores apresentam um ambiente de simulação orientadas a eventos desenvolvido em Java. O simulador é capaz de realizar simulações seqüenciais e distribuídas com o uso da estratégia otimista com o protocolo *time-warp* [7]. O artigo não apresenta exemplos nem resultados de desempenho, impossibilitante a comparação com o nosso trabalho.

Em [8], os autores propõem um simulador distribuído que utiliza a Internet como plataforma de execução. Os autores criaram um simulador distribuído baseado em Java com o uso do protocolo de sincronização CMB (*Chandy/Misra/Bryant*) [2] (veja a seção 4.2.4 para maiores detalhes sobre o protocolo de sincronização CMB). O simulador é de propósito geral, de maneira que pode ser utilizado com vários modelos de simulação diferentes. O código, uma vez escrito, pode ser executado em qualquer lugar da Internet e os autores mencionam um ganho de desempenho da simulação distribuída entre 1,62 a 5,84 vezes sobre o desempenho da simulação seqüencial. Esses ganhos de desempenho teriam sido obtidos na execução do simulador em rede local. Os autores não forneceram qualquer indicação sobre as aplicações utilizadas nos testes de desempenho. Além disso, o código desse simulador não está disponível na Internet sendo impossíveis comparações de seu desempenho com o do nosso simulador.

Em [9], os autores propõem o *Simkit*, um conjunto de classes feitas em Java para criar modelos para simulação orientada a eventos discretos. O *Simkit* pode ser utilizado como aplicativo ou como *applets* em um navegador de Internet. Esse simulador pode ser executado de maneira distribuída, com o uso das facilidades disponíveis na linguagem Java. Os autores não apresentam resultados de desempenho no artigo. A comparação do *Simkit* com o nosso simulador não é possível, uma vez que o *Simkit* foi criado com o propósito de ser uma ferramenta de baixo custo com a capacidade de executar em qualquer plataforma através da Internet. O nosso simulador tem como um dos seus objetivos a redução do tempo de simulação, tendo toda a sua concepção voltada para o aumento de desempenho.

1.4 Principais Características do Simulador

Resumimos a seguir as principais características do nosso simulador. As explicações e exemplos nos próximos capítulos mostram o papel desempenhado por essas características durante o projeto de sistemas embutidos.

- O modelo do sistema é uma versão modificada do grafo de fluxo de dados
- Uso de Java como linguagem de especificação de sistemas [10]:
    - Permite ao projetista descrever concorrência, através de *threads* incluídas na linguagem.
    - Java é uma linguagem com um controle rígido de acesso à memória, não permite o uso de ponteiros e outros tipos de acessos que possam gerar erros difíceis de rastrear.
- O simulador é orientado a eventos (veja o Capítulo 3.).
- O simulador implementa a metodologia de simulação distribuída conservadora, que garante uma operação sem *deadlock* (veja o Capítulo 4.).
- O relatório emitido pelo simulador no final da simulação contém as seguintes informações: tempo total de execução, listagem completa dos eventos, número de execuções de cada módulo, quantidade de eventos destinados a cada módulo, informações sobre o uso dos módulos e dos canais, tamanho máximo atingido pelos *buffers* e número de eventos nos *buffers* após o término da simulação.
- O simulador trabalha com módulos genéricos. Isto é, o projetista pode simular qualquer módulo codificado em Java, com pouquíssimas alterações.
- A mesma descrição da aplicação em Java pode ser utilizada para a simulação seqüencial ou distribuída. Para a simulação distribuída, os computadores podem estar situados em qualquer local, desde que estejam conectados na Internet e possuam os arquivos necessários para o funcionamento do simulador.
- O simulador foi codificado em Java 2, sendo usado o compilador *JBuilder* 3 para execução na plataforma *Windows* e o compilador JDK 2 para execução na plataforma *Solaris*.

1.5 Organização da Tese

Essa tese está organizada em 6 capítulos e 3 apêndices. O Capítulo 2 descreve o modelo do sistema, compara o modelo proposto com outros modelos semelhantes e mostra os parâmetros necessários para o funcionamento do simulador. O Capítulo 3 descreve o funcionamento do simulador orientado a eventos seqüencial, detalhando o funcionamento dos módulos do simulador. O Capítulo 4 discute a simulação orientada a eventos distribuída, mostra como foi implementado o simulador distribuído desenvolvido nessa tese e discute as vantagens e desvantagens da metodologia escolhida. O Capítulo 5 descreve os exemplos usados para testar o simulador e apresenta os resultados de simulação. O primeiro exemplo consiste numa implementação do algoritmo de Barnes-Hut para cálculo de força gravitacional. O segundo exemplo é uma pequena rede de comutadores ATM. Na execução dos exemplos são apresentados resultados comparando o desempenho do simulador distribuído com o do simulador seqüencial. No Capítulo 6 são apresentadas as conclusões do trabalho e algumas sugestões para trabalhos futuros. O Apêndice A apresenta um sistema de controle descrito com o modelo empregado pelo simulador. O Apêndice B descreve todos os objetos criados na implementação do simulador. O Apêndice C contém exemplos dos arquivos de resultados emitidos pelo simulador.

2. Modelo do Sistema e Comportamento Dinâmico

Nesse capítulo detalharemos o modelo empregado para representar o sistema e as informações de entrada e saída necessárias para o funcionamento do simulador.

2.1 Modelo do Sistema

O sistema é representado por uma versão modificada do diagrama de fluxo de dados [22]. Esse modelo é funcional e pode ser usado desde a etapa de especificação no projeto de um sistema embutido. O diagrama de fluxo de dados (DFG) é um diagrama que contém um conjunto de *atividades* (transformações) conectado por canais, que representam o fluxo de dados. Nesse caso, as *atividades* são representadas graficamente por um conjunto de nós, e os *canais* são representados por um conjunto de vértices orientados. Os nós (*atividades*) podem ser antecessores ou sucessores, que indicam a dependência de dados/controle entre os módulos do sistema, de acordo com o seu posicionamento no grafo. As *atividades,* que correspondem aos módulos funcionais do sistema, devem ser descritas por objetos da linguagem Java para o uso dos simuladores descritos nesse trabalho.

A versão modificada do diagrama de fluxo de dados, utilizada na descrição do sistema para o simulador, permite especificar a quantidade de informações necessárias para ativar cada módulo do sistema. Esse modelo permite ao projetista especificar módulos que processam em uma única execução dados produzidos em mais de uma execução dos módulos antecessores.

A Figura 2.1. mostra um exemplo de DFG. No DFG são representados os módulos e os canais. Nos canais é indicado o nome do canal e o número mínimo de execuções do nó antecessor para que o módulo sucessor possa ser executado. Pela figura podemos perceber que o módulo *t3* necessita de 3

execuções do módulo *t1* e uma execução do módulo *t2* para executar uma única vez. O módulo *t3* só deve ser ativado quando todas essas informações estiverem disponíveis. Podemos notar também que os módulos *t1* e *t2* são antecessores de *t3,* que por sua vez é sucessor de *t1* e *t2*. Nos próximos capítulos desse trabalho, essa versão modificada do diagrama de fluxo de dados é referenciada como modelo **multitaxa**. Esse modelo é equivalente ao modelo síncrono de fluxo de dados (SDF- *synchronous data flow*) [5], sob a restrição de buffers com dimensão mínima. Esse modelo também pode ser representado pelo diagrama de *Petri* estendido descrito em [23].

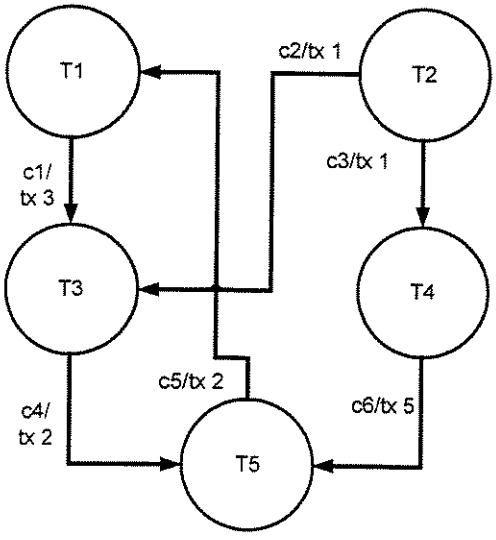


Figura 2.1. Exemplo de DFG.

O modelo proposto permite ao projetista analisar a dependências de dados entre os módulos do sistema. Com essa análise, o projetista pode visualizar a concorrência na especificação do sistema. A representação da taxa de envio de dados entre os módulos permite estimar o consumo de memória do sistema, pela análise dos tamanhos dos *buffers* que acumularam os dados enviados nos canais de entrada dos módulos.

Veja o Apêndice A para um exemplo da representação de um sistema de controle com o uso do modelo multitaxa.

2.1.1 Comparação entre o modelo SDF e o modelo Multitaxa

Nessa seção vamos mostrar um estudo comparativo entre o modelo SDF (*Synchronous Dataflow*) e o modelo Multitaxa proposto nesse trabalho de tese. O modelo SDF [24] foi criado para representar sistemas embutidos, particularmente DSPs.

A Figura 2.2 mostra uma comparação entre representações gráficas de um mesmo sistema usando o modelo SDF e o modelo Multitaxa. Tanto o modelo SDF quanto o modelo Multitaxa são modelos de fluxo de dados, onde os módulos só podem ser disparados quando todos os dados de entrada necessários estiverem disponíveis. No modelo SDF são especificadas as taxas de envio e recepção de dados, sendo que a taxa de envio é indicada na saída do módulo enquanto que a taxa de recepção é indicada na entrada do módulo [24]. O modelo Multitaxa representa apenas a taxa de recepção de dados, indicada na entrada do módulo.

No modelo SDF os dados são enviados no final da execução do módulo. No modelo Multitaxa, os dados podem ser enviados durante a execução do módulo.

Tanto o modelo SDF quanto o modelo Multitaxa são modelos de fluxo de dados, onde os módulos só podem ser disparados quando todos os dados de entrada necessários estiverem disponíveis.

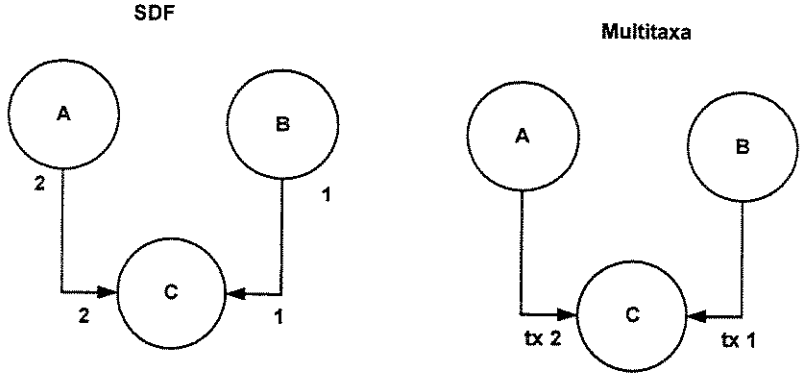


Figura 2.2 Comparação entre o SDF e o modelo Multitaxa

No modelo SDF, os módulos respeitam uma ordem de execução pré-determinada. É possível determinar vários escalonamentos para um conjunto de módulos [24], sendo que os mesmos devem ser

compatíveis com o tamanho do *buffer* de entrada disponível. Portanto, com modelo SDF, o projetista determina o escalonamento dos módulos do sistema.

No modelo SDF da Figura 2.2, por exemplo, caso o módulo *C* possua um *buffer* de entrada com 10 posições, 8 posições para armazenar saídas do módulo *A* e 2 posições para armazenar saídas do módulo *B*, alguns dos escalonamentos possíveis são os seguintes:

AABC→ 2 execuções do módulo A e uma do módulo *B*, resultando em uma execução de *C*

BAAC→ uma do módulo *B* e 2 do módulo *A* e *B*, resultando em uma execução de *C*

AAAABBCC→ 4 execuções do módulo *A* e 2 do módulo *B*, resultando em 2 execuções de *C*

BAAAABCC→ 2 execuções do módulo *B* e 4 do módulo *A* resultando em 2 execuções de *C*

Note que o módulo *C* só pode ser executado quando o módulo *A* executar 2 vezes e o módulo *B* executar uma vez módulo. Desde que obedeça a essa restrição, o projetista tem várias opções para escalonamento dos módulos do sistema. Por sua vez, o modelo Multitaxa realiza o escalonamento com *buffers* de tamanho mínimo, então, o módulo *C* é disparado assim que os dados estiverem disponíveis, resultando no escalonamento mostrado abaixo:

AABC

BAAC

AAAABCBC

BAACAABC

Note que o módulo *C* é disparado logo depois que 2 execuções *A* e uma execução de *B* estiverem completas. Essa condição de execução imediata do módulo quando os dados de entrada estiverem disponíveis possibilita a utilização de *buffers* de entrada, sendo interessante para reduzir o tamanho da memória requerida para o sistema.

Esse trabalho adota o modelo multitaxa e emite no final da simulação o tamanho máximo atingido pelos *buffers para o conjunto de entradas do sistema simulado*. Essas entradas são especificadas pelo projetista.

2.2 Parâmetros para a Simulação - Informações sobre o Sistema

Nessa seção, detalharemos os parâmetros a serem fornecidos pelo projetista para o uso do simulador distribuído.

2.2.1 Discretização do Tempo para Simulação

O simulador trabalha com sistemas discretos, isto é, o tempo progride em intervalos inteiros, por exemplo, 1, 2, 3, cabendo ao projetista realizar o mapeamento matemático necessário para converter o "tempo real" do sistema para o "tempo discreto" do simulador.

2.2.2 Módulos do Projetista

O projetista deve fornecer ao simulador uma estimativa para o tempo de execução de cada módulo. Esses são os tempos estimados para a execução dos módulos em uma determinada plataforma computacional. Esses tempos devem ser expressos de acordo com a discretização de tempo adotada pelo simulador, pois serão utilizados, junto com o atraso de propagação nos canais de comunicação, para gerenciar a execução de eventos durante a simulação.

É possível simular sistemas de tempo real que possuam *deadlines* rígidos de execução, pois nesses sistemas, os módulos possuem um tempo máximo de execução que não deve ser ultrapassado. Com a ajuda do simulador, podemos descobrir se os módulos do sistema cumprem os prazos previstos de execução.

O modelo multitaxa permite especificar diferentes tempos de execução para um módulo. Isso pode ocorrer quando o módulo possuir mais de um canal de saída, representado por uma linha de execução para cada canal de saída. Valores distintos de atraso de propagação nos canais de saída permitem uma maior flexibilidade na descrição do módulo, para melhor representar sistemas de tempo real.

É importante ressaltar que o simulador não trata módulos com tempos de execução que variam de acordo com os valores de entrada [25], pois o tempo de execução de cada módulo deve ser especificado antes da simulação.

A Figura 2.3. ilustra o conceito de linhas de execuções. A figura mostra 4 módulos, cada um com seus canais de comunicação, pertencentes a uma das duas linhas de execução, *L1* e *L2*. A linha *L1* é composta pelos canais de comunicação *c1*, *c2* e *c3*. A linha *L2* é composta pelos canais de comunicação *c4* e *c5*. Cada canal de comunicação possui um atraso associado. Note que o *Módulo 1* possui canais nas duas linhas de execução, com atrasos distintos.

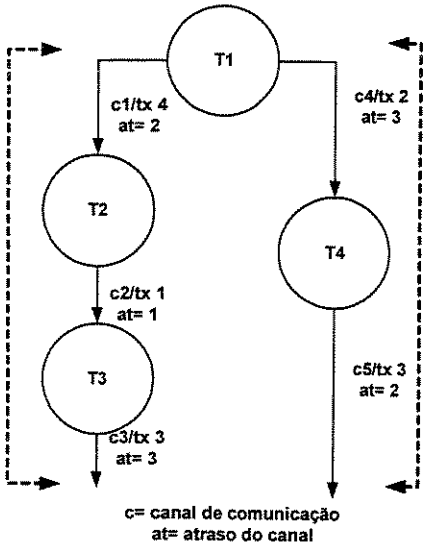


Figura 2.3 Linhas de execução.

O projetista deve preencher um arquivo com os nomes dos módulos do sistema. Essa informação é necessária para que o simulador possa criar e executar os objetos que representam os módulos do projetista (veja a seção B.3 para maiores detalhes sobre o arquivo com os nomes do módulo).

2.2.3 Canais de Comunicação

As informações referentes aos canais de comunicação entre os módulos do sistema, necessárias para o uso com o simulador, são listados a seguir:

- Número de identificação dos canais: Representa o canal de entrada de dados de um determinado módulo do sistema. Esse número irá constar no campo *Destinatário,* do evento de simulação.
- Quantidade mínima de eventos: é a quantidade de eventos que o módulo precisa receber nesse canal para ativar a execução do módulo. Essa quantidade é representada por um número inteiro positivo.
- Tempo de propagação dos dados no canal (atraso do canal): é o tempo transcorrido entre o envio e a recepção de um evento. Esse tempo permite ao simulador calcular o percentual de tempo de uso do canal em relação ao tempo total da simulação. Esse tempo é representado por números inteiros positivos.

2.2.4 Descrição Funcional dos Módulos

O projetista deve fornecer o código dos módulos do sistema em linguagem Java. Os módulos devem ser descritos como objetos e incluir pequenas alterações para o funcionamento do módulo no simulador. Essas alterações são listadas a seguir:

- A classe do projetista deve ter o formato: *Class Nome_Do_Módulo implements Processo.*
- O construtor do objeto deve ter o formato: *Nome_Do_Módulo (EnviaDistribuído Evd, EnviaEventos Ee, Relógio Rl)* para uso com o simulador distribuído. Esse construtor permite ao simulador criar automaticamente os objetos da classe do projetista, e permite que o projetista inicialize os objetos necessários para que o seu módulo possa interagir com o simulador.
- O módulo do projetista deve criar os seguintes objetos: *EnviaEventos*, *EnviaEventosDistribuídos* e *Relógio*. O objeto *EnviaEventos* permite ao módulo do projetista enviar eventos para outros módulos do sistema que estejam localizados no mesmo processador,

o método *EnviaEventosDistribuídos* permite ao módulo do projetista enviar eventos para módulos situados em outros processadores e o objeto *Relógio* permite ao módulo do projetista tomar conhecimento do tempo corrente da simulação. Veja o Apêndice B. para maiores detalhes sobre os objetos do simulador.

2.2.5 Informações Adicionais

Descreveremos nessa seção as informações adicionais que o projetista deve fornecer para o funcionamento do simulador.

- O projetista deve criar um arquivo com os eventos a serem utilizados como entradas do sistema. Outra possibilidade consiste em gerar eventos dentro do construtor dos seus objetos. Esses eventos são utilizados para iniciar a simulação.
- O projetista pode especificar a duração da simulação em unidades de tempo discreto. Caso esse tempo não seja fornecido, o simulador encerra a simulação quando não houver mais eventos para serem executados. Se, durante a simulação distribuída, não houver o envio de eventos pelos módulos remotos, ainda que o tempo de término estipulado para a simulação não tenha sido atingido, os simuladores locais possuem a capacidade de detectar a ausência de comunicação no canal e encerrar a simulação. Esse procedimento é necessário na hipótese de ocorrer algum tipo de erro em um dos computadores utilizados na simulação. Com isso, os outros computadores podem terminar a sua simulação e emitir resultados. Veja a seção B.11 para maiores detalhes.
- O projetista pode escolher para que módulos serão listados os eventos de entrada. Isso é importante porque os arquivos de eventos podem atingir tamanhos consideráveis. Como essa listagem de eventos tem a finalidade de facilitar a depuração dos módulos do projetista, nem sempre é desejável que todos os eventos sejam rastreados.
- A alocação dos módulos nos diversos computadores deve ser feita manualmente. É importante notar que a simulação terá um melhor desempenho se apenas módulos *convergentes* forem alocados em cada computador. Módulos convergentes são aqueles onde as linhas de execução

alocadas para o computador tem algum módulo em comum, como mostrado na Figura 2.4. Módulos não-convergentes são aqueles onde as linhas de execução não possuem módulos em comum, podendo executar em paralelo em computadores diferentes.

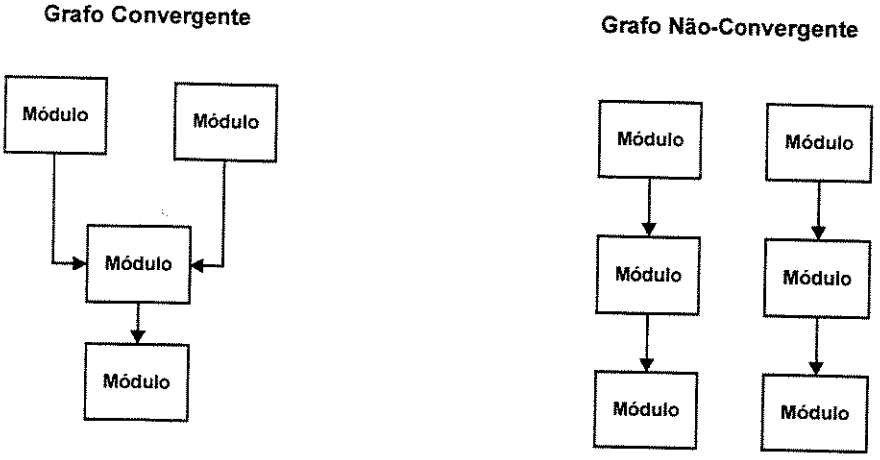


Figura 2.4. Exemplo de tipos de grafo.

2.3 Relatório da Simulação: Comportamento Dinâmico

Ao término da simulação, o simulador emite arquivos com os resultados descritos abaixo:

- Tempo total de execução. O simulador fornece o tempo total transcorrido em unidades de tempo discreto.
- Número de execuções e tempo total de execução para cada canal e para cada módulo. O simulador fornece também o percentual do tempo de utilização dos módulos e canais em relação ao tempo total da simulação, dado pelas fórmulas abaixo:

$$Utilização\_Módulo = 100 * \frac{Tempo\_Total\_de\_Execução\_do\_Módulo}{Tempo\_Total\_Simulação}$$

$$Utilização\_Canais = 100 * \frac{Tempo\_Total\_de\_Execução\_do\_Canal}{Tempo\_Total\_Simulação}$$

onde:

Tempo_Total_de_Execução_do_Módulo= número de execuções do módulo multiplicado pelo tempo de execução do módulo.

Tempo_Total_de_Execução_do_Canal= número de execuções do canal multiplicado pelo atraso do canal.

- Dados relativos aos eventos:
  - Quantidade de eventos destinada a cada módulo. O simulador fornece a quantidade de eventos destinados ao módulo. Observe que alguns desses eventos podem não resultar em execução do módulo, ficando acumulados no *buffer* de entrada após o término da simulação.
  - Listagem dos eventos. O simulador fornece a listagem nos eventos dos canais especificados pelo projetista.
  - Número de eventos nos buffers de entrada dos módulos após a simulação. O simulador fornece o número de eventos que sobraram nos buffers de cada canal dos módulos do projetista. O simulador também fornece o tamanho máximo atingido por esses buffers durante a simulação.

2.4 Exemplo

Nessa seção mostraremos uma classe com as alterações necessárias para o uso do simulador.

O trecho de código da Figura 2.5 ilustra uma classe utilizada no exemplo descrito na seção 5.1. As partes que fazem a interface do código do projetista com o simulador estão realçadas em negrito, e o código está em itálico.

Essa classe envia apenas eventos locais, por isso ela cria um objeto do tipo *EnviaEventos*. Caso fosse necessário enviar eventos distribuídos, seria necessária a criação de um objeto da classe *EnviaDistribuído*.

O método *Consumidor* é necessário para que o simulador possa enviar os eventos para o módulo do projetista. Esse método faz o papel de interface entre os módulos do projetista e o simulador. A classe recebe eventos através do objeto *evento*.

O envio de eventos é feito pelo uso do método *Ev.Enviar*, e o módulo do projetista só precisa informar ao simulador o conteúdo a ser transmitido e o destinatário do evento.

public class Control ***implements Processo*** ➔ é necessário para o simulador poder inicializar os objetos do projetista automaticamente

```
{
```

*public Control(**EnviaEvento e, EnviaDistribuído evd, Relogio rr**)* ➔ é necessário para que o módulo do projetista possa enviar eventos locais e distribuídos.

```
{
 Ev=e;
}
```

public void Consumidor(Evento[][][] evento) ➔ o objeto evento contém os eventos destinados a esse módulo.

```
{
  boolean aux= ((Boolean)evento[0][0].RetornaConteudo()).booleanValue();
  if(aux==false)
    Ev.Enviar(new Boolean(false),1);
  else
    Ev.Enviar(new Integer(0),1);
}
EnviaEvento Ev;
boolean[] flag;
```

Figura 2.5. Exemplo de código de um módulo de aplicação.

2.5 Conclusão

Mostramos nesse capítulo o modelo do sistema e explicamos o grafo de fluxo de dados multitaxa. Para maiores detalhes sobre os arquivos que armazenam os parâmetros utilizados pelo simulador, veja a seção B.3, no Apêndice B. No próximo capítulo, iremos explicar o funcionamento da simulação seqüencial orientada a eventos.

3. Simulação Orientada a Eventos

Nesse capítulo iremos introduzir os conceitos básicos da simulação orientada a eventos.

3.1 Introdução

A simulação orientada a eventos é uma metodologia muito utilizada para simular sistemas de tempo real como: circuitos digitais [26], redes ATM [27], sistemas de comunicação [28], além de situações do cotidiano, como o fluxo de clientes numa lanchonete e filas de bancos [29].

Os objetos da simulação orientada a eventos geralmente são modelados com base em objetos físicos do mundo real, que permitem a simulação das suas características assíncronas [8]. Esses objetos são programados para reagir como os objetos do mundo real [30]. Esse realismo da modelagem do problema unido à capacidade de simulação de modelos assíncronos torna a simulação orientada a eventos um recurso aplicável a essas situações. A simulação orientada a eventos pode também ser aplicada a modelos síncronos. Os modelos assíncronos podem incluir atrasos definidos antes ou durante a simulação.

Durante a simulação orientada a eventos, a comunicação entre os módulos do sistema é feita através da propagação de eventos, que são retirados de uma *fila de eventos*, ordenada por estampa de tempo de execução.

O simulador seqüencial orientado a eventos descrito nesse capítulo foi desenvolvido para funcionar como núcleo do simulador distribuído, além de servir para realizar simulações nas situações onde a simulação distribuída não é possível ou desejável.

A maior desvantagem da simulação orientada a eventos está relacionada com o gerenciamento da fila de eventos. A inserção, propagação e ordenação de eventos demandam um custo computacional considerável, fazendo com que o desempenho dessa metodologia de simulação seja inferior ao desempenho de outras metodologias.

Os simuladores de código compilado [31][32][33], por exemplo, possuem um desempenho muito superior aos simuladores orientados a eventos. Essa categoria de simuladores é utilizada para simular circuitos lógicos. A descrição do circuito a ser simulado é transcrita diretamente em código, que é organizado em níveis para garantir que sempre que um componente for simulado, os valores corretos de suas entradas estão disponíveis. A simulação é realizada pela execução seqüencial do código, e cada componente é avaliado apenas uma vez por ciclo, o que elimina o gerenciamento de eventos. A maioria desses simuladores, entretanto, não pode simular modelos assíncronos.

A metodologia proposta em [26], promete unir a eficiência dos simuladores compilados com a generalidade dos simuladores orientados a eventos. O método pretende diminuir o *overhead* causado pelo gerenciamento da fila de eventos através da determinação da ordem de eventos através de uma "pré-simulação", realizada durante a compilação. A única restrição é que qualquer atraso especificado deve ser conhecido em tempo de compilação. Isso impede a simulação de modelos que possa gerar eventos com tempos arbitrários e não-determinísticos em tempo de simulação.

O exemplo usando o Algoritmo de Barnes-Hut, descrito na seção 5.1, não poderia ser simulado com o auxílio dessas metodologias, pois os eventos são conhecidos somente em tempo de execução, não sendo possível prever o momento em que os mesmo seriam gerados.

No exemplo descrito na seção 5.2 (Rede de Comutadores ATM), os módulos geradores de tráfego funcionam de forma aleatória, também impossibilitando o uso de simuladores compilados.

3.2 Funcionamento Básico

Num simulador orientado a eventos, os módulos do sistema agendam eventos na fila de eventos [29], ordenando-as por estampa de tempo de execução. Estampa de tempo é o instante de tempo no qual o evento será executado. Caso haja mais de um evento com a mesma estampa de tempo, a ordem obedece a prioridade atribuída pelo projetista. Caso os eventos, além da mesma estampa de tempo, tenham a mesma prioridade, a ordem de chegada na fila de eventos é adotada. Essa ordem depende da ordem de execução dos módulos durante a simulação.

Durante a simulação, o gerenciador de eventos remove o primeiro evento da fila e o envia para o módulo destinatário. O simulador mantém um *relógio* com o *tempo global da simulação*, que é atualizado sempre que um evento é retirado da fila. Os eventos são retirados em ordem crescente de estampa de tempo, o que impossibilita a inserção de eventos cuja estampa de tempo seja menor que o tempo corrente do relógio da simulação.

A Figura 3.1. mostra um exemplo para ilustrar o funcionamento básico de um simulador orientado a eventos. Na figura, os módulos *A*, *B*, *C* agendam eventos na fila. O módulo *A* agenda um evento destinado ao módulo *C* com estampa de tempo 3 (esse é o instante de tempo no qual o evento é retirado da fila). O módulo *B* agenda um evento destinado ao módulo *A* com estampa de tempo 2. O módulo *C* agenda um evento destinado ao módulo *B* com estampa de tempo 4. Os eventos são então adicionados na fila.

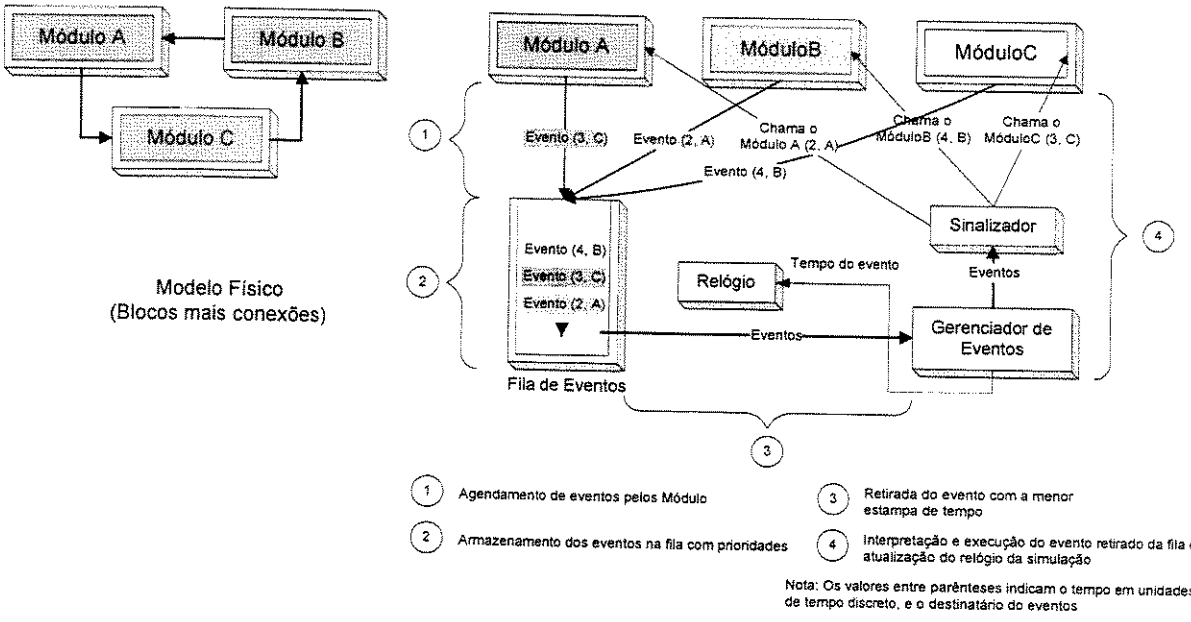


Figura 3.1. Funcionamento da simulação orientada a eventos.

O gerenciador de eventos retira o evento no topo da fila e o envia para o módulo correspondente. Nesse exemplo, o primeiro evento da fila é o (*2, A*), que é enviado pelo gerenciador de eventos para o módulo *A*. O gerenciador então atualiza o relógio da simulação para 2 e remove o próximo evento da fila. A execução do módulo pode ocasionar a inserção de outros eventos na fila. Esse ciclo de

simulação se repete até que não haja mais eventos na fila ou que o tempo de término especificado para a simulação seja atingido.

3.3 Simulador Orientado a Eventos

A implementação do simulador seqüencial orientado a eventos tem a estrutura mostrada na Figura 3.2.

O *Gerenciador* da *Simulação* ativa todos os objetos do simulador, chamando a seguir o objeto *RecebeEventos*, que remove o primeiro evento da árvore de eventos. O objeto *RecebeEventos* analisa o evento removido da árvore, atualiza o relógio de simulação e envia-o para o objeto *Sinalizador* correspondente. Depois de enviar o evento para o objeto *RecebeEventos*, o objeto *Árvore* executa uma chamada ao objeto *Resultados*, para o qual envia dados relativos a esse evento.

O objeto *Sinalizador* envia o evento para o módulo, quando todos os eventos necessários para a ativação desse módulo estiverem disponíveis. Veja a seção 3.3.1.1. para maiores detalhes sobre o objeto *Sinalizador*.

A execução dos módulos da aplicação pode gerar novos eventos. A aplicação deve incluir chamadas ao objeto *EnviaEventos*, que "monta" os eventos anexando as informações fornecidas pela execução do módulo, como a soma do tempo de execução do módulo mais o tempo de atraso do canal de saída do módulo (veja as seções 2.2.3 e 2.2.2). O novo evento é então armazenado na árvore de eventos.

O ciclo se repete até que não haja mais eventos a serem executados ou até que o instante de término da simulação seja atingido. Quando isso ocorre, o *Gerenciador* da simulação envia um requerimento para o objeto de *Resultados*, que emite relatórios a partir dos dados coletados durante a simulação.

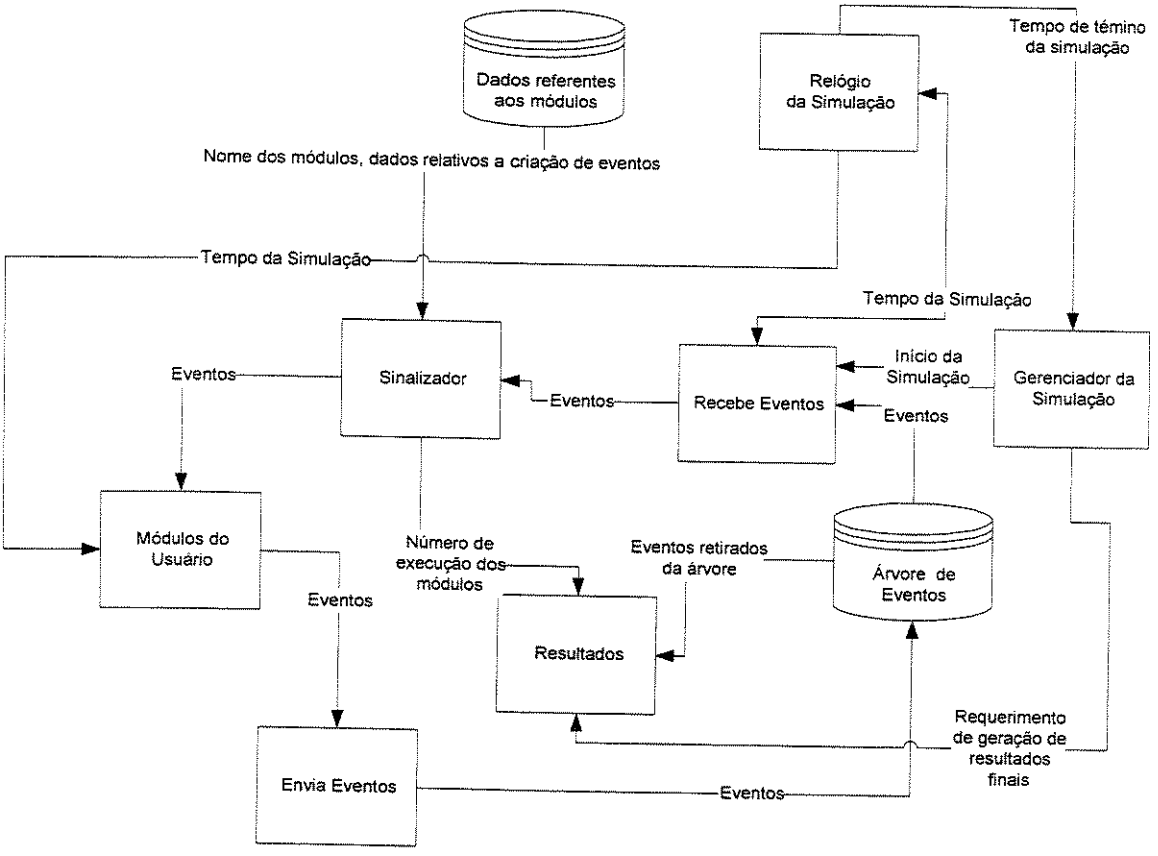


Figura 3.2. Estrutura do simulador seqüencial.

3.3.1 Características Especiais desse Simulador Seqüencial

3.3.1.1 Objeto Sinalizador

O objeto *Sinalizador* implementa a função **multitaxa** do simulador. Esse objeto armazena os eventos nos buffers de entrada de um módulo da aplicação, até que todos os eventos necessários para ativação do módulo da aplicação estejam disponíveis, de acordo com a especificação multitaxa. Quando esses eventos estão disponíveis, o objeto *Sinalizador* envia uma matriz com todos os eventos armazenados para o módulo, chamando também o objeto *Resultados*, que incrementa o contador de execuções referente a esse módulo da aplicação.

A estrutura do objeto *Sinalizador* é mostrada na Figura 3.3. O simulador fornece o tamanho máximo atingindo pelos buffers do objeto *Sinalizador* no final da simulação. Essa informação auxilia o projetista na determinação do tamanho dos *buffers* do sistema e na eliminação de erros responsáveis pelo número excessivo de eventos em determinados canais.

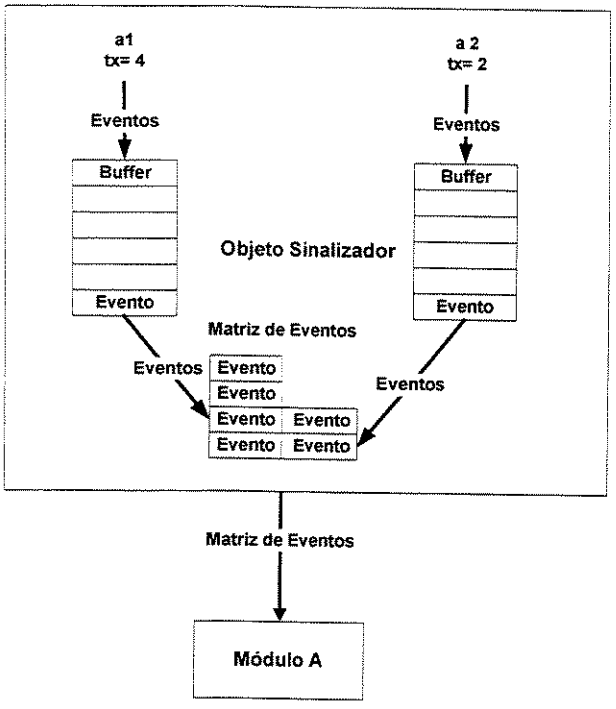


Figura 3.3. Estrutura do objeto Sinalizador

Por exemplo, suponha que um módulo do projetista tenha 2 entradas *a1* e *a2*, e execute 1 evento na entrada *a1* e 3 eventos na entrada *a2*. Se, o módulo receber 10 eventos na entrada *a1* e apenas 1 na entrada *a2*, o módulo não irá executar e o buffer da entrada *a1* deverá ter um tamanho suficiente para armazenar esses 10 eventos. Suponha que o módulo estará recebendo 30 eventos na primeira entrada para cada 3 eventos na segunda entrada e executará uma única vez durante a simulação. O simulador fornecerá no final da simulação a informação que o buffer da primeira entrada atingiu o tamanho

máximo de 30 eventos, o que permite ao projetista dimensionar o sistema ou corrigir erros de projeto para compensar esse problema.

3.3.1.2 Descrição do Evento

Os eventos representam ocorrências que alteram o estado do sistema. Durante a simulação, todas as comunicações, trocas de informações, dados de entrada e instruções de controle entre os módulos são realizadas através de eventos. Os eventos são representados por um objeto que contém as seguintes informações:

- Estampa de tempo do evento: a estampa tempo de execução é inserida automaticamente no evento pelo objeto *Envia Evento*. Essa estampa de tempo é a soma do atraso associado ao módulo e do atraso do canal de comunicação entre o módulo de origem do evento e o módulo de destino do evento, sendo inserido automaticamente pelo simulador. Estampas de tempo diferentes podem ser fornecidas pelo projetista (veja a seção B.1 para maiores informações)
- Prioridade do evento: valor inteiro utilizado para resolver a ordem de retirada dos eventos da árvore de eventos, quando existir mais de um evento com a mesma estampa de tempo.
- Destinatário : inteiro que identifica o módulo para qual o evento é destinado.
- Valor associado: é a informação transferida de um módulo da aplicação para outro. O valor pode ser qualquer tipo de dados do Java, ou qualquer tipo de objeto definido pelos módulos do projetista.
- Vetor de Valores: O projetista pode necessitar transferir um vetor de valores ou de objetos, por isso, o evento disponibiliza um vetor genérico que suporta vetores de qualquer tipo de dados do Java, ou qualquer vetor de objetos criados pelos módulos do projetista.
- Remetente: Opcionalmente, o projetista pode fornecer o número de identificação do módulo que criou o evento. Essa característica é útil quando desejamos que o módulo que recebeu o evento tome conhecimento da origem do mesmo.

3.3.1.3 Árvore de Eventos

Como mencionado acima, o gerenciamento da fila de eventos representa a maior carga de processamento entre as funções do simulador. Para reduzir o tempo de acesso da fila de eventos, foi utilizada uma árvore binária para armazenar os eventos. Como principal vantagem, a estrutura de árvores apresenta um menor tempo médio de inserção de elementos do que uma lista ligada ou um vetor ordenado. Se a árvore possuir *n* elementos, então o número médio de comparações necessário para encontrar o lugar do elemento na árvore é de $\log_2 n$ [13]. Por exemplo, se a árvore já possuir 1.000 elementos, serão necessários, em média, 10 comparações para encontrar a posição de um novo elemento. Para uma lista encadeada, seriam necessárias em média *n/2*,ou seja, 500 comparações com nós da lista para encontrar a posição do elemento a ser inserido. O evento com a menor estampa de tempo e maior prioridade sempre está na raiz da estrutura da Árvore.

Utilizamos a estrutura de árvore da biblioteca *(TreeSet)* da linguagem Java para implementar a árvore binária no simulador. Essa estrutura apresenta um desempenho de busca superior a da lista encadeada, mas possui um campo único para a ordenação. Assim, não é possível inserir dois eventos com a mesma estampa de tempo. Para resolver esse problema, foi atribuída uma prioridade (única) para cada canal, o que determina a prioridade de execução do evento. No caso de haver eventos com um mesmo tempo de execução, o evento com maior prioridade será executado primeiro. No caso de haver eventos com a mesma estampa de tempo, direcionados a um mesmo canal e com a mesma prioridade, a árvore atribui a cada evento um valor de ordem de chegada, o que vai determinar a ordem de inserção do evento na árvore.

3.4 Conclusão

Nesse capítulo mostramos o funcionamento básico da nossa implementação de um simulador orientado a eventos seqüencial. Esse simulador foi utilizado como ponto de partida para a implementação do simulador distribuído, descrita no próximo capítulo. O simulador seqüencial pode

também ser utilizado para simular sistemas onde a realização da simulação distribuída não é possível ou desejável.

Esse simulador seqüencial foi utilizado para comparar o desempenho entre a simulação seqüencial e a distribuída para as aplicações descritas no Capítulo 5.

4. Simulação Orientada a Eventos Distribuída

Nesse capítulo, vamos primeiro descrever a metodologia de simulação distribuída conservadora, e depois apresentar a nossa implementação de um simulador destinado ao projeto de sistemas embutidos.

4.1 Introdução

A simulação orientada a eventos seqüencial é ditada pela fila de eventos, uma vez que a cada ciclo da simulação, apenas um evento é removido da fila e enviado ao módulo-destino. Nesse mesmo ciclo, novos eventos podem ser inseridos na fila em decorrência da execução do módulo ativado por esse evento.

A simulação distribuída orientada a eventos oferece uma abordagem diferente. O relógio e a fila centralizada são descartados e substituídos por uma versão local em cada computador participante na simulação [2]. Cada computador irá simular uma parte dos processos físicos do sistema. Os processos se comunicam pelo envio de eventos, que são também utilizados pelos simuladores para realizar a sincronização entre os computadores.

A simulação distribuída oferece várias vantagens além da redução do tempo de simulação do sistema. O controle da simulação é descentralizado, sendo que nenhum computador exerce a função de gerente da simulação. Assim, a simulação de um sistema pode ser adaptada de acordo com o número de computadores disponíveis, isto é, se apenas alguns computadores estão disponíveis para realizar a simulação, então um computador pode simular vários processos seqüencialmente [1].

Existem várias metodologias de simulação distribuída, sendo as principais a simulação distribuída conservadora e a simulação distribuída otimista.

A simulação distribuída apresentada nesse trabalho segue o protocolo de sincronização de Chandy/Misra/Bryant (CMB) [2][34][35], sendo portando conservadora. Depois que o sistema é descrito como um conjunto de módulo e canais de comunicação, que interligam pares desses módulos,

esse modelo é dividido em vários sub-modelos (particionamento espacial) para possibilitar o aproveitamento do paralelismo e reduzir o tempo de simulação. Esses sub-modelos são chamados de *Processos Lógicos* (*Logical Process- LP*) [2]. Cada computador participante na simulação distribuída possui o seu próprio simulador seqüencial orientado a eventos e gerenciador de simulação distribuída [1].

Na metodologia conservadora, os vários simuladores locais só podem executar os eventos considerados seguros, isto é, que não violam a restrição de causalidade local (veja seção 4.2.2). Essa metodologia evita a ocorrência de *deadlocks*, através do uso de eventos nulos (veja seção 4.2.4), sendo recomendada para simular sistemas com intensa troca de eventos entre os *LPs*. Em sistemas com baixa taxa de eventos, essa metodologia é pouco eficiente, pois os *LPs* participantes da simulação só podem avançar seus relógios internos quando receberem eventos de todos os outros *LPs*.

Outra metodologia é a simulação distribuída otimista. Essa metodologia, que não foi implementada nessa tese, difere da conservadora no fato de que permite que ocorra a violação da restrição de causalidade local, fornecendo mecanismos de recuperação. O relógio local da simulação avança até a chegada de um evento com estampa de tempo inferior ao valor do relógio local (estampa de tempo "no passado"). Quando esse evento chega, a simulação é interrompida e o simulador retorna a um estado previamente armazenado. A simulação prossegue a partir desse ponto. Essa ação é chamada de *time warp roll-back* [7] e requer que haja armazenamento de estados do sistema em vários instantes durante a simulação. Essa metodologia é aplicada para sistemas que apresentam tráfego de eventos menor do que a metodologia conservadora, pois se o tráfego de eventos entre os vários *LPs* do sistema for muito intenso, a quantidade de *roll-back* requeridas podem diminuir consideravelmente o desempenho do sistema.

4.2 Estrutura

Do ponto de vista do projetista do sistema, o funcionamento do simulador orientado a eventos distribuído é semelhante ao do simulador orientado a eventos seqüencial (veja seção 3.2). entretanto, no simulador distribuído, existem 2 tipos de eventos:

1. Eventos locais, que funcionam exatamente como os eventos do simulador seqüencial.
2. Eventos externos, que são eventos enviados entre os computadores participantes da simulação distribuída.

A comunicação lógica entre os módulos do sistema é feita através de canais, que podem ser de entrada ou saída. O canal de entrada de um módulo recebe eventos de outros módulos com os quais ele possui ligações lógicas de entrada de dados. Os canais de saída de um módulo são os canais onde um módulo envia os eventos destinados aos módulos com os quais ele possui ligações lógicas de saída de dados. Cada canal conecta apenas um par de módulos (entrada e saída).

Cada *LP* possui um horizonte de execução LVTH (*Local Virtual Time Horizon*). O LVTH é o horizonte de tempo, representa o limite de tempo para as estampas de tempo dos eventos internos e externos processados pelo simulador sem violar a restrição de causalidade local (ver seção 4.2.2 para maiores detalhes sobre a restrição de causalidade local). Para que um *LP* possa incrementar o seu horizonte de execução e consequentemente prosseguir com a simulação, ele precisa receber eventos de *todos os outros LPs* com os quais ele possui comunicação de entrada, através dos seus canais de comunicação.

Nesse modelo, em cada *LP* funciona um cliente (para enviar eventos) e um servidor (para receber eventos). A parte relacionada com o servidor de cada *LP* executa em uma *thread* concorrente, enquanto que o cliente é executado pelo programa principal.

4.2.1 Descrição dos Módulos

A Figura 4.1. mostra a estrutura básica do simulador distribuído. Os módulos que fazem parte desse simulador são descritos a seguir.

- *Buffers de entrada e de saída*: implementam os canais de comunicação de entrada e saída. São compostos por uma lista FIFO por canal, onde são armazenados os eventos externos recebidos pelo módulo.
- *Controle da simulação*: remove eventos dos buffers de entrada, insere eventos nos buffers de saída, controla o tempo de término e o relógio da simulação.

- *Simulador seqüencial*: realiza as atividades descritas no capítulo anterior.
- *Módulos da aplicação*: são os módulos do sistema simulados no computador em questão.

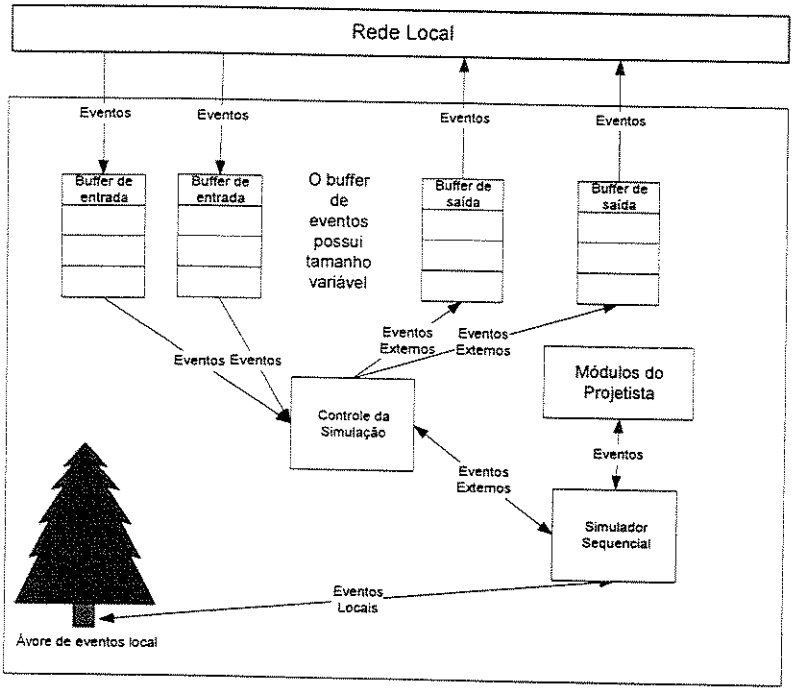


Figura 4.1. Estrutura do Simulador Distribuído.

4.2.2 Fluxo de Informações

O Algoritmo 4.1. constitui o algoritmo básico da simulação distribuída conservadora, sem mecanismos de prevenção de *deadlocks*.

Início:: LVTH:= 0 ➔ No início da simulação o horizonte é igual a 0

Enquanto o critério de término da simulação não for atingido:

{

simulação de LPi até o instante LVTH executando os passos a seguir:

1. *Espera até receber eventos em todos os canais (buffers) de entrada*
2. *Receber os eventos em todos os canais de entrada, armazena em LVTH a menor estampa de tempo dos eventos contidos nos canais de entrada*

3. *Execução dos eventos locais que possuem estampa de tempo menor ou igual a LVTH:*
4. *Execução dos eventos de todos os canais externos de entrada que possuem estampa de tempo igual a LVTH:*
5. *Armazenamento na fila local os eventos destinados aos módulos locais*
6. *Envio pelos canais de comunicação externos os eventos destinados aos outros LPs*

}

Algoritmo 4.1. Simulação distribuída.

Para manter a ordem de execução entre os eventos locais e externos, os eventos locais são executados primeiro, pois os mesmos podem possuir estampas de tempo menor que o LVTH. Eventos externos sempre tem a estampa de tempo igual ao LVTH (veja a linha 2. no algoritmo), logo, são executados depois dos eventos locais.

4.2.3 *Deadlocks* e Restrição de Causalidade Local

Deadlock e a violação da restrição de causalidade local são os principais problemas da simulação distribuída.

4.2.3.1 *Deadlock*

Deadlock é a situação na qual nenhum processo do sistema consegue executar e causa o bloqueio de toda a execução do sistema.

Um conjunto de *LPs* entra em deadlock quando *uma das* condições abaixo é satisfeita [2][19]:

- *Todos* os LPs estão esperando para receber um evento ou terminaram de executar.
- *Pelo menos um LPi* está esperando para receber um evento de um *LPj* que precisa receber um evento de *LPi* para executar.

- Os *LPs* estão dispostos de forma cíclica, onde cada *LP* está esperando por um evento de seu antecessor para prosseguir com a execução.

Pelas condições acima, é possível concluir que nenhum *LP* irá realizar nenhum tipo de computação, uma vez que cada *LP* estará esperando pela chegada de eventos.

Na Figura 4.2, temos um exemplo de deadlock. O número em cada canal é o tempo do último evento recebido pelo LP, isto é, o último evento recebido pelo *LP y* tem a estampa de tempo 20 (*TEv*, na figura), e assim por diante. Nenhum dos *LPs x*, *y* e *z* podem executar e enviar eventos a não ser que recebam um evento em cada canal de entrada, para evitar erros de causalidade local.

Podemos ver que *z* não vai enviar um evento a não ser que *x* envie um evento para *y*. Analisando a figura, podemos perceber que *x* não precisa esperar por *z* para enviar o evento para *y*; *x* poderia processar o evento proveniente da *Origem*. Porém, em nenhum dos *LP x*, *y*, e *z* essa informação está disponível. Eles só possuem o conhecimento do comportamento do que está ocorrendo no computador local. Portanto, *x* não pode prosseguir enquanto não receber eventos de *z*, *z* não pode prosseguir enquanto não receber eventos de *y* e *y* não pode prosseguir enquanto não receber eventos de *x*, levando a um deadlock.

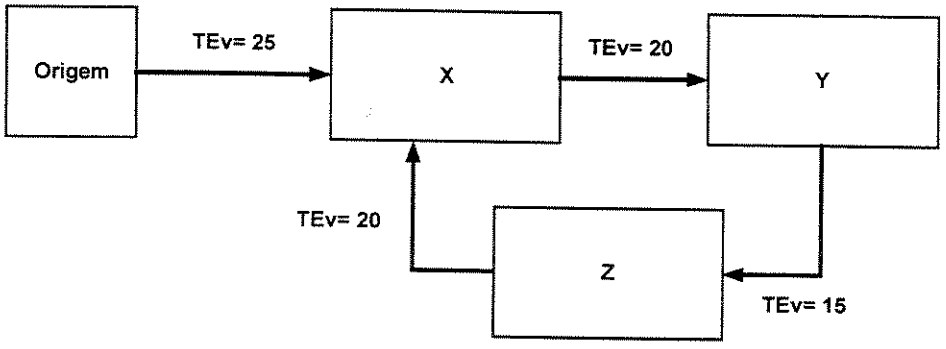


Figura 4.2 Simulação Distribuída com deadlock.

4.2.3.2 Restrição de Causalidade Local

Além do deadlock, outro problema da simulação distribuída conservadora é garantir que não hajam violações da causalidade local na simulação [1].

Para obedecer a restrição de causalidade local, exige-se que num sistema distribuído todos os eventos de todos os *LPs* executem em ordem não decrescente de estampa de tempo. Se a violação dessa restrição ocorrer, um *LP* pode avançar seu relógio interno para um valor *t* e depois receber um evento externo com um valor *t-1* ou menor. Essa violação de causalidade local faz com que os eventos sejam executados fora da sua ordem cronológica, o que causa erros na simulação.

4.2.4 Prevenção de Deadlocks e Sincronização com Eventos Nulos

A simulação distribuída apresentada nesse trabalho utiliza o protocolo de sincronização de Chandy/Misra/Bryant (CMB) [2][34]. Nesse protocolo, os vários simuladores locais só podem executar os eventos considerados seguros. Eventos seguros são aqueles que não violam a restrição de causalidade local (veja a seção 4.2.2), isto é, são executados na ordem *cronológica* não decrescente de estampa de tempo.

Essa metodologia procura evitar o deadlock, ao invés de realizar operações para a recuperação do mesmo, como é feito na simulação distribuída otimista. A sincronização entre os diversos simuladores do sistema é conseguida com a propagação de eventos nulos. Esses eventos não carregam nenhuma informação, apenas indicam ao *LP* que recebe o evento nulo que o *LP* que o enviou não enviará nenhum outro evento com estampa de tempo menor do que a desse evento nulo.

A utilidade do evento nulo é possibilitar ao *LP* que recebe esse evento nulo avançar o seu relógio local, para poder sair de uma situação de espera. Os eventos nulos não são representados no modelo do sistema, são apenas um mecanismo interno do simulador gerado automaticamente durante a simulação. O uso de eventos nulos evita o deadlock e garante o sincronismo temporal entre os *LPs* do sistema.

Ao receber eventos nos canais de entrada do *LP*, o LVTH será igual ao tempo do evento que possui a menor estampa de tempo. O simulador remove esse evento do canal de execução externo, remove todos os eventos nulos que estiverem no topo das filas dos outros canais externos e executa eventos de sua fila local até o valor do LVTH. O tempo do último evento da fila local é armazenado no LVT (*Local Virtual Time*).

Para melhorar o desempenho do simulador, apenas um evento nulo é armazenado de cada vez nos *buffers* dos canais de entrada. Quando um evento nulo vai ser inserido no *buffer*, o *LP* testa se já há outro evento nulo no *buffer*. Se houver, esse evento é removido, ficando no *buffer* só o evento nulo com a maior estampa de tempo. Ao remover esses nulos do *buffer*, diminui-se o tráfego de eventos nulos entre *LPs*, o que evita o desperdício de ciclos de simulação com o processamento de eventos nulos.

Depois de executados, os eventos internos e externos, o simulador computa o *lookahead*. O *lookahead* é uma previsão de tempo para a qual o simulador não envia eventos. O *lookahead* é computado somando o tempo do LVTH e o tempo do menor caminho formado pelas linhas de execução do grafo dos módulos do simulador.

A Figura 4.3. mostra um exemplo do cálculo do *lookahead*. Nessa figura, os atrasos ao longo das duas linhas de execução (soma de todos os atrasos dos canais de comunicação mais o tempo de execução dos módulos), *L1* e *L2* é 12 para a linha *L1*, que é composta pelos canais *c1,c2* e *c3*. Para a linha de execução *L2* que é composta pelos canais *c4* e *c5*, o atraso é igual a 10. Portanto, o *lookahead* é 10, igual ao atraso da linha de execução *L1*. O *lookahead* tem o propósito avançar o relógio da simulação, sendo propagado pelos eventos nulos. Quanto maior for o tempo do evento nulo enviado aos outros canais, maior a chance de um canal ser desbloqueado e ter um maior número de seus eventos internos executados.

O simulador então envia os eventos externos existentes, enviando também eventos nulos com estampa de tempo igual LVTH+ *lookahead* para todos os canais de saída. O envio desses eventos nulos possibilita ao LP que recebe esse evento avançar o seu LVTH.

Só são enviados eventos externos com estampas de tempo menor ou igual ao LVTH+ *lookahead*. Eventos externos que possuem estampas de tempos superiores ao LVTH+ *lookahead* são armazenados num *buffer* de saída, para não violar a restrição de causalidade local.

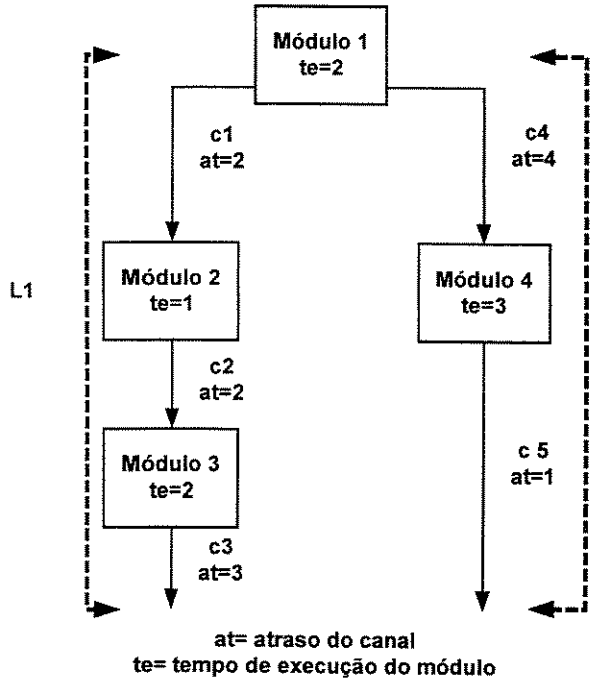


Figura 4.3 Exemplo de cálculo do *lookahead.*

Exemplo: para um *LP* com dois módulos, mostrados na Figura 4.4, onde o módulo 1 possui tempo de execução igual a 2, enquanto que o módulo 2 possui tempo de execução igual a 10. O *lookahead* desse *LP* será 2, pois nesse caso, é o menor tempo de execução entre os dois módulos. Ambos os módulos recebem eventos da *Origem.*

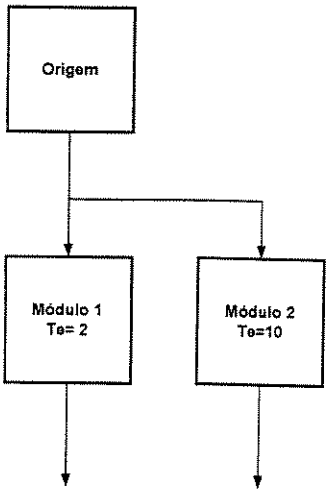


Figura 4.4. Conexão entre os módulos.

A Tabela 4.1 mostra as estampas de tempo dos eventos recebidos e enviados pelos dois módulos mais o *LVTH+ lookahead* correspondente durante 3 ciclos de simulação. Caso fossem enviados eventos dos dois módulo no término do primeiro ciclo de simulação, um evento com a estampa de tempo 4 e o outro com estampa 12, no segundo ciclo da simulação ocorreria uma violação de causalidade, pois o módulo 1 gera um evento com estampa de tempo 5, enquanto que no primeiro ciclo foi enviado o evento do módulo 2 com estampa de tempo 12 do módulo 2. Por essa razão que são enviados apenas eventos externos com estampa de tempo menor ou igual a *LVTH+ lookahead.*

Ciclos de simulação	**Eventos recebidos pelos módulos**	**Eventos enviados pelo Módulo 1**	**Eventos Enviados pelo Módulo 2**	**LVTH+ lookahead**
1	2	4	12	4
2	3	5	13	5
3	5	7	15	7

Tabela 4.1. Exemplo de envio de eventos

Apesar do simulador nunca entrar em deadlock, a aplicação que faz uso do simulador pode entrar em deadlock. Nesse caso, o simulador continua a trocar eventos nulos até o término da simulação. O

projetista pode rastrear os eventos e tentar localizar a causa do *deadlock* no sistema em simulação. Note que os eventos nulos não são armazenados no arquivo de rastreamento de eventos.

4.2.4.1 Exemplo de Troca de Eventos entre Computadores

Vamos apresentar nessa seção um exemplo do fluxo de eventos nulos entre 4 computadores durante 4 cenários de simulação, seguindo as ilustrações na Figura 4.5.

A figura mostra as conexões entre os computadores como vértices orientados. Em cada vértice existe um *OutEvn*, que indica o evento nulo gerado pelo computador nos seus canais de saída, e um *InEvn*, que indica o evento nulo recebido pelo computador nos seus canais de entrada. O *OutEvn* terá uma estampa de tempo igual a do *InEvn* recebido mais o *lookahead* do computador. O *lookahead* de cada computador é indicado por *LA* na figura. Note que o *OutEvn* de um módulo num cenário será o *InEvn* do seu sucessor no próximo cenário.

No primeiro cenário, os computadores enviam eventos nulos (*OutEvn*) com estampa de tempo igual ao tempo de execução do computador (*lookahead*), para todos os seus canais de saída. No segundo cenário da simulação, podemos perceber que os computadores somam a estampa de tempo do evento recebido (*InEvn*) com o *lookahead*.

O computador 4 recebe 2 eventos, um com estampa de tempo 1 e o outro com estampa de tempo 2. O computador vai considerar para o cálculo do seu próximo evento nulo o evento recebido com a menor estampa de tempo dentre os dois eventos recebidos, enviando então um evento nulo com estampa de tempo igual a 4 para o computador 1 (1 da estampa de tempo do evento recebido mais 3 do *lookahead* do computador 4). O evento nulo recebido pelo computador 4, que foi enviado pelo computador 3, é descartado. O LVTH de cada computador vai ser igual ao tempo do último evento nulo recebido. Note como as estampas de tempos dos eventos sempre avançam, pois a metodologia não permite a geração de eventos com estampas de tempos inferiores aos já enviados.

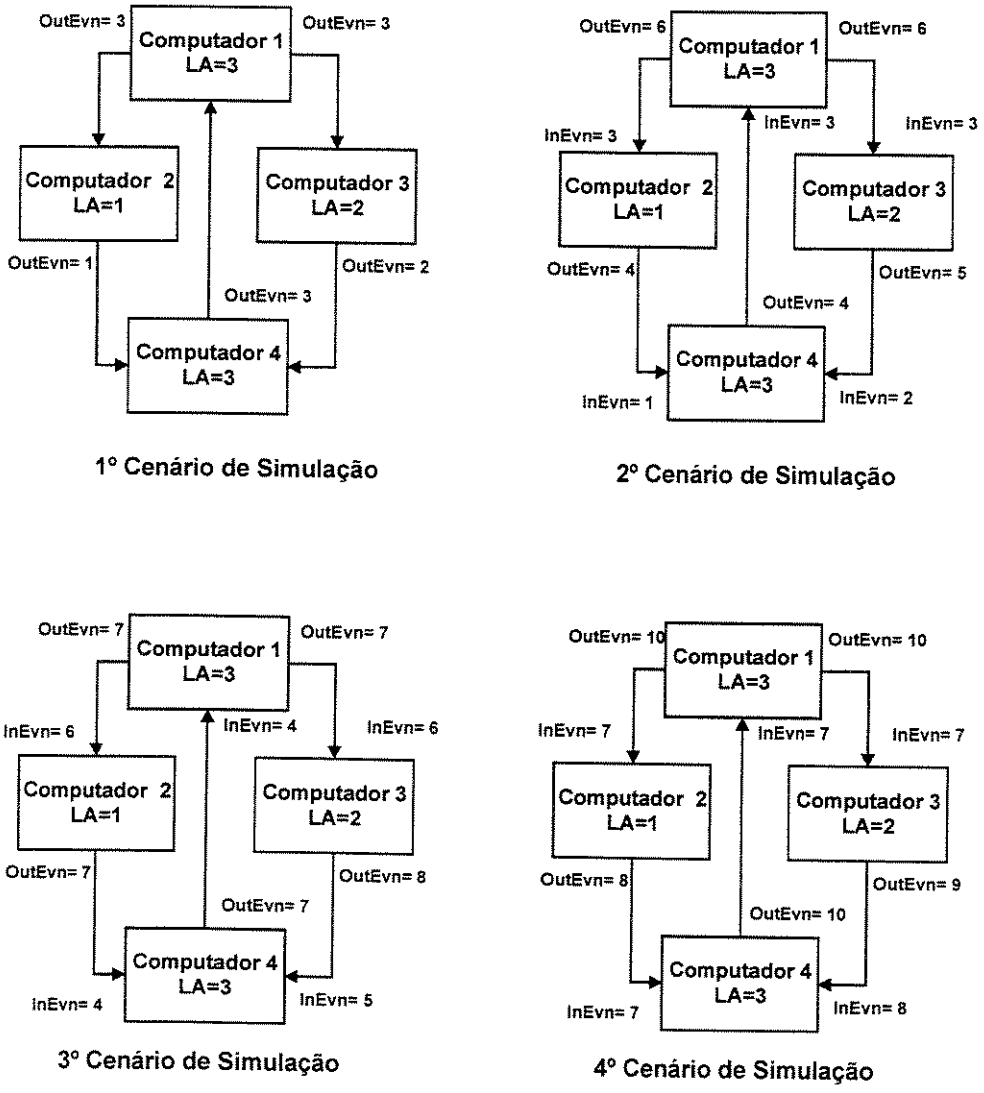


Figura 4.5. Fluxo de eventos nulos entre os módulos.

É importante ressaltar que a ordem do envio e recebimento de eventos pode ser alterada de acordo com a velocidade, tráfego da rede, velocidade de processamento do computador, percentual da

capacidade de processamento do computador dedicada ao simulador e a velocidade de acesso dos discos rígidos dos computadores.

4.3 Implementação

A simulação distribuída foi feita com o auxílio das classes RMI (*Remote Method Invocation*), um conjunto de classes da linguagem Java que facilitam a criação de programas distribuídos [13]. As classes RMI permitem que um programa tenha acesso a um objeto criado em um outro computador. Nesse trabalho, as classes RMI foram utilizadas para implementar os *buffers* de entrada dos canais de comunicação, que são organizados em listas encadeadas. Os *buffers* são, efetivamente, a única parte do simulador compartilhada entre os computadores participantes na simulação. Cada simulador possui um servidor e um cliente. O servidor funciona numa *thread,* e gerencia as inserções de eventos nos seus *buffers* de entrada realizadas pelos outros *LPs* do sistema. Como cliente, o simulador estabelece uma conexão com os servidores dos outros *LPs* para possibilitar o envio de eventos para esses *LPs*.

A estrutura do simulador distribuído é mostrada na Figura 4.6. Inicialmente, o simulador inicializa todos os seus objetos, inicializa o cliente e ativa o servidor local. Depois que todos os servidores e clientes são ativados, o objeto *Kernel Distribuído* irá gerenciar a simulação. A primeira tarefa realizada é enviar eventos nulos para todos os canais de saída do *LP*. Esses eventos são enviados com o valor de *lookahead.* Esse tempo é enviado porque indica que esse *LP* não vai gerar nenhum evento com estampa de tempo menor que o *lookahed.* Em seguida, o método consulta seus buffers de entrada.

O menor tempo do evento contido nos *buffers* de entrada será o tempo do LVTH. São executados todos os eventos internos com tempo igual ou menor que o LVTH. Caso a execução desses eventos gerem eventos externos, os mesmos são encaminhados para os *buffers* de saída. Eventos internos gerados são armazenados na árvore local.

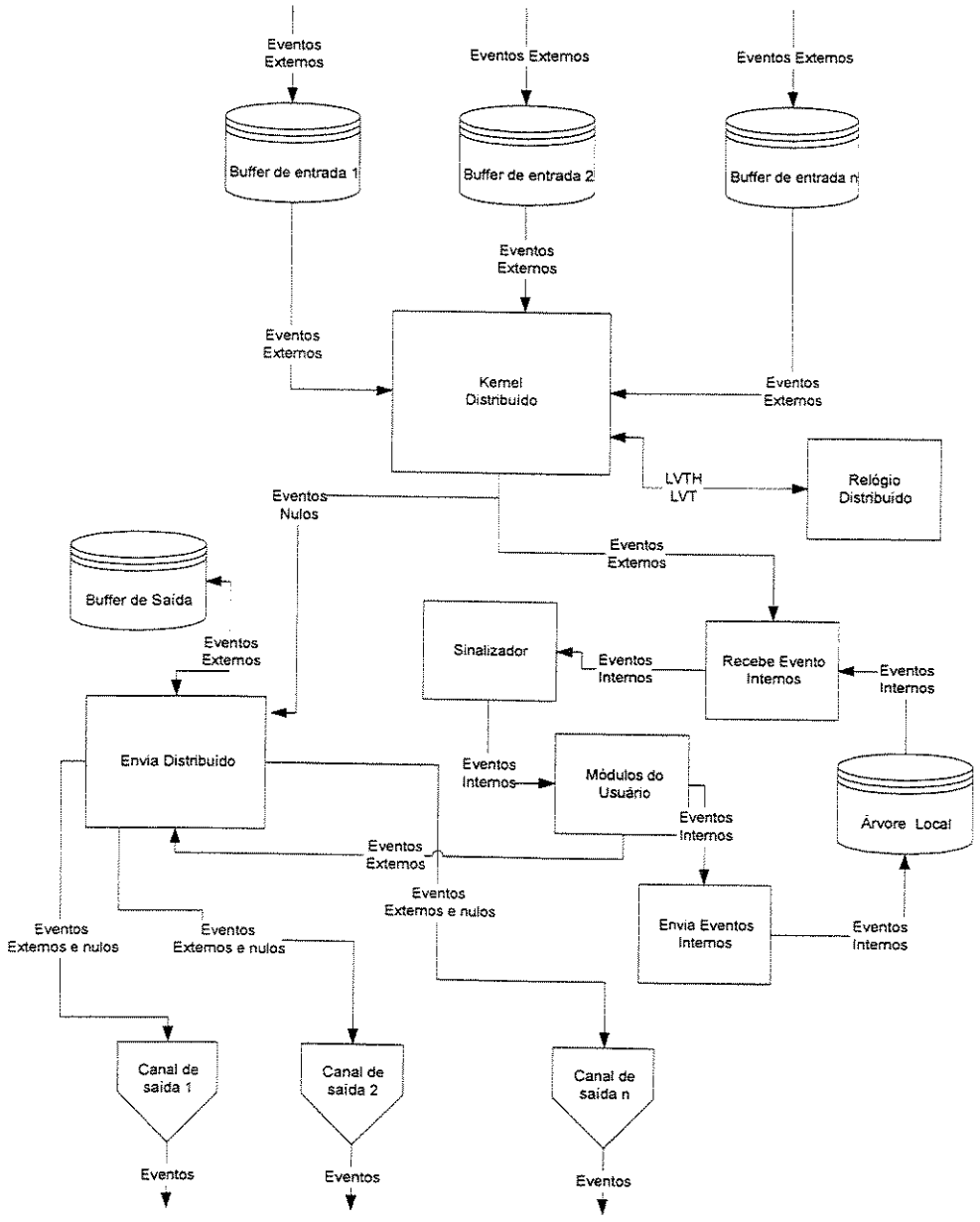


Figura 4.6. Estrutura do Simulador Distribuído.

A seguir são executados os eventos externos com estampa de tempo iguais ao LVTH. Os eventos nulos contidos no topo dos b*uffers* de todos os canais são removidos. A seguir, são enviados os eventos externos com tempo menor ou igual a LVTH +*lookahead.* Caso haja eventos externos a serem enviados com tempo maior que LVTH +*lookahead*, os mesmos são armazenados num *buffer* de saída,

para não violar a restrição de causalidade local. São também enviados eventos nulos com tempo LVTH +*lookahead* para todos os canais, para atualizar os valores dos LVTH nos demais *LPs* do sistema. Esse ciclo se repete até que não haja mais eventos a serem executados ou até que a condição de término seja atingida (tempo máximo de simulação).

A execução de eventos interna e externa segue as mesmas etapas que no simulador seqüencial. A classe *Recebe Eventos* envia o evento para o objeto *Sinalizador* correspondente. Esse objeto armazena esse evento e o envia para o módulo da aplicação (veja a seção 3.3.1.1.1). Caso os módulos da aplicação gerem algum evento interno, os mesmos são colocados na árvore de eventos. Caso gerem eventos externos, os mesmos são tratados pela classe *Envia Evento Distribuído.*

Os eventos externos com estampa de tempo menor ou igual a *LVTH+ Lookahead* são enviados para os destinatários enquanto que os eventos com tempo maior que *LVTH+ Lookahead* são armazenados no buffer de saída, como explicado na seção 4.2.4.

4.4 Comentários Finais

Nessa seção comentaremos sobre algumas técnicas utilizadas na implementação do simulador descrito nesse capítulo.

- O simulador permite ao projetista uma grande flexibilidade na programação de seus módulos, uma vez que são poucas as alterações necessárias para que os módulos da aplicação utilizem o simulador.
- O simulador inicializa os objetos da aplicação, bastando apenas ao projetista criar os arquivos de parâmetros necessários para a descrição do sistema para o simulador.
- Caso, durante a simulação do sistema, os módulos implementados pelo projetista gerem algum tipo de exceção (erro fatal), o simulador trata essas exceções (caso o projetista não o tenha feito) emitindo na tela o tipo de erro ocorrido. Esse tratamento de exceção não interrompe a simulação, sendo que a mesma termina normalmente, emitindo o relatório de simulação. Com isso, o projetista pode verificar o tipo de erro ocorrido e analisar o arquivo de rastreamento dos eventos para tentar identificar a origem do erro.

- Na simulação distribuída, um canal de entrada pode receber vários eventos com a mesma estampa de tempo. Pelo algoritmo original, cada um desses eventos é retirado em um ciclo da simulação, o que ocasiona um grande desperdício de processamento, uma vez que em cada ciclo é necessário enviar eventos nulos para todas os canais de saída do *LP*. Para minimizar esse problema, nós simplesmente realizamos um rastreamento em todos os canais de entrada para poder executar (no caso de eventos) ou remover (no caso de eventos nulos) todos eventos com a mesma estampa de tempo. A realização desse procedimento, aliado com a remoção dos eventos nulos dos *buffers* no momento da inserção de novos eventos, diminui o tráfego de eventos nulos entre os computadores e reduz o tempo da simulação.

5. Simulação Distribuída Aplicada a Projetos

Nesse capítulo vamos mostrar dois exemplos de aplicações simuladas com o nosso simulador distribuído. O primeiro exemplo é uma implementação do algoritmo de Barnes-Hut. Esse algoritmo foi escolhido para mostrar o uso do simulador em uma aplicação com alta carga computacional e de baixa taxa de comunicação entre os módulos. No segundo exemplo, o sistema simulado é uma rede simplificada de comutadores ATM. Esse exemplo foi escolhido para exemplificar o projeto de um componente, o comutador ATM e para avaliar o comportamento do simulador em uma situação onde o tráfego de mensagens entre os módulos da aplicação é intenso.

5.1 Exemplo 1: Algoritmo de Barnes-Hut

Esse problema foi escolhido por possuir um elevado custo computacional e ser bastante utilizado na comparação de desempenho de sistemas paralelos [39].

O problema do cálculo de força gravitacional exercida entre corpos tem aplicações em áreas como mecânica celestial, física de plasma e dinâmica molecular. A maneira clássica de resolver o problema é realizar o cálculo das forças atuando nos corpos é calcular a força exercida sobre um corpo por cada um dos demais corpos e encontrar a força resultante. Essa resolução tem a complexidade de $O(n^2)$, onde n é o número de corpos. Devido ao cálculo de forças entre todos os pares de pontos, essa resolução é ineficiente quando n é muito grande.

Vários algoritmos foram propostos para realizar esse cálculo, entre esses, o algoritmo de Barnes-Hut [36][37][38].

5.1.1 Aplicação

O algoritmo de Barnes-Hut adota a estratégia de dividir para conquistar. Esse algoritmo realiza o cálculo da força gravitacional e da trajetória de corpos sob o efeito dessa força. Como a força gravitacional diminui na taxa de $1/r^2$,

dos corpos. Esses passos são repetidos até que o número de iterações desejado seja atingido. Cada iteração do algoritmo de Barnes-Hut calcula as forças resultantes e as posições ocupadas pelos corpos supondo um intervalo de tempo de duração *h*, em relação a última iteração.

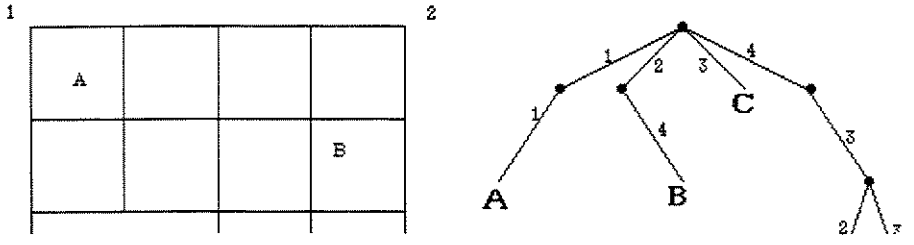
5.1.1.1 Armazenamento das Informações

A Figura 5.2. mostra corpos posicionados em um espaço bidimensional, e a árvore quaternária associada. Em cada nó estão armazenadas as coordenadas dos 4 nós, descendentes gerados pela subdivisão do nó. Quando um corpo é inserido, o algoritmo verifica em qual das folhas o corpo deve ser inserido. Caso tenha um outro corpo nessa folha, a mesma é transformada em nó e os corpos são alocados como folhas desse novo nó. O algoritmo atua recursivamente até que exista um único corpo em cada extremidade da árvore.

O Algoritmo 5.1. descreve a inserção de corpos na árvore:

1. Descobrir o tamanho do espaço, calculando um quadrado cujas extremidades são as maiores e menores coordenadas entre os corpos.
2. Dividir o espaço em 4 áreas iguais, colocando numa árvore quaternária, onde na raiz é considerado o espaço total e cada nó pode conter uma sub-árvore ou um corpo.
3. Se uma área possuir mais de um corpo, repetir 2 até que haja apenas um corpo por folha.

Algoritmo 5.1. Algoritmo de criação da árvore.



5.1.1.2 Cálculo do Centro de Massa e da Massa Total

O cálculo dos centros de massa para os nós da árvore é realizado logo em seguida à criação da árvore. Esses valores são utilizados para calcular a força quando a distância do corpo em relação ao centro de massa de um conjunto de corpos for muito grande. O algoritmo começa nas folhas da árvore e prossegue até chegar a raiz da árvore calculando as coordenadas e a massa do centro de massa de cada nó, a partir dos valores calculados para seus descendentes. Para realizar o cálculo do centro de massa, é necessário percorrer toda a árvore recursivamente, seguindo o Algoritmo 5.2:

1. Como nas folhas da árvore existe apenas um corpo, as coordenadas e a massa do centro de massa da folha correspondem à posição e a massa desse corpo.
2. Para cada nó *A* cujos descendentes tem coordenadas e massa do centro de massa conhecidos, o algoritmo calcula o valor das coordenadas *Xcm(A)*, *Ycm(A)* e a massa do centro de massa *Mcm(A)* de *A*, utilizando as equações 5.1, 5.2 e 5.3.

Algoritmo 5.2. Cálculo de massa.

$$Xcm(i) = \frac{\sum_{di=4}^{di=1} Xcm(di) * mass(di)}{\sum_{di=4}^{di=1} mass(di)}$$

Equação 5.1. Cálculo da coordenada *x* do centro de massa.

$$Ycm(i) = \frac{\sum_{di=4}^{di=1} Ycm(di) * mass(di)}{\sum_{di=4}^{di=1} mass(di)}$$

Equação 5.2. Cálculo da coordenada *y* do centro de massa.

$$Mcm(i) = \sum_{di=4}^{di=1} Mcm(di)$$

Equação 5.3 Cálculo da massa do centro de massa.

Xcm(i) coordenada *x* do centro de massa para os corpos na sub-árvore com raiz no nó *i*.

Ycm(i) coordenada *y* do centro de massa para os corpos na sub-árvore com raiz no nó *i*.

Mcm(i) Massa do centro de massa para os corpos numa sub-árvore com raiz no nó *i*.

mass massa dos corpos numa sub-árvore.

i nó no qual está sendo calculado o centro de massa.

di nós descendentes do nó *i*.

5.1.1.3 Percorrendo a Árvore

O Algoritmo 5.3. descrito abaixo é utilizado para percorrer a árvore durante o cálculo da força gravitacional atuando em cada corpo.

1. Começando do topo da árvore, comparar o quociente *d/r* e θ para o corpo *Ci* (veja a seção 5.1.2.2 sobre θ).
2. Se *d/r* for menor que θ, calcular a força usando o centro de massa do nó.
3. Caso contrário, descer mais um nível da árvore, e repetir 2.
4. Repetir o procedimento até que a força resultante atuando em todos os corpos tenha sido calculada.

Algoritmo 5.3. Algoritmo para percorrer a árvore.

Ao percorrer a árvore durante o cálculo da força gravitacional atuando sobre um corpo, é importante identificar a folha associada ao próprio corpo, para que não seja calculada a força gravitacional de um corpo sobre ele mesmo.

5.1.1.4 Calculando as Forças

A força gravitacional atuando num corpo localizado no ponto (x, y) é igual ao somatório das forças exercidas por todos os outros corpos no universo considerado. O algoritmo percorre a árvore até que o ponto em que a razão *d/r* seja menor que θ. Os cálculos são repetidos até que todos os corpos tenha sido levados em conta. O cálculo da força é dado pela Equação 5.4:

$$Força(x,y)_n = G * m * Mcm * \left(\frac{Xcm - x}{r^3}, \frac{Ycm - y}{r^3} \right)$$

que fornece os componentes da força gravitacional atuando sobre o corpo localizado na posição (x, y) devido aos corpos situados numa sub-árvore.

onde $r = \sqrt{\left((Xcm - x)^2 + (Ycm - y)^2\right)}$

Equação 5.4. Cálculo da força atuando sobre um corpo no instante de tempo *n*.

Onde:

m é a massa do corpo

$(x,y)_n$ são as coordenadas do corpo no instante de tempo $n*h$

(Xcm, Ycm) são as coordenadas do centro de massa dos corpos localizados na sub-árvore.

Mcm é a massa total dos corpos na sub-árvore.

G é a constante gravitacional

5.1.1.5 Reposicionamento dos Corpos

Depois de calcular a força gravitacional resultante em cada corpo, é necessário atualizar a posição dos corpos no espaço, de acordo com os deslocamentos provocados pela força gravitacional. O reposicionamento dos corpos é feito com o uso das Equações abaixo [36]:

$$V(x,y)_{n+1/2} = V(x,y)_n + a_n\left(\frac{h}{2}\right)$$

Equação 5.5. Cálculo da velocidade do corpo instante *(n+1/2)*h*.

$$r(x,y)_{n+1} = r(x,y)_n + h * V(x,y)_n$$

Equação 5.6. Cálculo da nova posição do corpo.

Depois que a posição do corpo é atualizada pela Equação 5.6, é realizado o cálculo da velocidade final do corpo (Equação 5.7), para ser utilizado na próxima iteração do algoritmo, onde a mesma será a velocidade inicial.

$$V(x,y)_{n+1} = V(x,y)_{n+1/2} + a_{n+1}\left(\frac{h}{2}\right)$$

Equação 5.7. Cálculo da velocidade do corpo no instante *(n*h)*.

onde:

$v_{i+1/2}$ é a velocidade do corpo no instante $i*h+h/2$.

v_i é a velocidade do corpo no instante $i*h$.

r_i é a posição do corpo no instante $i*h$, $r_n = (x_n, y_n)$.

a_i é a aceleração do corpo no instante $i*h$.

h é o intervalo de tempo correspondente a uma iteração do algoritmo.

O cálculo da velocidade no instante $(n+1/2)*h$, ou seja, entre duas iterações do algoritmo de cálculo de força, visa melhorar a precisão dos resultados.

5.1.2 Implementação

Nessa seção explicaremos o fluxo do programa, detalharemos as classes utilizadas na implementação do algoritmo de Barnes-Hut e depois mostraremos os parâmetros utilizados na simulação desse exemplo.

O algoritmo foi implementado em Java, podendo ser executado de forma distribuída ou local. Nossa implementação do algoritmo de Barnes-Hut distribui a execução do algoritmo de uma maneira

simples, que reduz o tempo de execução do problema em relação a execução seqüencial do mesmo, apesar de ser menos eficiente do que a solução apresentada em [40].

5.1.2.1 Fluxo do Programa

A distribuição dos módulos é mostrada na Figura 5.3. Na nossa implementação, os corpos são organizados numa árvore quaternária. No final da inserção dos corpos na árvore, feitas pelo *Nó central*, cada sub-árvore da raiz é associada a um computador. Cada computador calcula as forças resultantes de seus corpos em relação as suas sub-árvores locais. Terminado o cálculo de força local, cada computador envia seus corpos aos outros computadores para calcular a força nas sub-árvores associadas a esses computadores. Em seguida, cada computador calcula a nova posição dos corpos, e envia uma mensagem para todos os demais computadores indicando o encerramento dos cálculos, com a finalidade de permitir que o algoritmo realize uma nova iteração.

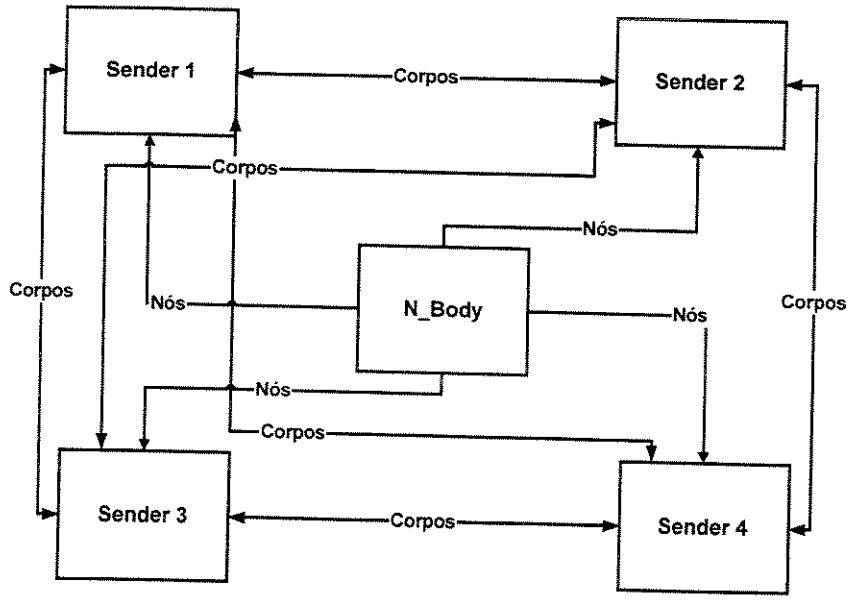


Figura 5.3 Módulos da implementação do algoritmo de Barnes-Hut.

A cada nova iteração, é feita uma avaliação do deslocamento dos corpos e calculando o *fator de distanciamento FD* (Equação 5.8)para verificar o quanto o lado do quadrado de uma região aumentou ou diminuiu em relação a iteração anterior do algoritmo.

$$FD_{n+1} = \frac{Medida\_do\_lado\_do\_quadrado_{n+1}}{Medida\_do\_lado\_do\_quadrado_{n}}$$

Equação 5.8. Cálculo do Fator de distanciamento dos corpos.

O *fator de distanciamento* tem a função de indicar quando é necessário realizar uma reconfiguração, isto é, mudar os corpos que constituem cada uma das sub-árvores. Quando os corpos se distanciam em demasia, as regiões acabam ficando com uma largura do quadrado *d* muito grande, fazendo com que as regiões sejam superpostas aumentando a probabilidade com que o cálculo de *d/r* entre o corpo e a região seja menor que θ, exigindo que o algoritmo calcule a força resultante usando sub-árvores mais próximas das folhas e, conseqüentemente, aumentando o tempo de execução do algoritmo.

Se o fator de distanciamento for maior que 5 ou menor que 0,06 então o nó em questão poderia ser beneficiado por uma reconfiguração. O limite superior quantifica o quanto o lado do quadrado da região da sub-árvore correspondente ao nó aumentou, e o limite inferior quantifica o quanto o lado do quadrado da região diminuiu. Os valores acima foram determinados através de testes práticos durante a simulação, com o objetivo de manter o número de reconfigurações num patamar aceitável.

Aumentando o limite superior, ou reduzindo o limite inferior reduz-se o número de reconfigurações. Se esse limite for menor, seriam feitas mais reconfigurações.

A justificativa para o limite superior é que, se corpos de uma região se distanciam em demasia, aumenta-se o tamanho da região. Isso faz o algoritmo descer mais níveis da árvore de corpos, aumentando o tempo da execução. Por essa razão, uma reconfiguração diminui o tempo de execução do algoritmo. Caso as regiões não tenham alterado muito o seu tamanho, não é necessário realizar a reconfiguração.

É também adotado um limite inferior de 0.06 para o *fator de distaciamento FD.* Nesse caso, é provável que, com uma reconfiguração, mais corpos possam ser incluídos na sub-árvore

correspondente a esse nó, tornando as próximas iterações do algoritmo de cálculo de força mais rápidas.

A reconfiguração consiste em inserir todos os corpos em uma nova árvore levando em conta as novas posições dos mesmos, e é realizada pelo *Nó central*. As 4 sub-árvores dessa árvore são redistribuídas entre os computadores que participam da simulação. Essa operação é executada seqüencialmente, tem complexidade *O(n)* e pode tornar o algoritmo muito lento, dependendo da quantidade de corpos utilizados na simulação.

A Figura 5.4. demonstra o que ocorre durante a execução do algoritmo. A Figura 5.4.A. mostra a região no início do cálculo da força. A Figura 5.4.B. mostra a nova posição dos corpos logo em seguida ao cálculo da força. Podemos notar que alguns corpos já saíram fora da região inicial. Na Figura 5.4.C, o nosso algoritmo recalcula o valor dos lados dos quadrados da sub-regiões. Podemos notar que há uma superposição de sub-regiões. A Figura 5.4.D. mostra a região resultante após uma operação de reconfiguração. Podemos notar que houve um corpo que mudou de sub-região.

A Figura 5.5. mostra o fluxo do algoritmo implementado. Um computador pede a reconfiguração das regiões caso haja em sua sub-árvore pelo menos 3 nós cujos fatores de distanciamento forem maiores que 5 ou menores que 0,06. Pelo menos 2 computadores tem que pedir a reconfiguração para que uma nova árvore seja feita. Note que enquanto a reconfiguração é realizada, todos os outros computadores ficam ociosos a espera de suas novas sub-árvores. Todos os computadores têm que finalizar a iteração corrente antes de prosseguir para a próxima, para manter as informações relativas aos corpos sempre atualizadas. O algoritmo prossegue até que o número desejado de iterações seja atingido.

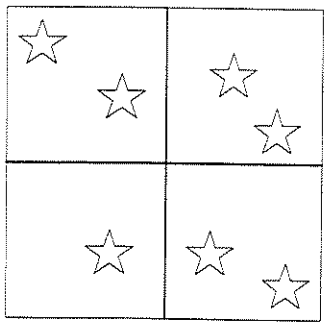


Figura A

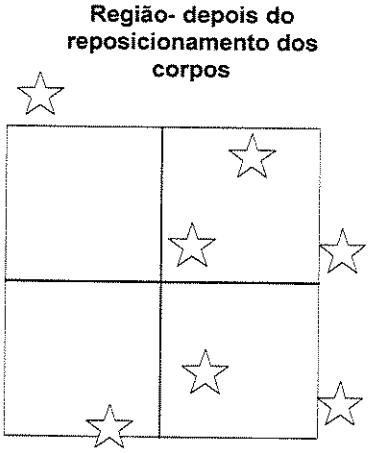


Figura B

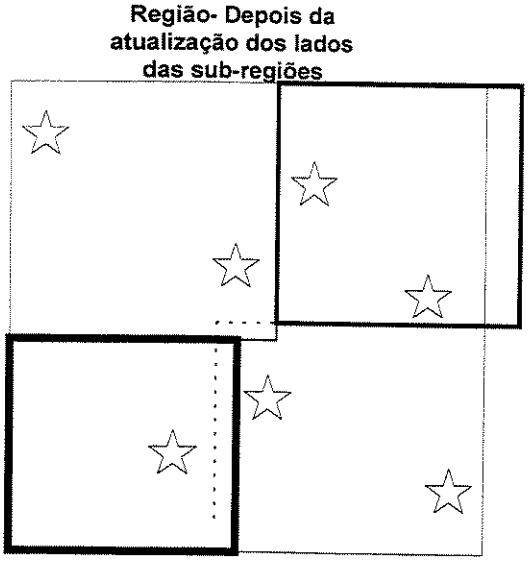


Figura C

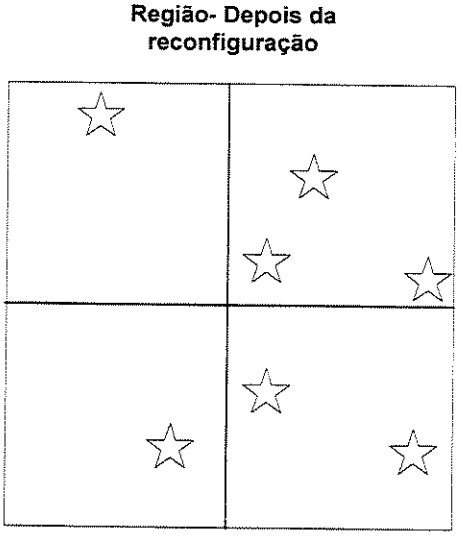


Figura D

Figura 5.4. Efeito da Reconfiguração.

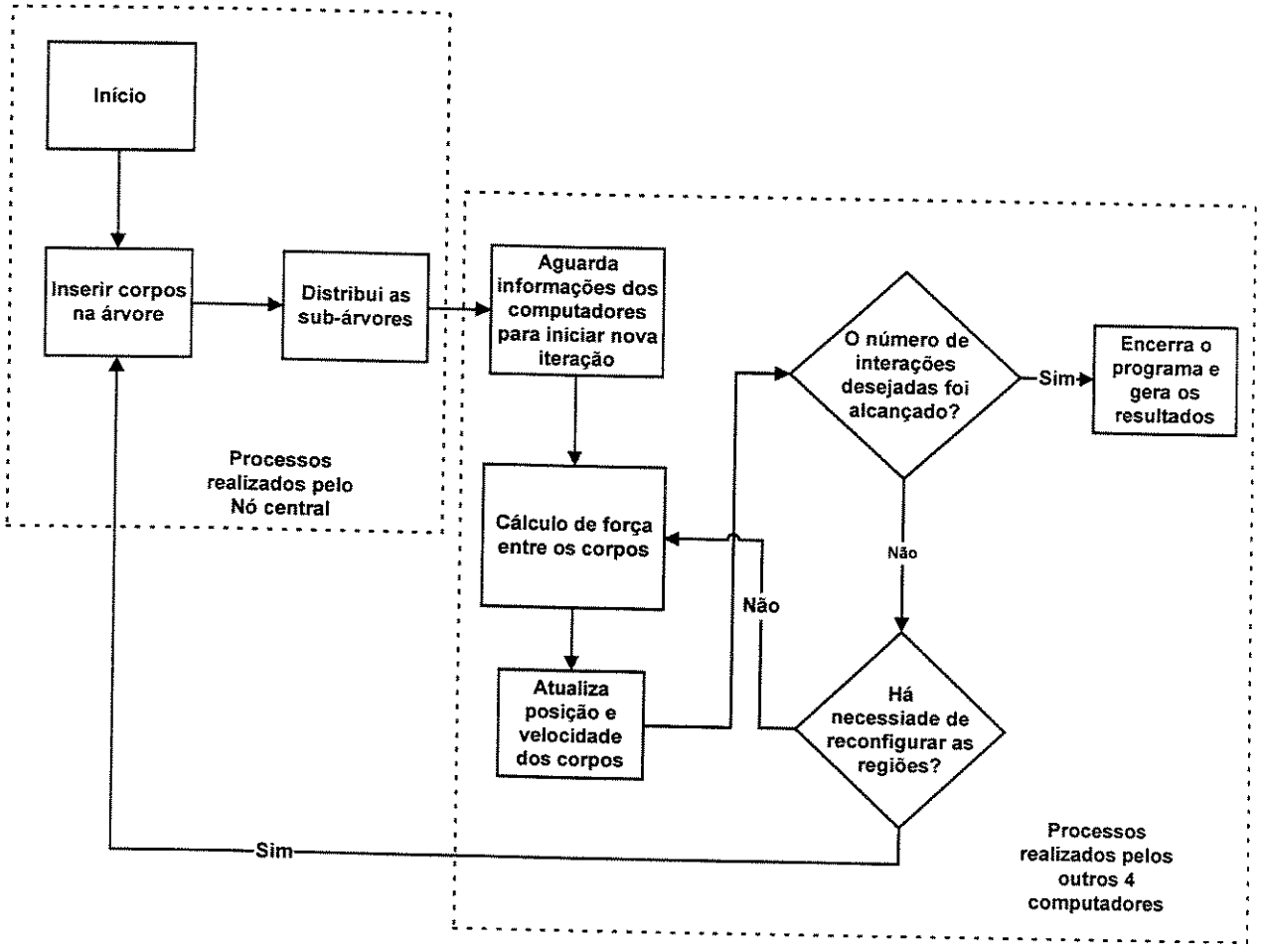


Figura 5.5 Fluxo da implementação do algoritmo de Barnes-Hut.

5.1.2.2 Parâmetros dessa Implementação

Essa implementação permite a alteração de vários parâmetros, de entrada e configuração conforme a descrição a seguir:

Parâmetros de Entrada:

- *Número de corpos*: É possível variar o número de corpos utilizados na simulação. O programa necessita que o projetista forneça um arquivo com a massa, posição e velocidade inicial de todos os corpos. Para os testes, utilizamos um programa que gera corpos aleatoriamente. Foram utilizados 900 corpos gerados aleatoriamente.

Parâmetros de configuração:

- *Fator de reconfiguração*: É possível alterar os limites superiores e inferiores utilizados na avaliação da necessidade de reconfiguração da árvore de corpos. É possível alterar também o número de nós das sub-árvores localizadas nos computadores cujos fatores de distanciamento excedam os limites mínimos e máximos necessários para a realização da reconfiguração. No exemplo executado é necessário que pelo menos 3 nós excedam os limites mínimos e máximos (maior 5 ou menor que 0,06), para que haja a necessidade de reconfiguração. É possível ainda definir a quantidade de computadores pedindo reconfiguração, no exemplo executado, são necessários que pelo menos 2 computadores requeiram a reconfiguração para que a mesma ocorra. Reduzindo o valor máximo ou aumentando o valor mínimo, o número de reconfigurações tende a aumentar.
- *Valor de θ*: É possível alterar o valor de θ necessário para considerar se um corpo está distante o suficiente de um conjunto de corpos. θ é utilizado na comparação com d/r, onde d é o valor do lado do quadrado onde se situa o conjunto de corpos e r é a distância do corpo em relação ao centro de massa desse conjunto de corpos. Em [36], os autores recomendam para θ valores entre 1 e 0,5. Quanto maior for o valor de θ, maior será o valor de r para que não seja necessário descer mais um nível da árvore. Isso quer dizer que com valores de θ menores diminuem a quantidade de cálculos de força realizada, enquanto que com valores maiores de θ aumentam a quantidade de cálculos de força realizados. Em [38], os autores realizaram um estudo detalhado sobre o efeito do valor de θ no número de cálculos de força.
- *Número de iterações*: Número de ciclos de cálculo de força em todos os corpos da simulação.

5.1.2.3 Resultados

O arquivo de resultados contém as informações sobre todos os corpos em todas as iterações do programa. As informações incluem:

1. Número de identificação do corpo
2. Valor da massa de cada corpo

3. Coordenadas x e y de cada corpo.
4. Valor da velocidade de cada corpo
5. Valores da força aplicada no corpo, um valor para cada coordenada

5.1.3 Distribuição e Parâmetros da Aplicação

A distribuição é feita conforme mostrado na Figura 5.3. Os computadores utilizados foram 5 Pentium 3 de 1 Ghz, com 256 megaBytes de memória RAM. Utilizamos 900 corpos gerados aleatoriamente, e cada computador recebeu uma região com aproximadamente 220 corpos. A quantidade de corpos em cada computador varia no decorrer da simulação, de acordo com o deslocamento dos mesmos. No caso dessa implementação, o valor de θ utilizado foi 0,8 e os limites superiores e inferiores do fator de reconfiguração foram 5 e 0,06 respectivamente.

5.1.4 Comparação de Desempenho entre as Simulações Seqüencial e Distribuída

A Tabela 5.1. mostra os tempos de execução obtidos no algoritmo de Barnes-Hut para a simulação seqüencial e simulação distribuída. Os tempos são referentes aos computadores indicados na segunda linha da tabela.

A simulação prosseguiu até 350 iterações de cálculo de força. Nessas 350 iterações, a árvore foi reconfigurada 54 vezes. Os tempos foram medidos em segundos, utilizando o comando *time* do sistema operacional *Linux*. Foram feitas 3 medidas para cada simulação. A Tabela 5.1 apresenta a média aritmética dessas 3 medições.

Seqüencial	Distribuído
234,88 segundos	109,24 segundos

Tabela 5.1. Tempos de execução da implementação do algoritmo de Barnes-Hut.

Podemos notar que nesse exemplo houve um ganho de desempenho de 2,15 com o uso do simulador distribuído. O ganho de desempenho foi calculado como a razão entre o tempo da simulação seqüencial e o tempo da simulação distribuída.

O programa realiza cálculos com funções recursivas, utilizando números de ponto flutuante sem limite de precisão, o que implica uma grande carga de computação, mas o tráfego de eventos não é elevado. O desempenho é limitado pelo fato de que os computadores só podem realizar a próxima iteração quando todos os outros computadores estiverem terminado a iteração corrente. A realização da reconfiguração também diminui o desempenho, pois todos os outros computadores ficam ociosos enquanto a mesma é realizada.

Cada um dos 4 processos *Sender* executou 3204 vezes, e receberam 3204 eventos. O processo *n_body* executou 55 vezes e recebeu 55 eventos, e foram necessárias 2426 unidades de tempo do simulador para encerrar a simulação .

5.2 Exemplo 2: Rede de Comutadores ATM

Nesse exemplo, simulamos uma rede de comutadores ATM (*Asynchronous Transfer Mode*). ATM é uma tecnologia de comunicação orientada à conexão, de multiplexação e comutação de dados. É utilizada para transportar pequenos pacotes de tamanho fixo denominados como *células* através de uma rede de alta velocidade. A tecnologia ATM permite a integração e o transporte de voz, vídeo, imagens e dados sobre uma mesma rede [27].

Neste exemplo, também estamos interessados em ilustrar a aplicação da simulação distribuída no projeto de um sistema embutido. Por essa razão, escolhemos implementar uma rede de comutadores ATM, cuja configuração depende de parâmetros a serem selecionados de acordo com as informações obtidas através da simulação.

5.2.1 Comutação de Células ATM

A função de um comutador ATM é receber células nas portas de entrada e redirecioná-las para as portas de saída correspondentes, mantendo a ordem de chegada das células para cada conexão [42].

5.2.1.1 Formatos das Células ATM

As células ATM possuem o tamanho de 53 bytes, sendo 5 bytes para o cabeçalho e os outros 48 bytes para informações (carga útil) [26].

O cabeçalho da célula ATM possui vários campos, entretanto, para simplificar esse exemplo, somente consideramos alguns desses campos. São eles:

- Identificador de Conexão Virtual, VCI (*Virtual Channel Identifier*)
- Identificador de Caminho Virtual, VPI (*Virtual Path Identifier*)
- Campo de informações, que contém os dados transportados pela célula ATM.

Os identificadores de VCI e VPI são necessários para que os comutadores possam efetuar o chaveamento das células ATM, como veremos na seção 5.2.1.4.

5.2.1.2 Conexões Virtuais

As células numa rede ATM são transportadas através de uma *conexão de canal virtual* (*Virtual Channel Connection- VCC*). Um VCC é uma conexão entre a origem e o destino de uma célula, e é composto de conexões virtuais estabelecidas nos enlaces da rede, denominadas de *enlaces de canal virtual* (*Virtual Channel Link- VCL*). O comutador ATM encaminha as células para o próximo VCL da conexão estabelecida pelo VCC. O canal virtual descreve o percurso unidirecional de células com um mesmo número de VCI (*Virtual Channel Identifier-* VCI) [26][43][44].

O VCI é interpretado apenas pelo *enlace de canal virtual* (VCL). Isso quer dizer que o VCI das células percorrendo um determinado VCC pode mudar de acordo com os comutadores que fazem parte da conexão.

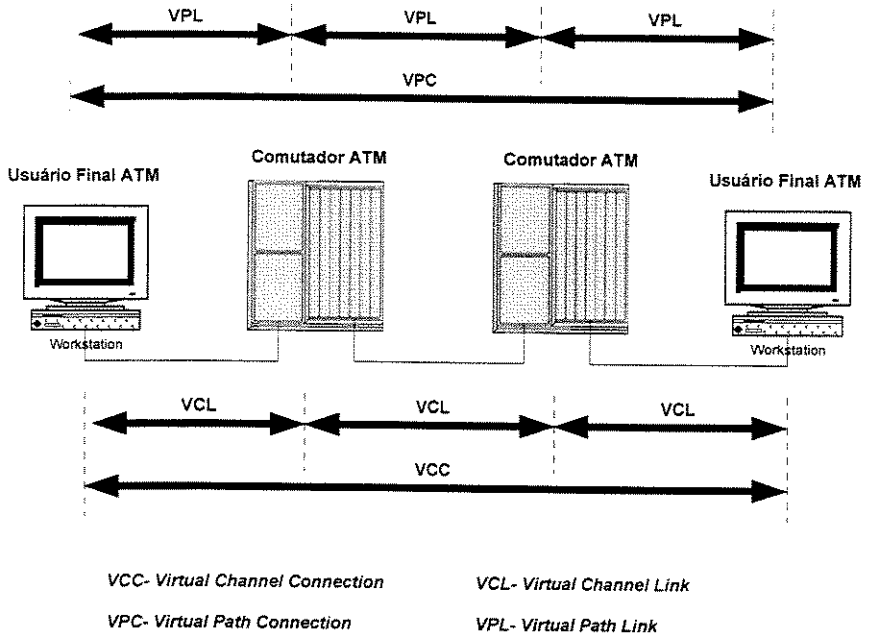


Figura 5.6. Relação entre VCC, VCL, VPC e VPL [26].

A Figura 5.6. mostra a relação entre um VCC e um VCL. Observe que o VCC é uma conexão entre os usuários finais de origem e de destino, enquanto que o VCL é uma conexão entre dois componentes da rede (usuários/comutadores).

5.2.1.3 Caminhos Virtuais

As conexões virtuais trafegam por caminhos virtuais para poderem chegar a seu destino. Cada enlace de canal virtual VCL pertence a um *caminho virtual ATM* (Virtual Path - VP), sendo que vários canais virtuais podem fazer parte de um mesmo caminho virtual. Um caminho virtual é representado por um *enlace de caminho virtual* (*Virtual Path Link* - VPL). Um VPL é o caminho virtual entre 2 pontos, enquanto que uma *conexão de caminho virtual* (*Virtual Path Connection* - VPC) é a concatenação de um ou mais VPLs, formando um caminho virtual entre o usuário ATM de origem e o usuário final ATM [26][43][44].

O VPI de uma célula ATM apenas faz sentido em um enlace de caminho virtual (VPL). Isso quer dizer que o VPI das células percorrendo um determinado VPC pode mudar de acordo com os comutadores que fazem parte da conexão.

A Figura 5.6. também mostra a relação entre um VPC e um VPL. Podemos notar que o VPC é a caminho entre a origem e o destino, enquanto que o VPL é um caminho entre 2 componentes da rede (usuários/comutadores).

5.2.1.4 Roteamento das Células

O roteamento de células ATM é feito através de troca de identificadores virtuais. Em cada comutador existe uma tabela de roteamento que contém pares de identificadores utilizados no roteamento. Quando uma célula chega no comutador, o mesmo identifica o enlace de origem (VCL) e a porta de entrada da célula. O comutador consulta a tabela de roteamento que relaciona o VCL de origem e a porta de entrada com o VCL de destino e a porta de saída. A célula é então atualizada e retransmitida pela porta de saída adequada [26][42]. Ou seja, o encaminhamento efetivo das células é feito através da troca dos campos VCI e VPI.

Como existem dois identificadores em redes ATM, sendo um para as conexões virtuais e o outro para os caminhos virtuais, pode haver dois tipos de comutação: comutação por canal virtual (VC) e comutação por caminho virtual (VP). Na comutação por canal virtual, ambos os identificadores VCI e VPI das células são trocados quando a mesma passa pelo comutador. Já na comutação por caminho

virtual, apenas o identificador VPI das células é trocado quando a mesma passa pelo comutador. Nesse exemplo utilizamos os dois tipos de comutadores.

A Figura 5.7. mostra um exemplo de como os identificadores virtuais VCI e VPI são modificados pelo comutador. Note que as conexões com VCI de entrada 1 e 2 são chaveadas por ambos VCI e VPI (comutação por canal virtual), enquanto que as conexões com VCI de entrada 5 e 6 são direcionadas para apenas um VPI diferente (comutação por caminho virtual).

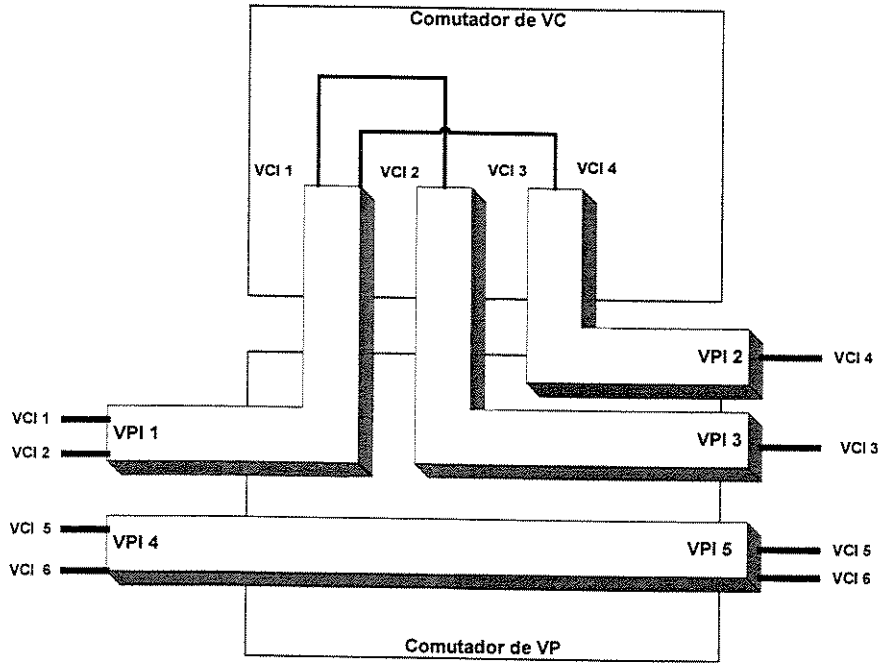


Figura 5.7. Comutadores de VC e VP.

5.2.1.5 Exemplo

A Figura 5.8. mostra as conexões e os caminhos virtuais utilizados na transmissão de células entre 3 usuários. O exemplo ilustra as diferenças entre os identificadores VC e VP do protocolo de conexão do ATM e mostra também a troca de identificadores realizada pelos comutadores. No exemplo, ilustramos a transmissão de uma célula do usuário *A* até o usuário *B* e a transmissão de uma célula do usuário *A* até o usuário *C*.

- Quando *A* transmite para B: VPI= 1 e o VCI= 1.

- Quando *B* recebe de A: VPI= 3 e o VCI= 1.
- Quando *A* transmite para *C*: VPI= 1 e o VCI= 2.
- Quando *C* recebe de *A*: VPI= 4 e o VCI= 2.

Podemos notar que por um mesmo caminho virtual VP podem passar várias conexões virtuais VC.

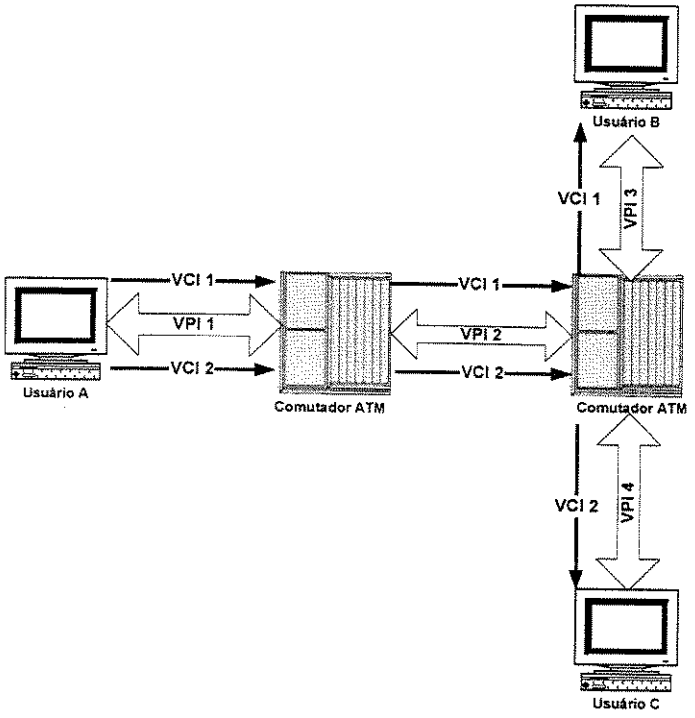


Figura 5.8. Exemplo do funcionamento dos identificadores VC e VP.

5.2.2 Modelo do Comutador

O comutador ATM *NxM* que implementamos tem a estrutura mostrada na Figura 5.9. O comutador mostrado tem *N* controladores de porta de entrada (*Input Port Controllers* - IPC) e *M* controladores de porta de saída (*Output Port Controllers* - OPC). Cada IPC possui uma fila de B_i células, enquanto cada OPC possui uma fila de B_o células. O conjunto composto por um IPC_i e um

OPC_i com o mesmo índice *i* é implementado como um único controlador de porta PC_i (*Port Controller- PC*) [45].

Cada comutador também possui uma rede de interconexão *(Inter-connection Network - IN)*. A rede de interconexão realiza o chaveamento das células entre os IPCs e os OPCs. A *IN* só pode encaminhar uma célula a cada instante de tempo para um determinado OPC. Caso mais de uma célula seja destinada para um mesmo OPC em um mesmo instante de tempo, a mesma deve ser armazenada em uma fila ou descartada.

Comutadores com fila de armazenamento de células podem ser de 3 tipos:

- Com fila de entrada, onde as células que não puderem ser enviadas são armazenadas no próprio IPC.
- Com fila de saída, onde as células que forem direcionadas para um mesmo OPC em um mesmo instante de tempo, são primeiro chaveadas pelo *IN* (rede de interconexão) e depois armazenadas na fila, para então serem transmitidas pelo OPC para o próximo comutador.
- Com fila compartilhada, onde existe uma fila centralizada compartilhada por ambos IPCs e OPCs. Esta fila está disponível para todas as células que não puderem ser enviadas.

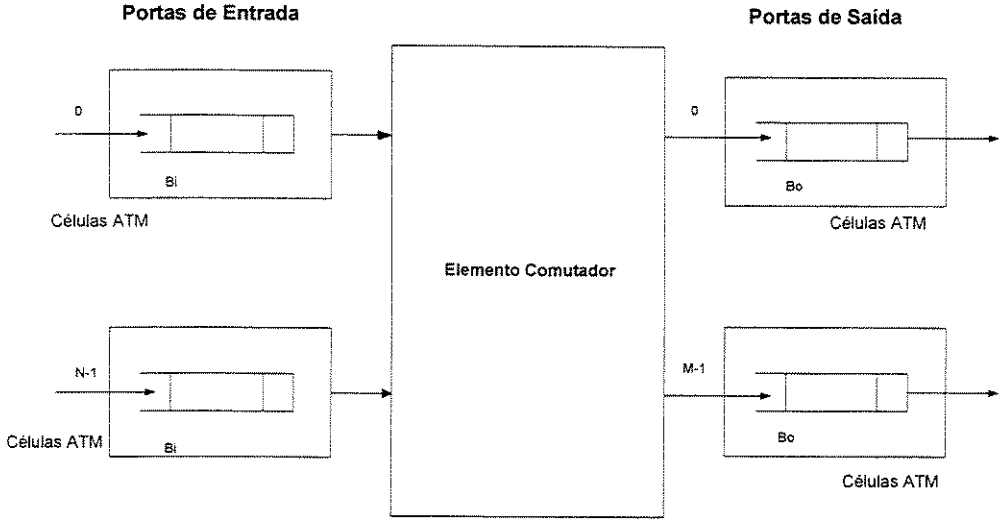


Figura 5.9. Comutador genérico.

No nosso exemplo, implementamos comutadores com fila de entrada.

5.2.3 Comutador com Fila de Entrada

Em nosso exemplo utilizamos um comutador de três fases com fila nas portas de entrada [45]. A estrutura de blocos desse comutador é mostrada na Figura 5.10. O comutador com fila de entrada possui N controladores de porta de entrada IPC, cada um com uma fila com capacidade para armazenar B_i células. O comutador possui N controladores de portas (PC_i, $i=0, ...,N-1$), cada um desses com uma entrada IPC e um controlador de porta de saída OPC. O comutador possui também uma rede de ordenação (*Sorting Network* - SN), uma rede de roteamento (*Routing Network* -RN) e uma rede de alocação de canal (*Allocation Network - AN*). Essas 3 redes serão explicadas nas próximas subseções.

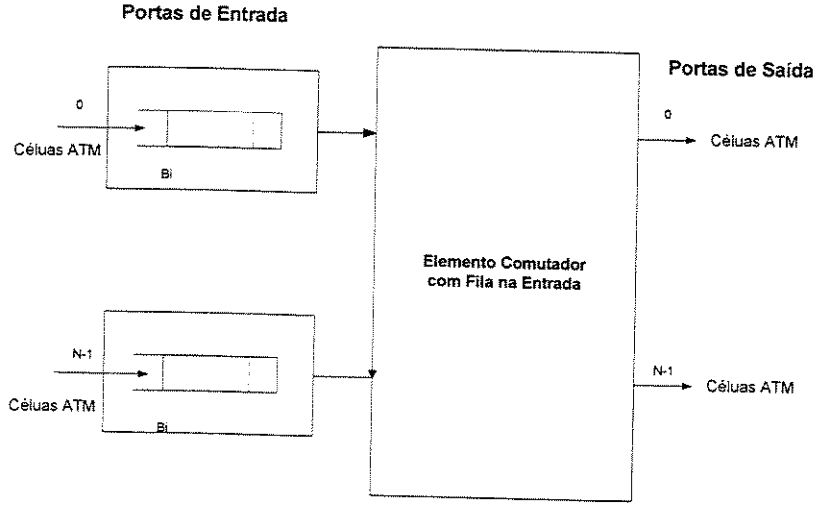


Figura 5.10. Comutador com fila nas portas de entrada.

Para encaminhar as células, o algoritmo de encaminhamento utiliza 3 tipos de pacotes de controle:

- Pacote de requisição REQ(*j, i*): É composto pelo endereço de destino *j* (endereço da porta de saída) da célula que estiver no topo da fila do PC_i, e o endereço *i* que indica qual o PC_i que gerou o pacote de requisição REQ.
- Pacote de aceitação ACK(*i, a*): Inclui o endereço *i* do PC_i para o qual o pacote é destinado e o bit de liberação *a*, que indica o resultado da disputa realizada na fase 2.

- Pacote de dados DATA(*j, cell*): Contém o endereço de destino *j* (endereço da porta de saída) e a célula a ser transmitida

O algoritmo de encaminhamento de células possui 3 fases. Essas fases são executadas em um ciclo de serviço do comutador (veja a seção 5.2.4.3 para maiores detalhes sobre tempos de serviço), onde cada ciclo é iniciado com a execução do algoritmo de 3 fases e encerrado com o envio das células. As fases do algoritmo de 3 fases são:

1. Fase de sondagem: controladores de porta (PC) requerem permissão para transmitir a célula armazenada na sua fila de entrada; os requerimentos são processados de maneira a permitir um requerimento por porta de saída.
2. Fase de reconhecimento: Baseado no processamento realizado na fase 1, os sinais de reconhecimento são enviados de volta para cada PC requerente.
3. Fase de dados: os PCs que tiveram seus requerimentos deferidos transmitem as suas células.

5.2.3.1 Fase de Sondagem

Nessa fase, cada controlador de porta (PC_i) que possuir células em sua fila de entrada envia um pacote de requisição REQ(*j, i*) através da rede de interconexão. Os pacotes de requisição REQ(*j, i*) enviados pelos PC_i são ordenados pela rede SN em ordem crescente, utilizando os campos de destino e origem como chaves primárias e secundárias de ordenação, para que os requerimentos destinados para uma mesma porta de saída fiquem adjacentes na saída da rede SN.

Os pacotes de requisição REQ(*j, i*) são então enviados para a rede AN que libera apenas um pacote por porta de saída. A arquitetura do comutador só permite que cada controlador de porta de saída OPC envie apenas uma célula por ciclo de serviço. Por isso é necessário que haja um sistema de arbitragem para poder decidir que células serão enviadas por cada OPC.

A rede AN gera então o bit de liberação, sendo 0 para o pedido liberado e 1 para pedidos não liberados. A prioridade de liberação das células é feita pela ordenação realizada pela rede SN.

5.2.3.2 Fase de Reconhecimento

Na fase de reconhecimento, a rede AN gera os pacotes de aceitação ACK(*i, a*) incluindo o campo de origem recebido da rede SN dentro do pacote de requisição REQ(*j, i*), mais o bit de liberação gerado pela rede AN. O pacote de aceitação ACK(*i, a*) é entregue através da rede SN para o controlador de porta PC_i que gerou o pedido. Os pacotes de aceitação ACK(*i, a*) não colidem entre si, uma vez que o endereço *i* é único. Por isso cada controlador PC_i não pode gerar mais do que um requerimento por ciclo de serviço do comutador.

5.2.3.3 Fase de Dado

Na fase de dados, logo que o controlador PC_i recebe o pacote de aceitação ACK(*i, a*). Caso o resultado do bit de liberação seja 0, o PC_i transmite o pacote de dados DATA(*j, cell*), que transporta a célula do topo de sua fila de entrada para a porta de saída *j*. Caso o resultado do bit de liberação seja 1, a célula é mantida na fila e nenhuma célula desse PC_i é transmitida nesse instante de tempo. O PC_i gerará um novo requerimento no próximo ciclo de serviço do comutador para tentar enviar as células que não foram servidas.

5.2.3.4 Exemplo de Comutação

A Figura 5.11. mostra um exemplo da troca de pacotes que ocorre durante a execução do algoritmo de 3 fases, com 8 controladores de porta PC. Pela figura podemos ver o progresso dos pacotes ou células de acordo com a fase do algoritmo. Note que apenas 4 dos 7 requerimentos são aceitos, uma vez que duas portas de saída são endereçadas por mais de um requerimento.

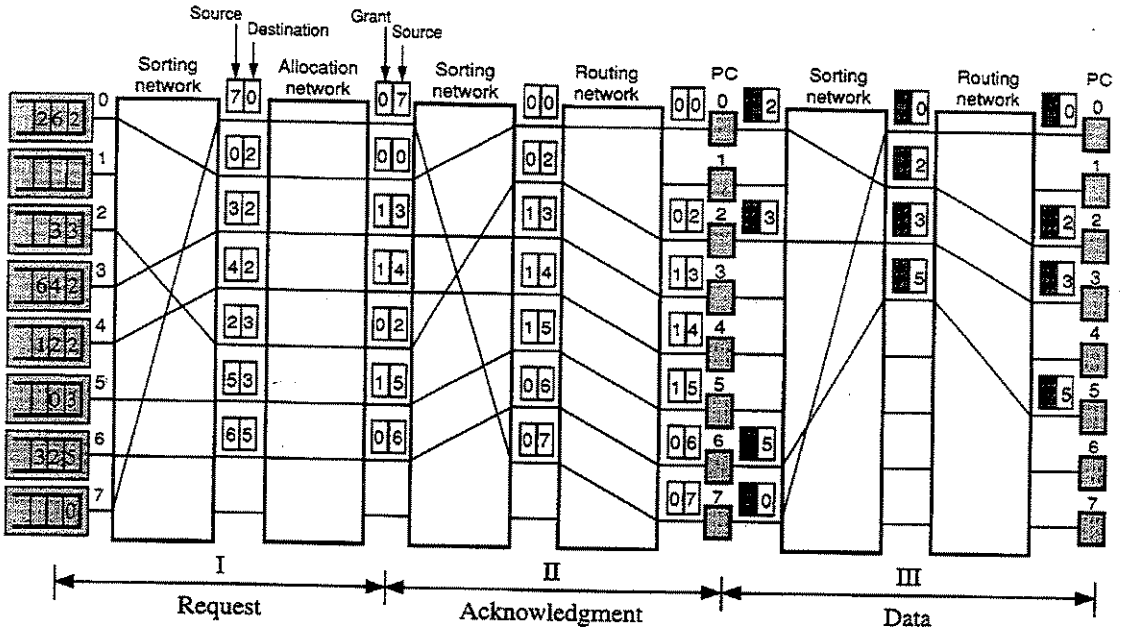


Figura 5.11. Exemplo de comutação com o algoritmo de 3-fases [45].

5.2.4 Visão Simplificada do Comutador

O modelo do comutador foi implementado em Java, podendo ser executado de forma distribuída ou local.

5.2.4.1 Modelo Simplificado

Os comutadores podem ser de dois tipos:

- Comutador de canal virtual, que analisa ambos os campos VCI e VPI das células (tipo 1)
- Comutador de caminho virtual, que analisa apenas o campo VPI da célula (tipo 2).

O comutador do tipo 1 possui 4 controladores de porta (PCs), com um gerador de célula para cada PC. Esse comutador é um comutador de destino, isso quer dizer, que ao receber células com um determinado valor de VCI, as mesmas não são retransmitidas.

O comutador do tipo 2 possui 4 controladores de porta (PCs), que recebem células de comutadores de ambos os tipos, retransmitindo-as de acordo com o valor do VPI contido nas células.

Uma classe em Java que chamamos de *Switch* implementa o comutador. Ambos os tipos de comutadores possuem:

- *N* objetos da classe *PC*, onde *N* é o número de portas de entrada do comutador. A classe *PC* modela os controladores das portas de entrada, implementa a fila de entrada, gera os pacotes pacote de requisição REQ e os pacotes DATA.
- Um objeto *AN*, que implementa a rede de alocação. A classe *AN* analisa todos os pacote de requisição REQ e indica quais deles devem ser liberados.
- Um objeto *SN*, que implementa a rede de ordenação. A classe *SN* ordena os pacotes de requisição REQs de uma maneira um pouco diferente do que a mostrada em 5.2.3.1. A ordenação é feita com o campo destino como chave primária de ordenação, como chave secundária, tem um campo de prioridade aleatório. Esse campo de prioridade atribui um valor aleatório de prioridade para os pedidos feitos pelos controladores *PCs* para a liberação das células. Com as prioridades aleatórias, todos os *PCs* são servidos homogeneamente, mantendo todas as filas de entrada com tamanhos praticamente equivalentes.

Para a simulação, criamos uma rede de comutadores composta por 27 comutadores, divididos em 3 grupos de 9. Desses 27, 12 são comutadores de canal virtual, (*tipo 1*), enquanto que o restante são comutadores de caminho virtual, (*tipo 2*).

A rede de roteamento é implementada dentro da classe *Switch*, realizando a troca dos identificadores *VCI* e *VPI* das células no caso de ser um comutador do *tipo 1*, e realizando a troca apenas do VPI no caso de um comutador do *tipo 2*. Depois de realizar a troca de identificadores, a rede de roteamento gera eventos para enviar as células para os seus comutadores de destino.

5.2.4.2 Geração do Tráfego de Células

O tráfego das células é gerado pela classe *CellGenerator*. Cada *PC* pertencente a um comutador do *tipo 1* precisa de um objeto *CellGenerator*.

A classe *CellGenerator* é um gerador de tráfego aleatório *on/off* [46]. Inicialmente, o programa determina qual será o intervalo de tempo que o gerador irá permanecer ocioso (*off*). Esse intervalo é aleatório. Por exemplo, se o número sorteado pelo gerador aleatório for 4, isso quer dizer que a classe irá enviar um evento para ele mesmo com tempo $t+4$. Ao receber esse evento (*on*), a classe gera outro número aleatório para determinar quantos eventos serão enviados ao *PC*. Por exemplo, se o número sorteado pelo gerador aleatório for 3, isso significa que a classe irá enviar para o controlador *PC* células nos tempos $t+1$, $t+2$, $t+3$. Isso é feito para simular surtos de células na entrada do *PC*, fato que acontece em situações reais. O programa gera aleatoriamente o número da porta de saída para qual a célula será enviada.

5.2.4.3 Funcionamento do Comutador

A primeira coisa a ser feita pelo comutador é agendar o próximo período de serviço. Comutadores possuem um tempo mínimo de serviço. Por exemplo, caso o tempo de serviço seja igual a 3, isso quer dizer que o comutador necessita de 3 unidades de tempo para enviar as células. Por exemplo, se o comutador iniciou o serviço no tempo t e uma célula chegou no controlador *PC* no tempo $t+2$, então essa célula só será processada no tempo $t+3$. Caso cheguem células enquanto o comutador estiver ocupado, as mesmas são armazenadas nas filas de entrada dos *PCs*.

Quando o comutador recebe o evento indicando o início do serviço, o programa segue o algoritmo de 3 fases descrito em 5.2.3. Se no final do envio das células ainda houver células a serem enviadas, então o comutador escalona outro evento de serviço. Se não houver células a serem enviadas, o comutador fica ocioso, esperando a chegada de alguma célula para poder reiniciar o período de serviço.

A simulação continua até que um determinado tempo de simulação seja atingido.

5.2.4.4 Resultados

A execução do modelo permite analisar os seguintes resultados:

- Tamanho máximo das filas dos *PCs*. Esse resultado permite avaliar se o tamanho máximo que as filas atingiram durante a simulação é condizente com o tamanho da fila que será utilizada no projeto do hardware.

- Atraso médio de envio das células. Mede o tempo médio de trânsito da célula dentro do comutador, do momento de sua chegada até o momento de sua retransmissão.
- Atraso médio das células destinadas ao comutador. Essa medida é referente aos comutadores do grupo 1. Ela mede o tempo de vida total da célula, de sua origem ao seu destino final.
- Quantidade de células enviadas pelo comutador. Mede a quantidade de células retransmitidas pelo comutador.
- Quantidade de células recebidas pelo comutador. Essa medida é referente aos comutadores do grupo 1. Mede a quantidade de células cujo destino final é o comutador em questão (células que não são retransmitidas).

5.2.4.5 Parâmetros do Exemplo do Comutador

O modelo apresentado permite a alteração de vários parâmetros, como os descritos abaixo:

- Número de controladores de porta *PCs*. Pode-se aumentar ou diminuir o número de *PCs*, o que aumenta o número de portas de entrada e de saída. Na nossa implementação, foram utilizados 4 *PCs* para os comutadores do tipo 1 e 4 para os comutadores do tipo 2.
- Tempo de serviço. O tempo de serviço é o parâmetro que mais influi no desempenho do comutador. Ao escolher um tempo alto de serviço, aumenta-se o tamanho das filas de armazenamento de células nos *PCs*. Diminuindo o tempo de serviço, diminui-se o tamanho das filas de armazenamento de células nos *PCs*.
- Intervalo de geração de células. O modelo permite a especificação do intervalo que será utilizado pela classe *CellGenerator* na geração das células. Por exemplo, se o intervalo especificado for de 1 a 8, isso quer dizer que serão geradas aleatoriamente de 1 a 8 células para serem enviadas para o *PC* do comutador. Esse parâmetro deve ser ajustado de acordo com o valor do tempo de serviço, pois se especificarmos um intervalo muito amplo e um tempo de serviço muito alto, as filas do *PC* e o atraso médio de envio podem aumentar consideravelmente. Na nossa simulação, foi utilizado o intervalo de 1 a 5.
- Intervalo de ociosidade da geração de células. Esse parâmetro especifica o período no qual a classe *CellGenerator* ficará ociosa, isto é, sem gerar células. Por exemplo, se o intervalo

especificado for de 1 a 8, isso quer dizer que a classe *CellGenerator* irá enviar um evento pedindo a geração de células com tempo *t+valor_gerado* para ela mesma. Como mencionado acima, esse parâmetro deve ser escolhido com cuidado para não influir negativamente no desempenho do comutador.

5.2.4.6 Exemplo da Utilização dos Parâmetros

Mostraremos nessa seção como a alteração do valor do tempo de serviço do comutador ATM pode alterar o resultado da simulação e as decisões de projeto.

As tabelas abaixo mostram os resultados obtidos para 3 tempos de serviço diferentes do comutador. Foi utilizado o mesmo tráfego de células nas 3 simulações realizadas.

Podemos notar pela tabela que a diminuição do tempo de serviço causa um impacto imediato no tamanho médio das filas de entrada do comutador. Podemos notar uma diminuição no tamanho das filas de 1149 para 94 com a redução do tempo de serviço de 3 para 2 unidades de tempo, isso para os comutadores do tipo 1. Para os comutadores do tipo 2 não houve redução no tamanho das filas de entrada porque eles começaram a receber um número de células muito maior proveniente dos comutadores de tipo 1, que começaram a enviar uma média de 5500 células a mais depois que o tempo de serviço foi reduzido. O atraso médio do envio de células também sofreu uma redução acentuada, caindo de 1754 para 132 ms. Esse atraso é a duração da célula, entre a sua origem e a chegada no seu comutador de destino. O atraso diminuiu devido a diminuição do tempo de serviço do comutador, que começou a enviar células com maior freqüência.

Ao reduzir o tempo de serviço para 1 unidade de tempo, notamos que o tamanho das filas foi reduzido ainda mais, chegando a apenas 6 na média de ambos os tipos de comutadores. O atraso de envio também foi reduzido drasticamente, chegando a 0 em alguns comutadores.

Com essas informações, o projetista pode dimensionar o tamanho das filas de entrada dos comutadores de acordo com o tráfego e o tempo de serviço escolhido.

Mostramos com esse exemplo como o projetista pode utilizar o simulador no auxílio na tomada de decisões de projeto. Mostramos também como a alteração de um parâmetro alterou drasticamente os resultados da simulação.

Tempo de serviço =3

	Comutador A (Tipo 1)	**Comutador B (Tipo 1)**	**Comutador C (Tipo 2)**	**Comutador D (Tipo 2)**
Tamanho médio das filas (n.° de células)	1149	1152	104	108
Quantidade de células enviadas	8773	8692	12557	12389
Atraso Médio de envio de Células em ms.	1754,4	1669,2	1908,9	1970,7

Tempo de serviço =2

	Comutador A (Tipo 1)	**Comutador B (Tipo 1)**	**Comutador C (Tipo 2)**	**Comutador D (Tipo 2)**
Tamanho médio das filas (n.° de células)	94	73	150	112
Quantidade de células enviadas	13074	12930	18873	18672
Atraso Médio de envio de Células em ms.	132,3	103,6	350,2	373,5

Tempo de serviço =1

	Comutador A (Tipo 1)	**Comutador B (Tipo 1)**	**Comutador C (Tipo 2)**	**Comutador D (Tipo 2)**
Tamanho médio das filas (n.° de células)	6	6	6	6
Quantidade de células enviadas	13442	13388	19761	19809
Atraso Médio de envio de Células em ms.	7,2	6,5	4,1	5,5

Tabela 5.2. Resultado da simulação dos comutadores ATM

5.2.5 Distribuição do Modelo

A distribuição é feita de acordo com a Figura 5.13. A rede simulada possui rotas com endereçamento fixo. As células podem ser direcionadas para qualquer comutador a partir de comutadores do tipo 1. São alocados 9 comutadores por computador, sendo utilizados 3 computadores nessa simulação. A Figura 5.12. mostra os 9 comutadores alocados em um computador. Os computadores utilizados foram 3 Pentium 3 de 1 Ghz, com 256 mega Bytes de memória RAM. Numa segunda simulação, foram utilizados 3 computadores SUN SPARC Ultra 10.

Na execução seqüencial do modelo, todos os comutadores foram alocados em um mesmo computador.

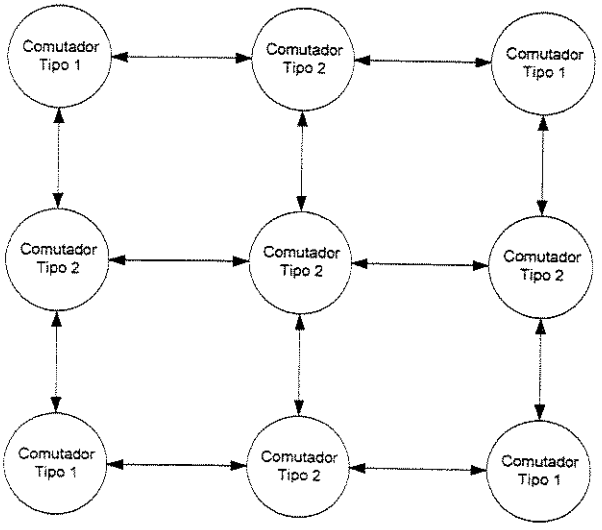


Figura 5.12. Comutadores alocados para um mesmo computador.

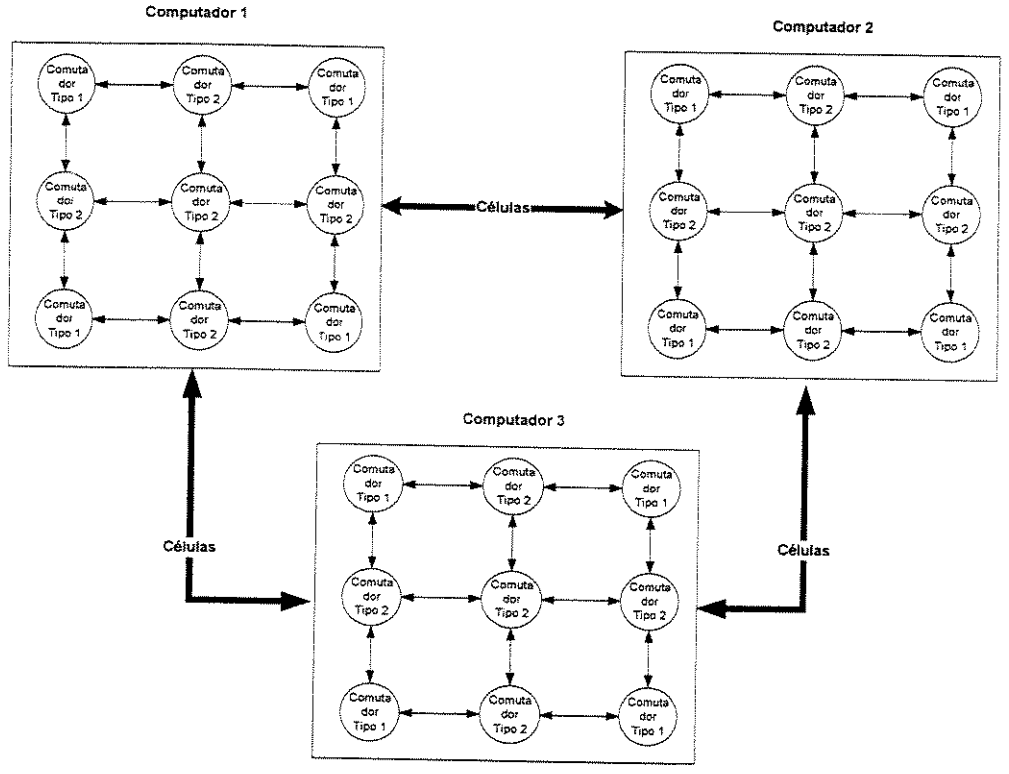


Figura 5.13. Distribuição dos comutadores nos computadores disponíveis.

5.2.6 Resultados da Simulação

A Tabela 5.3. mostra os tempos de execução obtidos pela execução do modelo do comutador no simulador remoto e no distribuído. O modelo foi simulado até o tempo de simulação 100000 em unidades de tempo discreto do simulador. Foram executadas três simulações da mesma rede de comutadores.

Seqüencial	**Distribuído**
Ultra SPARC 10 614,17 segundos	Ultra SPARC 10 187,23 segundos
Pentium 3 1GHz 391,97 segundos	Pentium 3 1GHz 136,20 segundos

Tabela 5.3. Comparação de desempenho entre o simulador seqüencial e o distribuído.

Podemos notar que nesse exemplo houve um ganho de desempenho com o uso do simulador distribuído da ordem de 3,28 vezes com o uso da Ultra SPARC 10 e 2,87 vezes com o uso do Pentium 3 1GHz. O ganho de desempenho é calculado pela divisão do tempo total da simulação seqüencial pelo tempo da simulação distribuída.

Esse exemplo tem como característica o enorme fluxo de eventos entre os comutadores, o que causa um grande *overhead* na árvore de eventos. Outro fator de estrangulamento é as filas dos *PCs*, que foram implementadas com o uso de vetores de tamanho variável. Esses vetores aumentam de tamanho por demanda. Essas estruturas geram uma carga computacional considerável caso tenham que alterar seu tamanho com freqüência, o que ocorre quando o tráfego de células é muito intenso.

Na implementação distribuída, o gerenciamento da árvore de eventos e das filas dos *PCs* é distribuído pelos 3 computadores que compõem a simulação, gerando o ganho de desempenho observado.

Os comutadores do tipo 1 executaram uma média de 26.000 vezes, recebendo a mesma quantidade de eventos. Comutadores do tipo 2 executaram uma média de 18.000 vezes, recebendo a mesma quantidade de eventos.

5.3 Conclusões

Nesse capítulo mostramos dois exemplos de aplicações simuladas com o nosso simulador distribuído. A simulação dos exemplos mostrou a capacidade do simulador distribuído de diminuir o tempo de simulação dos exemplos em relação a simulação seqüencial dos mesmos.

6. Conclusões

Esse trabalho de tese consistiu no desenvolvimento de um simulador distribuído para auxílio ao projeto de sistemas embutidos e de dois exemplos de aplicação.

Esse simulador permite simular as características únicas do projeto de sistemas embutidos, isto é, especificações de operação em tempo real, com restrições quanto às dimensões físicas, consumo de energia e tempo de desenvolvimento.

O modelo para sistema embutido utilizado é uma versão modificada do diagrama de fluxo de dados [22]. Por ser modelo funcional pode ser usado desde a etapa de especificação do projeto. A versão modificada do diagrama de fluxo de dados, referenciada como modelo **multitaxa,** permite especificar a quantidade de informações necessárias para ativar cada módulo do sistema.

Graças ao emprego da metodologia da simulação orientada a eventos discretos distribuída e conservadora, o simulador tem uma estrutura eficiente e flexível, funcionando sem deadlocks. Além disso, os recursos da linguagem Java permitem ao simulador continuar a funcionar mesmo quando ocorre falha resultante de um erro em um módulo da aplicação.

Implementado com o objetivo de reduzir o tempo de simulação de projetos de sistemas embutidos, o simulador pode também ser utilizado para auxiliar o desenvolvimento de projetos subdivididos em módulos armazenados em localidades distantes com comunicação via Internet. Desenvolvido em Java, no ambiente SOLARIS e Linux, com fácil adaptação para os ambientes Windows, NT, 2000 ou 9x. A versão atual pode funcionar com qualquer número de módulos do projetista e com qualquer quantidade de computadores.

O simulador possui apenas 2737 linhas de código e a execução de suas instruções geralmente constituem um parcela pequena do processamento durante a simulação. Apesar de não haver pretensão de atingir o padrão dos pacotes de software desenvolvidos para fins comerciais, a qualidade do código do simulador foi garantida pela realização de vários testes com até 19 horas de execução ininterrupta.

Algumas características merecem menção por diferenciar esse simulador de outros simulares: remoção de eventos com a mesma estampa de tempo, tratamento de exceções, inicialização dos módulos do projetista, facilidade de integração entre os módulos do projetista e o simulador.

Os exemplos de aplicação apresentados tiveram por objetivo avaliar o desempenho do simulador e ilustrar sua utilização durante o projeto de sistemas embutidos. Com 3 computadores, a simulação distribuída foi 3,3 vezes mais rápida que a seqüencial para o exemplo da implementação da rede de comutadores ATM. Características da implementação, como vetores de tamanho variável, e tamanho da árvore de eventos, influem consideravelmente no desempenho, favorecendo a implementação distribuída da rede de comutadores sobre a implementação seqüencial.

6.1 Sugestões para Trabalhos Futuros

Sugerimos como trabalhos futuros, inserir novas funcionalidades ao simulador, como:

- Alocação automática dos módulos do projetista. O simulador deve determinar a distribuição dos módulos do projetista de acordo com o número de computadores disponíveis e o tempo de execução de cada módulo.
- *Lookahead* dinâmico. Com o uso do algoritmo proposto em [1], podemos variar o valor do *lookahead* durante a simulação, reduzindo a quantidade de eventos nulos enviados e melhorando o desempenho do simulador.
- Projeto de interface do usuário. Interfaces gráficas mais sofisticadas podem vir a facilitar a interação entre o projetista e o simulador. Sugestões de aplicativos auxiliares para essa interface são um gerador de arquivos de parâmetros e um gerador automático de arquivos de eventos.
- Implementação da metodologia otimista. O projetista poderia escolher se deseja executar o simulador com a metodologia conservadora ou otimista. A idéia é usar simulações preliminares com a finalidade de escolher a forma de simulação, conservadora ou otimista, que oferece melhor desempenho para a aplicação.

7. Bibliografia

[1] A. Ferscha, S. K.Tripathi, "Parallel and distributed simulation of discrete event systems", In Parallel and Distributed Computing Handbook, ed. A. Y. Zomaya, 1003-1041. New York: McGraw-Hill 1995.

[2] J. Misra, "Distributed Discrete-Event Simulation", *Computing Surveys, Vol. 18, No. 1, March* 1986.

[3] http://webopedia.internet.com/TERM/E/embedded_system.html

[4] A.C. Lear, "Shedding Light on Embedded Systems," *IEEE Software, January/February*, pag. 122-125, 1999.

[5] E. A. Lee, J. Davis II, M. Goel, C. Hylands, B. Kienhuis, J. Liu, X. Liu, "Ptolemy II: Heterogeneous concurrent modeling and design in Java", Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 1999.

[6] B. A. W. van Halderen, B. J. Overeinder, "Fornax: Web-based Distributed Discrete Event Simulation in Java", *The ACM 1998 workshop on Java for high performance network computing*, 1998.

[7] D. Jeferson, H. Sowizral, "Fast Concurrent Simulation Using the Time Warp Mechanism", em *Distributed Simulations,* pag. 63-69, SCS, The Society for Computer Simulation, 1985.

[8] A. Ferscha, M. Richter, "Java Based Conservative Distributed Simulation", *Proceedings of the 1997 Winter Simulation Conference*, 1997.

[9] A. H. Buss, K. A. Stork, "Discrete Event Simulation on the World Wide Web using Java", *Proceedings of the 1996 Winter Simulation Conference, 780-785.*

[10] R. Helaihel, K. Olukotun, "Java as a Specification Language for Hardware-Software Systems," *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pp. 690-697. Novembro de 1997.

[11] J. Young, R. Newton, M. Shilman, A. Tabbara, "Design and Specification of Embedded Systems in Java Using Successive, Formal Refinement" *35th Design Automation Conference (DAC)*, 1998.

[12] http://getafix.stanford.edu/rashhel/javaspec.html

[13] C. S. Horstmann, G. Cornell, "*Core Java, Volume I e Volume II*", *Sun Microsystems Press*, 2000.

[14] G. Bollela, J. Gosling, "The Real-Time Specification for Java", *IEEE Computer Magazine*, pag. 47-53, Junho de 2000.

[15] http://catalog.com/softinfo/objects.html -What is Object-Oriented Software?

[16] http://www.icsi.berkeley.edu/~sather/Publications/article.html

[17] http://www-2.cs.cmu.edu/~jch/java/

[18] A. Ad-Tabataba, M. Cierniak, Guei-Yuan Lueh, V. M. Parikh, J. M. Stichnoth, " Fast, Effective Code Generation in a Just-In-Time Java Compiler" *SIGPLAN 98*, 1998.

[19] J. Magee, J. Kramer. "*Concurrency- State Models and Java Programs*" John Wiley & Sons, 1999.

[20] D. Lea, "Concurrent Programming in Java- Design Principles and Patterns" Addison Wesley, 1997.

[21] D. D. Gajski, F. Vahid, S. Narayan, J. Gong, "*Specification and Design of Embedded Systems*", Prentice Hall, 1994.

[22] E. A. Lee, D. G. Messerschmitt, "Synchronous Data Flow", *In Proceedings of the IEEE, Vol. 75, No. 9,* 1987.

[23] T. Murata, "Petri Nets: Properties, Analysis and Applications", *Proceedings of the IEEE*, Vol. 77, No 4, pp. 541-580 1989.

[24] M. C. Williamson, "Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications", *PHD Thesis, University of California, Berkeley*, 1991

[25] J. Liu, "*Real-Time Data Processing*", Prentice Hall, 2000.

[26] R. S. French, M. S. Lam, J. R. Levitt, K. Olukotun. "A General Method for Compiling Event-Driven Simulations" *32nd ACM/IEEE Design Automation Conference*, Junho, 1995.

[27] A. M. Alberti, "SimATM: Um Ambiente para a Simulação de Redes ATM", *Tese de Mestrado, UNICAMP*, 1998.

[28] S. K. Shanmugan, "*Simulation and Implementation Tools for Signal Processing and Communication Systems*", IEEE *Communications Magazine*, Julho 1994.

[29] A. Budd, "*Classic Data Structure in C++*", Addison-Wesley Publishing, 1994.

[30] Example Application: Event-Driven Simulation http://www.science.uva.nl/cng/documentation/Sun_Compilers_5.0/stdlib/stdug/general/11_3.htm

[31] C. Hansen, ``Hardware logic simulation by compilation," in *25th ACM/IEEE Design Automation Conference*, pp. 712-715, 1988.

[32] Laung-Terng Wang, N. E. Hoover, E. H. Porter, J. J. Zasio "SSIM: A Software Levelized Compiled-Code Simulator", *24th ACM/IEEE Design Automation Conference* 1987.

[33] Z. Wang, P. M. Maurer, "LECSIM: A Levelized Event Driven Compiled Logic Simulator", *27th ACM/IEEE Design Automation Conference,* 1990.

[34] J. Misra, K. M. Chandy, "Distributed Simulation: A case study in design and verification of distributed systems" *IEEE Trans. In Software Engeneer,* 1979.

[35] R. E. Bryant, "Simulation of packet communication architecture computer systems", Tech. Rep. MIT, LCS, TR-188, Massachusetts Institute of Technology, Cambridge, Mass, 1977.

[36] J. Barnes, P. Hut, "A Hierarchical O(N log N) Force Calculation Algorithm". *Nature*, 324:446-449, 1986.

[37] Data Structures and Algorithms http://www.cs.mcgill.ca/~kiu/cs251/proj24.htm

[38] Description of the Barnes-Hut Algorithm http://physics.gmu.edu/~large/lr_forces/desc/bh/bhdesc.html

[39] D. A. Patterson, J. L. Hennessy, "*Computer Architecture: a quantitative approach*", Morgan Kaufmann Publishers, 1996.

[40] D. Blackston, T. Suel, "Highly Portable and Efficient Implementations of Parallel Adaptive N-Body Methods"- *SC '97: High Performance Networking and Computing (Supercomputing '97)*, Novembro 1997 .

[41] http://www.supercomp.org/sc96/proceedings/SC96PROC/HU/INDEX.HTM- A Data-Parallel Implementation of O(N) Hierarchical N-body Methods

[42] L.F. Soares, G. Lemos, S. Colcher, "*Redes de Computadores*", Editora Campus, 1995.

[43] M. de Pricker, "Asynchronous Transfer Mode: Solution for Broadband ISDN" Prentice Hall, 1995.

[44] U. D. Black, "*ATM: Foundation for Broadband Networks*", Prentice-Hall, 1995.

[45] A. Pattavina, "*Architecture and Performance in Broadband ATM Networks*", John Wiley & Sons, 1998.

[46] J. M. Pitts, J. A. Schormans, "*Introduciton to ATM design and performance*" John Wiley & Sons, 1996.

Apêndice A.

Exemplo de Utilização do Modelo Multitaxa

Para demostrar a utilização do modelo Multitaxa, utilizaremos o mesmo para modelar um sistema de controle de velocidade de automóveis, *cruise control* [19].

O sistema de *cruise control* tem a finalidade de manter constante a velocidade do automóvel em um valor pré-determinado. O sistema é controlado por 3 botões, *resume, on* e *off*. Quando o motor do automóvel estiver funcionando e *on* é pressionado, o sistema grava a velocidade atual e mantém o automóvel nessa velocidade. Quando o acelerador, o freio ou o botão *off* é pressionado, o *cruise control* é desabilitado, e a velocidade do automóvel passa a ser controlada pelo condutor. Se *resume* for pressionado, o sistema acelera ou desacelera o automóvel até a velocidade ajustada anteriormente. A Figura 7.1 mostra a representação do sistema de *cruise control* com o uso do modelo Multitaxa.

A.1 Módulos do Sistema *cruise control*

O sistema de *cruise control* é composto dos módulos descritos abaixo:

- *Motor*: representa o motor do automóvel. Indica o estado do motor (ligado ou desligado) e a velocidade atual do automóvel.
- *Acelerador*: sensor que indica se o acelerador foi pressionado ou não.
- *Freio*: sensor que indica se o freio foi pressionado ou não.
- *Botões*: sensor que indica qual dos botões do *cruise control* foi pressionado. Os botões são: *On*, *off*, e *resume*.

- *Rastreamento de sensores*: recebe as informações dos sensores e do motor do automóvel. Do motor são recebidos duas informações, a velocidade atual e o estado do motor.
- *Cruise Control*: realiza o controle do *cruise control*. O módulo recebe informações sobre os sensores e o estado do motor. O módulo envia um comando para habilitar/desabilitar o *Controlador de Velocidade*.
- *Controlador de Velocidade*: Esse módulo é ativado pelo módulo *Cruise Control*. Recebe também a velocidade atual do automóvel, armazena a mesma e envia um comando para o módulo *Aceleração* para acelerar ou desacelerar o automóvel.
- *Aceleração*: recebe um comando indicando se deve acelerar o desacelerar o automóvel para uma velocidade determinada pelo módulo *Controlador de Velocidade*.

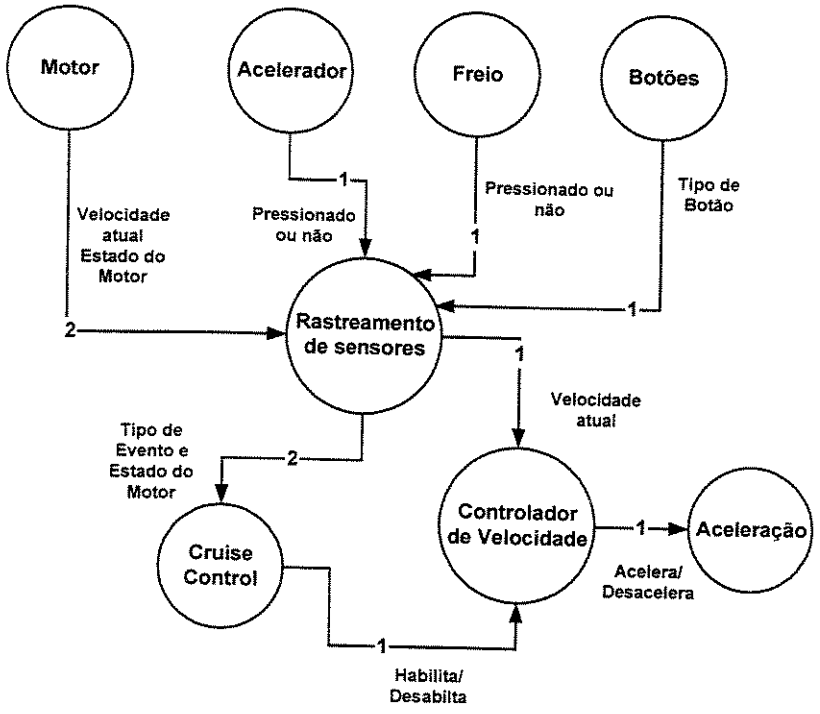


Figura 7.1 Representação do sistema *cruise control*

A.2 Fluxo de Informações

Inicialmente, os sensores enviam continuamente informações para o módulo *Rastreamento de Sensores*. Ao receber um evento correspondente ao botão *on* do sensor *Botão*, o módulo envia um evento que representa o botão *on* para o módulo *Cruise Control*. O módulo envia também um evento indicando se o motor do automóvel está ligado ou não.

O módulo *Cruise Control*, ao receber ambos os eventos do módulo *Rastreamento de Sensores*, verifica primeiro se o motor do carro está ligado, caso positivo, o módulo envia um evento para habilitar o módulo *Controlador de Velocidade*. Caso o motor esteja desligado, nenhum evento é enviado.

O módulo *Controlador de Velocidade* recebe do módulo *Rastreamento de Sensores* o valor atual da velocidade do automóvel, grava esse valor e envia um evento para o módulo *Aceleração* para o mesmo manter a velocidade atual.

Caso o módulo *Cruise Control* receba um evento correspondente ao botão *off* do *cruise control*, o mesmo envia um evento para desabilitar o módulo *Controlador de Velocidade*. O módulo *Controlador de Velocidade* então envia um evento para o módulo *Aceleração* para desativar a manutenção da velocidade do automóvel.

Caso o módulo *Cruise Control* receba um evento correspondente ao botão *resume* do *cruise control*, o mesmo envia um evento para habilitar o módulo *Controlador de Velocidade* e fazer com que o mesmo ajuste a velocidade do automóvel para o valor gravado anteriormente. O módulo *Controlador de Velocidade* então envia um evento para o módulo *Aceleração* para que o mesmo acelere ou desacelere o automóvel até que a velocidade desejada seja atingida.

Note que o módulo *Rastreamento de Sensores* necessita receber 2 eventos do módulo *Motor*, além de precisar receber 1 evento de cada sensor para poder executar uma vez. O módulo *Cruise Control* necessita receber 2 eventos do módulo *Rastreamento de Sensores* para poder executar uma vez.

Apêndice B.
Classes do Simulador

Descreveremos nessa seção o funcionamento das principais classes do simulador distribuído orientado a eventos implementado nesse trabalho de tese.

B.1 Classe *Eventos*

A classe *Eventos* implementa os eventos na simulação orientada a eventos. A classe possui os campos:

- *Tempo*, que contém a estampa de tempo em que o evento será executado
- *Conteúdo*, que contém a mensagem transmitida pelo evento, nesse caso a mensagem é um objeto ou tipo de dados da linguagem Java.
- *Canal*, que identifica o canal para o qual o evento é destinado.
- *Remetente*, que identifica o módulo que gerou o evento (esse campo é opcional).
- *Vetor*, contém um vetor de objetos que pode ser utilizado no lugar do campo *Conteúdo*, caso seja necessário.

Os campos *Conteúdo* e *Vetor* foram implementados com o uso da classe *Object*, nativa do Java, e podem receber quaisquer tipos de dados, como inteiros, *strings*, *double* ou outros objetos do Java criados pelo projetista.

B.2 Classe *Árvore*

A classe *Árvore* implementa a fila de eventos que armazena os eventos durante simulação. Foi utilizado a estrutura de árvore binária da biblioteca *(TreeSet)* da linguagem Java. Esta estrutura apresenta um desempenho de busca superior a da lista encadeada (veja a seção 3.3.1.3). A classe

Árvore possui métodos para escalonar um evento e remover e/ou consultar o evento com menor estampa de tempo. A classe *Árvore* envia os dados dos eventos removidos para a classe *Resultados*.

B.3 Classe *E*

```
6
1000000
barnes_hut.n_body
1|1|1|3|1
barnes_hut.Sender
2|1|1|1|5
barnes_hut.Sender2
3|1|1|1|9
barnes_hut.Sender3
4|1|1|1|6
barnes_hut.Sender4
5|1|1|1|3
barnes_hut.Control
6|4|2|1|2
```

Note o nome do *package barnes_hut* seguido do nome do módulo, por exemplo, *Sender*.

B.3.2 Arquivo *canalX.dat*

O arquivo *canalX.dat,* onde o *X* é o número do computador na qual os módulos irão executar, é utilizado para estabelecer os canais de saídas externos de um computador.

Esse arquivo tem o formato:

a|b|c|d|e➔onde:

a= número do canal de saída.

b= destinatário dos eventos, localizado em outro computador.

c= tempo de execução do módulo, em unidades de tempo discreto.

d= tempo de propagação do evento no canal em questão, em unidades de tempo discreto.

e= Prioridade de execução do evento.

Exemplo de um arquivo de canal:

```
0|5|6|1|8|
0|8|1|1|11|
```

B.3.3 Arquivo *urldata.dat*

O arquivo *urldata.dat* contém os endereços *IP* dos computadores participantes da simulação e os nomes dos canais de comunicação de entrada e saída dos módulos, utilizados para estabelecer as conexões entre os computadores participantes da simulação.

Esse arquivo tem o formato:

a → número de computadores que fazem parte da simulação.

url →número *IP* ou *url* dos computadores participantes da simulação.

b| →nome dos canais de comunicação de entrada. O *nc* indica que o módulo a ser simulado nesse computador não possui um dos canais de comunicação.

c| → nome dos canais de comunicação de saída.

Exemplo:

```
2
rmi://copacabana.dca.fee.unicamp.br:50000/
nc|
c1|c2|  → note que existem dois canais de entrada nos módulos alocados nesse computador
rmi://rocas.dca.fee.unicamp.br:50001/
c1|
c3|
```

B.4 Classe *GeraObjetos*

A classe *GeraObjetos* é utilizada para instanciar os objetos das classes criadas pelo projetista. As classes do projetista devem obedecer algumas diretrizes na sua criação, mostradas na seção 2.4.

A classe *GeraObjeto* foi criada para automatizar o funcionamento do simulador, pois a mesma inicializa todos os objetos do projetista sem que o mesmo precise alterar o código fonte do simulador.

B.5 Classe *ConsSimul*

A classe *ConsSimul* constrói um servidor que irá gerenciar os *buffers* dos canais de entrada dos módulos do projetista e criar conexões com outros computadores para os quais os eventos serão enviados. Veja a seção B.7 para maiores detalhes sobre o servidor.

Depois que todos os objetos do simulador são inicializados e as conexões de recebimento e envio de eventos estão estabelecidas, é chamado o método *Kernel* da classe *KernelDistribuído,* que irá gerenciar a simulação.

B.6 Classe *KernelDistríbuido*

A classe *KernelDistribuído* gerencia a simulação distribuída. A classe segue o algoritmo abaixo:

1. Chama o método *EnviaEventoNulo* da classe *EnviaDistribuído* para enviar eventos nulos para todos os canais de saída dos módulos simulados no computador.
2. Consulta os *Buffers* dos canais de entrada, identifica qual dos canais de entrada possui o evento com o menor estampa de tempo e faz o LVTH ser igual a essa estampa de tempo.
3. Chama o método *ExecutaInterno*, que vai executar todos os eventos internos com tempo menor ou igual que o LVTH.
4. Executa os eventos externos, pelo método *ExecutaExterno*. Esse método remove do topo dos *buffers* eventos externos com estampa de tempo igual ao LVTH, executando-os se forem eventos ou descartando-os se forem eventos nulos. O método realiza uma busca nos *buffers* por eventos com tempo igual ao LVTH (isso pode ocorrer com freqüência, pois é permitida a criação de eventos com a mesma estampa de tempo) e executa-os em caso afirmativo.
5. Envia eventos externos com estampa de tempo menor ou igual LVTH + *lookahead*
6. Envia eventos nulo com estampa de tempo igual a LVTH + *lookahead* para cada canal de saída.
7. A execução é encerrada quando o LVTH for igual ao tempo de término da simulação ou até que não haja mais eventos para executar.

B.7 Classe *Servidor*

A classe *Servidor* cria os objetos *BufferImpl* que implementam os canais de entrada. Essa classe roda em uma *thread*, concorrentemente com o resto do simulador, para possibilitar que os *LPs* situados nos computadores participantes da simulação possam enviar eventos externos concorrentemente com o funcionamento do simulador distribuído.

B.8 Classe *Sinalizador*

A classe *Sinalizador* implementa a relação multitaxa entre as entradas e saídas dos módulos do simulador. Os eventos destinados ao módulo são armazenados num vetor. A quantidade de eventos necessária para a ativação do módulo deve ser conhecida antecipadamente.

A classe *Sinalizador* prevê a possibilidade da chegada de uma quantidade de eventos maior do que a estipulada pelo projetista para a ativação do módulo. Os eventos excedentes que porventura forem enviados para o módulo serão armazenados em *Buffers* temporários, implementados com o uso de listas encadeadas. Escolhemos a lista encadeada pelo baixo custo computacional, pois os eventos que serão inseridos estarão ordenados.

A classe *Sinalizador* recebe como parâmetro dados referentes ao módulo do projetista que ela controla. Dados como: o número dos canais de entrada do módulo, taxa de recebimento de eventos em cada canal e uma referência ao módulo do projetista. Para enviar eventos, a classe *Sinalizador* chama o método *Consumidor* do módulo do projetista (veja a seção 2.4).

B.9 Classe *EnviaEventos*

Os métodos da classe *EnviaEvento* são chamado pela classe do projetista, para permitir a transmissão de eventos entre as classes do projetista e o simulador. Para o envio de eventos para outros computadores é necessário utilizar a classe *EnviaEventoDistribuídos*.

A classe *EnviaEvento* recebe os dados a serem inseridos no evento, cria o evento e o insere na árvore de eventos. Essa classe acrescenta automaticamente a prioridade e a estampa de tempo do evento. A classe fornece um método de envio de eventos que não insere a estampa de tempo do evento automaticamente, caso o projetista deseje inserir uma estampa de tempo diferente da definido nos parâmetros do simulador.

B.10 Classe *RecebeEventosDistribuídos*

A classe *RecebeEventoDistribuídos* distribui o evento para o módulo do projetista para qual esse evento é destinado.

B.11 Classe *BufferImpl*

A classe *BufferImpl* implementa os *Buffers* dos canais externos de entrada dos *LPs*. Os objetos dessa classe são acessados remotamente por outros computadores, o que permite a transmissão de eventos entre os computadores participantes da simulação.

Os *Buffers* são implementados por uma lista encadeada, uma vez que os eventos inseridos estão sempre ordenados. A classe possui os seguintes métodos:

- Método *Escalona:* insere o evento no *Buffer*. Esse método é acessado remotamente pelos *LPs* antecessores ao *LP* local.
- Método *RemoveEvento:* retira o evento do *Buffer*.
- Método *ConsultaBuffer:* realiza a consulta de eventos no *Buffer*, sendo utilizado pela classe *KernelDistribuido*. Esse método possui um mecanismo de bloqueio, que bloqueia o simulador se não houver eventos no *Buffer*. Quando um evento for inserido no *Buffer* por um *LP*, o simulador é desbloqueado. O método possui também um mecanismo de *timeout*, que envia uma interrupção para finalizar a simulação caso não haja eventos no *Buffer* depois de transcorrido um valor de tempo arbitrário.

B.12 Classe *RecebeEventos*

A classe *RecebeEvento* remove o evento com a menor estampa de tempo da árvore de eventos internos, envia esse evento para o objeto da classe *Sinalizador* apropriado e atualiza o *Relógio.*

B.13 Classe *Relógio* e *RelógioDistribuído*

A classe *Relógio* armazena o tempo do último evento extraído da árvore local.

A classe *RelogioDistribuído* herda os métodos da classe *Relógio*, armazena o tempo do LVTH e do LVT, que no caso é o tempo do último evento extraído da árvore.

B.14 Classe *EnviaEventosDistribuídos*

A classe *EnviaEventoDistribuídos* é utilizada para enviar eventos para os outros computadores que participantes da simulação. Essa classe possui vários métodos para realizar o envio de eventos de forma distribuída, explicados abaixo:

- O método *EnviaEventoNulo*, envia eventos nulos para todos os canais de saída do módulo. Esse método cria um evento nulo com estampa de tempo igual ao LVTH mais o *lookahead* (veja a seção 4.2). Esses eventos são inseridos nos *buffers* de entrada (objeto *BufferImpl*)dos computadores destinatários.
- O método *EnviaEventoD* insere os eventos externos gerados pelos módulos locais em uma árvore temporária.
- O método *EnviaDefinitivo* analisa os eventos contidos na árvore temporária e envia aqueles que possuírem a estampa de tempo menor ou igual ao LVTH mais *lookahead.* Esses eventos são inseridos nos *buffers* de entrada dos computadores destinatários.

B.15 Classe *Resultados*

Essa classe emite relatórios com informações obtidas durante a simulação. As informações incluem: tempo total de simulação, quantidade de eventos direcionadas a um determinado canal, número de execução de um módulo, tempo total gasto em um determinado canal, porcentagem de utilização do canal, listagem completa dos eventos, tempo de execução do módulo, número de eventos restantes nos *Buffers* da classe *Sinalizador*, tamanho máximo atingido pelos *Buffers*. As informações são obtidas para todos os canais e módulos.

Apêndice C.

Arquivos de Resultados

Nesse apêndice vamos mostrar exemplos dos arquivos de resultados emitidos pelo simulador.

C.1 Arquivo *Simulador*

Esse arquivo contém a quantidade de vezes que o módulo executou, o tempo de execução do módulo, o percentual de utilização do módulo, a quantidade de eventos destinados a cada canal, a utilização de cada canal, e o tempo total da simulação. A Figura B.1. ilustra esse arquivo.

```
O módulo barnes_hut.n_body executou 15  vezes
Tempo total de execução do modulo = 15
Utilização de 1 %

O módulo barnes_hut.Control executou 14  vezes
Tempo total de execução do modulo = 14
Utilização de 1 %

Quantidade de Eventos destinados para o canal 1 = 15

Tempo total de execução do canal 1 = 15
Utilização de 1 %


Quantidade de Eventos destinados para o canal 6 = 56
Tempo total de execução do canal 6 = 56
Utilização de 5 %
```

Figura B.1. Arquivo Simulador.txt

C.2 Arquivo *Buffers*

Esse arquivo contém o tamanho máximo e a quantidade de eventos que restaram nos buffers de entrada de cada módulo do projetista. A Figura B.2. ilustra esse arquivo.

```
Eventos que restaram nos buffers

Processo 0 Buffer 0 sobraram 3 Eventos
Buffer teve tamanho máximo de 4
Processo 1 Buffer 0 sobraram 0 Eventos
Buffer teve tamanho máximo de 2
Processo 2 Buffer 0 sobraram 1 Eventos
Buffer teve tamanho máximo de 3
```

Figura B.2. Arquivo Buffers.txt

C.3 Arquivo *Canal*

Esse arquivo contém o rastreamento completo de todos os eventos que foram enviados à um canal. O rastreamento inclui: canal para o qual o evento foi destinado, tempo que o evento foi executado e o conteúdo do evento. A Figura B.3. ilustra esse arquivo.

```
Evento destinado a=  11
No tempo  5
Valor=  ATM.Cell@5f2d4b01

Evento destinado a=  11
No tempo  5
Valor=  ATM.Cell@40514b01

Evento destinado a=  11
No tempo  5
Valor=  ATM.Cell@42a94b01

Evento destinado a=  11
No tempo  6
Valor=  ATM.Cell@40194b01
```

```
Evento destinado a=  11
No tempo  6
Valor=  ATM.Cell@42714b01

Evento destinado a=  11
No tempo  6
Valor=  true
```

Figura B.3. Arquivo Canal.txt.