UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE CIÊNCIAS E TECNOLOGIA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DISSERTAÇÃO DE MESTRADO

IMPLEMENTAÇÃO E VALIDAÇÃO DE COMPONENTES PARA A CONSTRUÇÃO DE AMBIENTES DE SIMULAÇÃO DE REDES TCP/IP

FLÁVIO GONÇALVES DA ROCHA

Maria Izabel Cavalcanti Cabral
Orientadora

Campina Grande, Agosto de 2002

UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE CIÊNCIAS E TECNOLOGIA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

IMPLEMENTAÇÃO E VALIDAÇÃO DE COMPONENTES PARA A CONSTRUÇÃO DE AMBIENTES DE SIMULAÇÃO DE REDES TCP/IP

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal de Campina Grande – Campus I, como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Redes de Computadores e Sistemas Distribuídos

FLÁVIO GONÇALVES DA ROCHA

Maria Izabel Cavalcanti Cabral
Orientadora

Campina Grande, Agosto de 2002

ROCHA, Flávio Gonçalves da

R672I

Implementação e Validação de Componentes para a Construção de Ambientes de Simulação de Redes TCP/IP.

Dissertação (Mestrado), Universidade Federal de Campina Grande, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande, Paraíba, Agosto de 2002.

125p. Il.

Orientadora: Maria Izabel Cavalcanti Cabral.

Palavras-Chave:

1. Redes de Computadores
2. Simulação Digital
3. JavaBeans.

CDU - 621.391.

"IMPLEMENTAÇÃO E VALIDAÇÃO DE COMPONENTES PARA A CONSTRUÇÃO DE AMBIENTES DE SIMULAÇÃO DE REDES TCP/IP"

FLÁVIO GONÇALVES DA ROCHA

DISSERTAÇÃO APROVADA EM 28.08.2002

PROFª MARIA IZABEL CAVALCANTI CABRAL , D.Sc
Orientadora

PROFª PATRÍCIA DUARTE DE LIMA MACHADO, Ph.D
Examinadora

PROF. GUIDO LEMOS DE SOUZA FILHO, Dr.
Examinador

CAMPINA GRANDE – PB

“Quando se busca o cume da montanha,

não se dá importância às pedras do caminho.”

(Anônimo)

Agradecimentos

Gostaria de agradecer primeiramente a Deus, pela possibilidade de realizar este trabalho. Nos momentos mais difíceis senti-me amparado no seu amor.

À minha esposa, Maysa, pelo seu amor, incentivo e apoio.

Aos meus pais e irmãos pela força que sempre me deram durante todo este trabalho.

De forma muito especial à professora Maria Izabel Cavalcanti Cabral, minha orientadora, principalmente pela paciência, pela confiança e pela seriedade com que conduziu esta Dissertação. Sua orientação foi fundamental durante todo esse tempo.

Ao professor Jacques Philippe Sauvé que sempre esteve disposto a me atender. Aprendi muito com nossas conversas.

Aos meus amigos do mestrado pelos momentos agradáveis e pelas trocas de conhecimento, em especial a Fernando e a Márcio com quem sempre pude contar nas horas difíceis.

Aos meus amigos do LMRS, em especial a Iana, Petrônio, Alexandre, Cecir e Eustáquio.

Aos funcionários do DSC e da COPIN, em especial a Aninha pela sua colaboração.

Enfim, obrigado a todos que de alguma forma me ajudaram e me inspiraram.

Dedicatória

Dedico todo o esforço empregado neste trabalho à minha esposa, Maysa Suelma Palmeira Leite Rocha, que sempre me transmitiu amor, força, incentivo e entusiasmo sem os quais este trabalho não teria sido realizado. Muito obrigado meu amor.

Resumo

Estudos e análises de sistemas de computação e de comunicação de dados podem ser complexos devidos, em geral, a grande quantidade de recursos a serem considerados e ao relacionamento dinâmico que esses recursos apresentam. Adicionalmente, novas tecnologias se apresentam exigindo estudos e análises mais detalhados quanto as suas utilizações e desempenhos. A dinâmica desses sistemas depende da ocorrência de eventos, que representam, por exemplo, em um sistema de comunicação de dados, o término de transmissão de uma mensagem em um canal de comunicação. Como conseqüência, ambientes de simulação orientados a eventos muitas vezes se apresentam como a melhor alternativa para estudos e análises desses sistemas. O processo de construção de tais ambientes pode ser feito com o uso de componentes de software que possibilitam uma maior reutilização de código aumentando a eficácia desse processo. Nessa Dissertação, dois tipos de componentes são especificados e implementados: os que permitem a construção facilitada de ambientes de simulação orientados a eventos (relógio simulado, lista de eventos, geradores de valores aleatórios e controle da simulação, entre outros), e os que representam elementos essenciais de uma rede de computadores *TCP/IP* (fontes de pacotes, *hosts, enlaces* e roteadores). Como estudo de caso, um ambiente de simulação de redes *TCP/IP* foi construído e validado. Na especificação dos componentes foi utilizada a Linguagem de Modelagem Unificada (*UML*). A tecnologia para a implementação escolhida foi *Java* (*JavaBeans*) e o Ambiente de Desenvolvimento utilizado *Jbuilder.* Finalmente, os componentes podem ser reutilizados na construção de qualquer ambiente de simulação orientado a eventos que se aplique a modelos que apresentam contenção de recursos.

Abstract

Studies and analysis of computation and data communication systems could be complex due to the great amount of resources to be considered and their relationship. As new technologies appear more detailed studies and analysis are required. The dynamic of these systems depends on event occurrence. Simulation event-driven tools are frequently chosen for aiding studies and analysis of those systems. In the software development process of those tools software components can facilitate the software reuse increasing the efficiency of this process. This work, specifies and implements two types of components for building simulation tools: components for constructing event-driven simulation tools (such as simulated clock, randon value generators, event list, simulation flow control) and another components for simulating *TCP/IP* computer networks basic elements (such as packet sources, hosts, routers and links). As a case of study, a simulation tool was constructed. The work adopts *JavaBeans* technology, *Unified Modeling Language* (*UML*) for specification the components and *Jbuilder* as development tool. Finally, the components construted in this work can be reused to build any event-driven simulation tool.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A técnica da Simulação Digital apresenta-se como uma importante opção de análise na modelagem e avaliação de desempenho de redes de computadores em geral [Kronbauer, 98]. Essa técnica visa projetar e construir modelos de sistemas, em sistemas computacionais, com o objetivo de analisar os seus comportamentos diante de situações específicas [Kelton, 98].

A construção de ambientes de simulação que possibilitem modelar e avaliar os desempenhos de sistemas de computação e de comunicação de dados pode ser feita explorando os benefícios da reutilização de sistemas de software. A reutilização de código ganha destaque no processo de desenvolvimento de sistemas de software porque, se bem aplicada, pode reduzir significantemente o tempo gasto para desenvolver uma aplicação. No entanto, não só o código deve e pode ser reutilizado, mas também as próprias fases de análise e de projeto com vistas à reutilizar o conhecimento empregado e as principais decisões tomadas durante estas fases [Freire, 00].

A reutilização de software atinge um alto nível de abstração com o uso de componentes de software reutilizáveis [Freire, 00]. Componentes são elementos de software auto-suficientes e reutilizáveis que podem interagir entre si seguindo um grupo de regras preestabelecidas [Englander, 97]. O desenvolvimento de software, segundo o paradigma da componentização, determina que uma aplicação seja construída a partir de um conjunto de

componentes interligados. Esta abordagem é uma evolução natural da orientação a objetos em que um componente corresponde a um conjunto de classes inter-relacionadas, com visibilidade externa limitada [Silva, 00].

A construção de ambientes de simulação baseados em componentes pode não ser simples. Estudos mostram que no desenvolvimento de sistemas, as fases de análise (o que é o problema?) e de projeto (como solucionar o problema?) são cruciais, consumindo maior parte do tempo [Landin, 98].

O grupo de rede de computadores da UFCG tem investido no desenvolvimento de ferramentas de software baseadas em componentes [Freire, 00] [Wagner, 00] [Lula, 01]. Em [Wagner, 00] é apresentado uma especificação de componentes de software que representam as funcionalidades mínimas dos elementos necessários para modelar uma rede *TCP/IP*. O nível de detalhamento focalizou principalmente a camada *IP* fazendo com que os mecanismos que viabilizam a camada *Internet* fossem abordados.

Em [Lula, 01] é feita uma especificação de um ambiente de simulação para redes *TCP/IP*, englobando a especificação de componentes apresentada em [Wagner, 00], juntamente com a especificação de novos componentes que representam os elementos essenciais de um ambiente de simulação (escalonador de eventos, relógio, acumuladores estatísticos, entre outros).

Os trabalhos de [Lula, 01] e de [Wagner, 00] foram realizados utilizando a linguagem de modelagem *UML (Unified Modeling Language)* [Rumbaugh et al., 99], seguindo o processo de desenvolvimento apresentado em [Larman, 98], enfocando as fases de Planejamento e de Elaboração, que correspondem às atividades de análise e de projeto do sistema, respectivamente.

Esta Dissertação aborda componentes de software para serem reutilizados na construção de ambientes de simulação para redes de computadores, dando continuidade aos trabalhos propostos em [Lula, 01] e [Wagner, 00].

1.1 Objetivos

1.1.1 Objetivos Gerais

Esta dissertação apresenta a especificação e a implementação de componentes de software que visam a construção facilitada de ambientes de simulação voltados para sistemas computacionais e de comunicação de dados, explorando a reutilização de software.

Dois tipos de componentes são especificados e implementados: os que permitem a construção de ambientes de simulação orientados a eventos (relógio simulado, lista de eventos, geradores de valores aleatórios, controle da simulação, entre outros) e os componentes que representam elementos essenciais de uma rede de computadores *TCP/IP* (fontes de pacotes, *hosts*, *enlaces* e roteadores). Como estudo de caso, um ambiente de simulação de redes *TCP/IP* foi construído, validando a implementação dos componentes apresentada.

Foi adotado um processo incremental de desenvolvimento de software tendo como ponto de partida as especificações, aos níveis de análise e de projeto, apresentados em [Lula, 01] e [Wagner, 00], utilizando *UML*. A tecnologia para a implementação dos componentes escolhida foi *Java* (*JavaBeans*) [Sun, 02] e o Ambiente de Desenvolvimento utilizado foi o *Jbuilder* (versão 5.0) da [Borland, 02].

1.1.2 Objetivos Específicos

- ✓ Realizar estudo sobre a tecnologia *TCP/IP*.
- ✓ Estudar a técnica da Simulação Digital e ambientes de simulação voltados para a avaliação de desempenho de redes de computadores.
- ✓ Estudar técnicas e fundamentos da engenharia de software para o desenvolvimento de software orientado a componentes (*JavaBeans*).
- ✓ Estudar *UML,* a linguagem *Java* e o ambiente de desenvolvimento utilizado *Jbuilder*.
- ✓ Implementar e testar os componentes apresentados em [Lula, 01] e [Wagner, 00].

- ✓ Validar os componentes através da construção de um ambiente de simulação voltado para redes *TCP/IP* (estudo de caso).

1.2 Relevância

Ambientes de simulação orientados a objetos (*Object Oriented Simulation – OOS*) têm sido alvo de intensas pesquisas nos últimos anos [Roberts, 94]. A principal razão é o aumento da demanda por sistemas de software cada vez mais complexos que são, por natureza, maiores e mais heterogêneos que outros sistemas além de, na maioria dos casos, serem desenvolvidos e mantidos por várias pessoas.

Apesar da tecnologia *TCP/IP* ser bastante conhecida e difundida, há uma demanda por ferramentas para modelar e avaliar o desempenho de redes de computadores que utilizam esta tecnologia. A implementação de um ambiente de simulação orientado a componentes voltado à modelagem de redes *TCP/IP,* permitirá estudos sobre a arquitetura *Internet,* buscando avaliar e otimizar o seu desempenho. Além disso, uma vez que o ambiente de simulação seja implementado segundo o paradigma da orientação a componentes, os benefícios oferecidos por este tipo de abordagem, em especial, a reutilização de software, possibilitarão acrescentar, com facilidade, novas funcionalidades a esse ambiente que atendam à evolução da tecnologia *TCP/IP*.

Ainda, os componentes implementados, conforme [Lula, 01], podem ser reutilizados na construção de qualquer ambiente de simulação orientado a eventos que se aplique a modelos que apresentam contenção de recursos, tais como aqueles que representam sistemas de computação e sistemas de comunicação de dados.

É importante também ressaltar a multidisciplinaridade deste trabalho, uma vez que o mesmo abrange várias áreas, tais como Simulação Digital, Engenharia de Software e Redes Computadores.

1.3 Organização do Trabalho

No capítulo 2, é introduzida a técnica da Simulação Digital, enfocando a simulação digital baseada em eventos e ambientes de simulação orientados a objetos. Exemplos de ambientes de simulação voltados para redes de computadores são apresentados.

O capítulo 3 apresenta a tecnologia usada para criar componentes de software na linguagem *Java*, os *JavaBeans*.

No capítulo 4 é apresentada a especificação dos componentes de software, mostrando com detalhes as fases de análise e de projeto, construídas utilizando um processo de desenvolvimento de software incremental. Essa especificação teve como ponto de partida as especificações apresentadas em [Lula, 01] e em [Wagner, 00].

O capítulo 5 apresenta os principais aspectos da implementação dos componentes de software e do ambiente de simulação construído (estudo de caso). Também apresenta os testes realizados para fins de verificação da implementação.

O capítulo 6 mostra a validação dos componentes implementados. Esta consistiu na comparação dos resultados de simulações de modelos de redes *TCP/IP* obtidos através: (i) do ambiente de simulação desenvolvido nesta Dissertação e, (ii) do ambiente profissional *Arena* [Takus, 97].

Por fim, no capítulo 7 são apresentadas as conclusões do trabalho ressaltando aspectos relevantes observados no desenvolvimento dos componentes de software e do ambiente de simulação construído. Este capítulo finaliza apresentando sugestões de continuidade dos trabalhos desta Dissertação.

Capítulo 2

Simulação

Neste capítulo é introduzida a técnica da Simulação Digital, apresentando uma revisão dos principais conceitos inerentes a esta técnica. Ressaltam-se a simulação digital baseada em eventos e ambientes de simulação orientados a objetos. Exemplos de ambientes de simulação voltados para redes de computadores são apresentados.

2.1 Introdução

Simulação é o processo de construir um modelo que represente o comportamento de um sistema real ou não para realizar experimentos com o intuito de obter medidas de desempenho de interesse.

Um sistema pode ser definido como um conjunto de partes organizadas funcionalmente para formar um todo. Um subsistema é uma das partes que permite ser tratada como um sistema isolado. Assim, um sistema pode conter subsistemas [Almeida, 99]. Tal classificação, sistema ou subsistema, irá depender do nível de abstração (detalhamento) que está sendo considerado na análise. Por exemplo, se considerarmos a *Internet* um sistema, um roteador nela pode ser considerado um subsistema. No entanto, o próprio roteador poder ser considerado um sistema e os seus protocolos subsistemas.

O propósito da simulação é o de analisar um sistema através de seu modelo de modo a definir estratégias para a operação do mesmo tomando por base o estudo dos

resultados obtidos. O modelo de um sistema é resultado da análise deste destacando apenas as características relevantes ao estudo em questão. Tal modelo é construído através de abstrações e estas podem ser de vários níveis [D'Souza, 98].

Duas formas são geralmente usadas para avaliar o desempenho de um sistema:

- ✓ Campanhas de medição em sistemas já existentes (benchmarks); e
- ✓ Modelagem matemática. Aqui, a análise é feita ao nível de projeto por meio de um modelo do sistema real ou não.

Campanhas de medição coletam informações diretamente no sistema em estudo de modo a caracterizar o seu comportamento. Apresenta a desvantagem de só poder ser realizada em sistemas existentes e em execução.

A modelagem matemática permite fazer projeções de diversas configurações e situações do sistema a um baixo custo. Com isso, minimizam-se as probabilidades de erros de dimensionamento e de utilização do sistema [Souto, 93]. Através de abstrações, um modelo é criado representando as características de interesse do sistema em estudo. Esta alternativa de análise pode ser realizada através de duas formas:

- ✓ Técnicas exatas (estudos analíticos) substanciadas pela Teoria das Filas [Kleinrock, 75].
- ✓ técnicas aproximadas que caracterizam-se por utilizarem métodos numéricos para solucionar os modelos (Simulação Digital por exemplo).

Apesar da solução analítica ser mais econômica e eficiente, em determinados sistemas, um tratamento analítico requer suposições acerca de sua estrutura e comportamento de modo a simplificar o seu modelo para que o mesmo seja computacionalmente aceitável. Em sistemas muito complexos tais simplificações podem resultar na criação de um modelo que não reflita mais o sistema original. Nesses casos, a Simulação Digital é a única alternativa viável [Dias, 92].

2.2 Simulação Digital

A Simulação Digital é uma técnica numérica utilizada no suporte à tomada de decisões. Na prática, ela constitui-se no projeto e construção de modelos computadorizados de

sistemas reais ou imaginários com o intuito de compreender seus comportamentos frente à determinadas situações ou eventos [Kelton, 98].

A técnica da simulação baseia-se no uso de um modelo que representa as características relevantes do sistema estudado do ponto de vista de quem esta modelando [Almeida, 99]. Dessa forma, um modelo pode ser visto como uma representação (abstração) de um sistema. Pode-se ter uma representação formal da teoria ou uma representação empírica mas, usualmente, encontra-se uma combinação das duas.

O nível de detalhamento ou de abstração na criação de um modelo é muito importante. Quanto menos detalhado mais abstrato será o modelo. Conseqüentemente, é interessante que o modelo seja definido em vários níveis de abstração ou de detalhamento partindo do nível mais alto, que representa uma visão geral do sistema e, através de refinamentos, chegar ao nível mais baixo que mostra maiores detalhes da funcionalidade do sistema [D'Souza, 98]. Dessa forma, fica claro que quanto maior o detalhamento e, conseqüentemente, menor a abstração, maior será o tempo gasto na elaboração do modelo.

Na simulação, a construção da lógica do modelo irá despender entre 30 a 40 % do tempo total do projeto de simulação [Wagner, 00]. No entanto, tal percentual está intimamente relacionado à experiência das pessoas envolvidas. A qualidade e os dados a serem usados no modelo irão influenciar os resultados obtidos na simulação. Portanto, quanto maior for a experiência da equipe envolvida no projeto, menor será o tempo de desenvolvimento e maior será a probabilidade da criação de um bom modelo e da escolha correta dos dados a serem inseridos neste. Bons modelos e dados coerentes levam a resultados satisfatórios, e vice-versa [Kelton, 98].

No início, o uso da técnica da Simulação Digital não era simples em virtude da necessidade da modelagem matemática dos sistemas e da implementação de algoritmos em linguagens de programação de propósito geral, como *FORTRAN*, por exemplo. Com o surgimento de linguagens orientadas à simulação, na década de 50, a modelagem de sistemas tornou-se mais fácil. *GPSS, SIMSCRIPT, SLAM*, e *SIMAN* [Kelton, 98] são exemplos de algumas dessas linguagens. Entretanto, a maturidade na área da Simulação Digital está sendo atingida com a criação dos simuladores ou ambientes de simulação de alto nível.

Ambientes de simulação de alto nível, geralmente, disponibilizam conjuntos de componentes pré-codificados permitindo personalizar ou adicionar novos módulos. Além

disso, estes ambientes apresentam *interfaces* amigáveis e permitem a criação e a animação de modelos de sistemas de forma bastante flexível. São exemplos de ambientes de simulação: (i) de domínio público: *SAVAD* [Cabral, 92], *NS* [NS, 98] e *MARS* [Alaettinoglu, 98], e (ii) produtos comerciais: *Arena* [Kelton, 98] e o *BONeS* [Cadense, 98].

2.3 Classificação dos Modelos

Os modelos de um sistema podem ser classificados de várias formas. Um modelo pode ser estático se as mudanças de estado não envolvem o tempo ou dinâmico quando envolvem [Almeida, 99]. Com relação aos tipos de dados de entrada podem ser classificados em determinísticos ou estocásticos. No primeiro uma entrada válida tem um efeito preciso e determinado no estado do modelo ao passo que nos modelos estocásticos uma entrada válida pode resultar em várias mudanças de estados em vírtude dos processos randônicos existentes (eventos aleatórios). Por fim, um modelo pode ser classificado como discreto, quando as mudanças de estado ocorrem em pontos discretos do tempo, ou contínuo quando quando essas mudanças ocorrem de forma contínua no tempo [Soares, 90].

2.4 Classificação dos Simuladores

Os simuladores classificam-se de acordo com os tipos de modelos (discretos ou contínuos) com os quais trabalham e/ou quanto ao modo como a execução de tais modelos é feita. Com relação ao modo de execução do modelo, o simulador pode ser classificado como orientado a evento ou orientado a processo.

Um simulador é orientado a eventos quando o sistema é modelado pela definição das possíveis mudanças de estado que ocorrem no instante de cada evento e a sua execução é realizada pelo processamento da lógica associada a cada evento, em uma seqüência ordenada no tempo [Soares, 90] [Wagner, 00]. Dentre outras vantagens, esta abordagem permite conhecer o estado de qualquer entidade do sistema em qualquer instante de tempo.

Um simulador é orientado a processos quando sua execução se dá por meio de uma seqüência de processos onde cada um manipula um conjunto de eventos do mesmo tipo [Wagner, 00]. Nesta abordagem, a seqüência em que os eventos são tratados não é a mesma do sistema real e não há entidades dedicadas ao controle da simulação como acontece com a simulação orientada a eventos.

2.5 Simulação Discreta Orientada a Eventos

A simulação discreta utiliza modelos discretos, ou seja, modelos onde as mudanças de estado ocorrem em pontos discretos do tempo tais como as redes de computadores. Uma simulação discreta apresenta eventos, entidades e estados [Lula, 01].

Os eventos são atividades que mudam o estado do sistema. As entidades são partes de um sistema sendo vistas como os objetos representados na simulação, descritos pelos atributos e para os quais os eventos ocorrem. Atributos são as propriedades ou características de um sistema ou de uma entidade. O estado do sistema pode ser definido como a descrição de todas as entidades, atributos e atividades num dado ponto do tempo. Por fim, ambiente ou meio de um sistema é onde o sistema está inserido. Mudanças no seu ambiente podem afetar o estado do sistema.

Em uma simulação discreta, os eventos são geralmente gerenciados pelo uso de listas ou filas [Lula, 01]. Essas listas controlam a execução dos eventos e estes são gerados pelo movimento das entidades pelo sistema. Dessa forma, um evento pode representar a chegada de um cliente em um banco ou início da transmissão de uma mensagem em uma rede de computadores. Um evento é uma perturbação instantânea do sistema ocorrendo em um ponto determinado do tempo. As mudanças de estado no sistema ocorrem em função dos eventos fazendo com que o estudo destes seja essencial à compreensão do comportamento de um sistema.

A análise do desempenho de sistemas discretos, tais como redes de computadores, é feita observando o comportamento de um conjunto de medidas de desempenho relevantes de interesse, definidas a priori, frente às variações de demanda de utilização dessas medidas. Esta demanda é caracterizada pelo tipo de aplicação na entrada do sistema (tráfego) [Celestino, 90]. É preciso conhecer qualitativa e quantitativamente o desempenho dos sistemas com o intuito de procurar possíveis alternativas que possam levá-lo a um aumento de desempenho.

A simulação discreta objetiva a reprodução das atividades das entidades encontradas em um sistema para conhecer melhor o comportamento e o desempenho deste. O processamento seqüencial dos eventos e a coleta de valores nos tempos de evento fornecem à simulação um comportamento dinâmico [Soares, 90].

2.6 Ambientes de Simulação Orientados a Objetos

A expressão "orientação a objetos" pode ser analisada superficialmente indicando que o software é organizado como uma coleção de objetos separados que incorporam tanto a estrutura quanto o comportamento dos dados [Wagner, 00].

Na orientação a objetos os problemas são resolvidos usando modelos organizados em torno de entidades do mundo real ou de entidades imaginárias denominadas de objetos. O paradigma da orientação a objetos se baseia principalmente em quatro princípios: identidade, classificação, polimorfismo e herança [Rumbaugh, 91] [Almeida, 99].

- ✓ Identidade: Os dados são quantizados em entidades discretas e distinguíveis chamadas de objetos.
- ✓ Classificação: Objetos que apresentam a mesma estrutura de dados (atributos) e comportamento (operações) são agrupados em uma mesma classe e esta pode ser vista como uma abstração que descreve propriedades importantes a uma aplicação e ignora o resto.
- ✓ Polimorfismo: é a possibilidade da mesma operação poder se comportar de forma diferente em classes diferentes.
- ✓ Herança: é o compartilhamento de atributos e operações entre classes baseadas em um relacionamento hierárquico.

A capacidade de modelar sistemas usando conceitos inerentes ao mundo real possibilita ao analista de modelagem uma maior simplicidade e flexibilidade durante a concepção de seu modelo, permitindo, inclusive, reaproveitar código através do mecanismo de herança que os objetos oferecem [Almeida, 99].

Ambientes de simulação orientados a objetos (*Object Oriented Simulation – OOS*) têm sido alvo de intensas pesquisas nos últimos anos [Roberts, 94]. A principal razão é o aumento da demanda por sistemas de software cada vez mais complexos que são, por natureza, maiores e mais heterogêneos que outros sistemas além de, na maioria dos casos, serem desenvolvidos e mantidos por várias pessoas. Logo, quanto mais complexo o sistema for, mais benefícios se obterão utilizando *OOS* [Almeida, 99].

2.6.1 Linguagens e Ambientes OOS

Desde o início da criação da orientação a objetos, várias linguagens têm sido utilizadas para o desenvolvimento e a solução de modelos de simulação. A primeira linguagem orientada a objetos, *SIMULA*, por exemplo, foi criada objetivando a Simulação Digital. No entanto, outras de propósito geral, não orientadas a objetos, também têm sido utilizadas para tal fim como *FORTRAN*, *Pascal* e *C* por exemplo. Da mesma forma, linguagens de propósito geral para simulação estruturada (*GPSS, SIMScript* e *SLAM* [Kelton, 98]) também foram desenvolvidas. Mais recentemente surgiram as linguagens de programação de simulação orientada a objetos (linguagens *OOS* – Object Oriented Simulation).

As linguagens *OOS* podem ser classificadas de duas formas: (i) de propósito geral como *SimJava* [Howell, 97], *Silk* [Silk, 97] e *ModSim-III* [ModSim-II, 89], ou (ii) específicas como *G2*, *Taylor ED* e *Simple++* [Roberts, 94]. No primeiro caso, elas podem ser utilizadas para simular os mais diversos tipos de sistemas. No outro caso, são dirigidas a um domínio específico de aplicação.

A vantagem das linguagens e dos ambientes *OOS* sobre as linguagens e ambientes *OO* (orientado a objetos) de propósito geral, é que eles possibilitam a redução no tempo de desenvolvimento por possuírem mecanismos e classes específicos voltados à simulação.

A seguir encontra-se uma breve descrição de alguns ambientes de simulação.

a) **SAVAD**

O Sistema de Avaliação de Desempenho de Modelos de Redes de Filas (*SAVAD*) é um sistema especialista para avaliação de desempenho de modelos de redes de filas que objetiva solucionar modelos que exibem contenção de recursos, particularmente modelos que representam redes de computadores. Ele é um ambiente de modelagem integrado em que, a partir de um modelo de redes de filas proposto pelo usuário, este ambiente pode escolher e aplicar o método mais adequado para sua solução e fornecer medidas de desempenho relevantes. O *SAVAD* utiliza técnicas analíticas, baseadas na Teoria das Filas, e a técnica da Simulação Digital [Souto, 93] [Oliveira, 95].

O *SAVAD* foi desenvolvido usando as linguagens de programação *C++* [Zortech, 90] e *Prolog* [Clocksin, 81] em ambiente compatível com a arquitetura *IBM-PC*. As

características pelas quais foi projetado exploram diversos aspectos do paradigma da orientação a objetos facilitando assim, o desenvolvimento de versões futuras [Oliveira, 95].

Para a modelagem de sistemas de redes de filas, o ambiente fornece elementos que descrevem o comportamento do problema a ser analisado. Tais elementos são: clientes, estações de serviço, fontes, sorvedouros, pontos de controle, classes e rotas [Cabral, 92].

b) **NS**

O *NS* (*Network Simulator*) é um simulador de eventos discretos direcionado à pesquisa de redes de computadores [NS, 98]. Ele fornece suporte substancial para simulação de *TCP*, roteamento, e protocolos *multicast* em redes com ou sem fio.

Essa ferramenta de simulação surgiu como uma variante do *REAL Network Simulator* em 1989 e tem evoluído substancialmente ao longo dos últimos anos. O *NS* sempre recebeu contribuição de diversas instituições de pesquisa tecnológica e de estudantes. A versão 2 do *NS* foi lançada em 1997, no entanto, ainda não é um produto pronto pois apresenta muitos erros que são aos poucos descobertos e corrigidos muitas vezes pelos próprios usuários.

O *NS* é um simulador orientado a objetos escrito em *C++* que utiliza um interpretador *OTcl* como linguagem de modelagem dos cenários. O módulo *OTcl*, desenvolvido no *MIT* (*Massachusetts Institute of Technology*), é uma extensão das linguagens *Tcl/Tk* para a programação orientada a objetos. O simulador suporta uma hierarquia de classes em *C++* e uma hierarquia similar no interpretador. Os usuários criam novos objetos no interpretador e tais objetos possuem correspondência de um para um na hierarquia compilada.

Entre as principais características do *NS* destacam-se: classificação de pacotes por fluxos, ambiente extensível para a manipulação de filas e escalonamento, suporte estatístico e estrutura extensível de geração de tráfego, desenvolvido para as plataformas *Unix* (*FreeBSD*, *Linux*, *SunOS* e *Solaris*) e *Windows*.

c) **NIST**

O *NIST – National Institute of Standards and Technology*, órgão ligado ao Departamento de Comércio dos *E.U.A*, vem desenvolvendo desde 1995 um simulador de redes *ATM* [Golmie, 95], o *ATM Network Simulator* ou *NIST*, com o objetivo de dotar pesquisadores e projetistas de redes com uma ferramenta de análise, planejamento e avaliação

de redes *ATM*. O simulador permite ao usuário criar diferentes topologias de rede, estabelecer parâmetros de operação para cada componente, visualizar a atividade da rede durante a simulação e salvar os resultados em arquivo para posterior análise.

Entre as principais características dessa ferramenta de simulação destacam-se:

- ✓ Técnica de Simulação – O simulador do *NIST* utiliza a técnica de simulação orientada por eventos (*event-driven simulation*).
- ✓ *Interface* Gráfica – A *interface* gráfica do simulador é baseada no ambiente de janelas do *Windows*, que funciona sobre a plataforma *UNIX*.
- ✓ Linguagem usada no desenvolvimento – O simulador foi desenvolvido em linguagem de programação *C*.
- ✓ Nível de Simulação *ATM* – O simulador executa a simulação *ATM* no nível de células, o que permite acompanhar cada célula transmitida através da rede.

A versão atual do simulador não possui biblioteca de modelos. Entretanto, é possível a criação de novos componentes a partir da "recompilação" e "linkedição" do simulador.

d) **ARENA**

O software comercial *Arena* é um ambiente de simulação de propósito geral desenvolvido em 1993, para a plataforma *Microsoft Windows*, pela empresa norte americana *Systems Modeling Corporation* [Takus, 97] que é representada no Brasil pela empresa *Paragon Software*. O ambiente pode se integrar com outras ferramentas que fazem uso da tecnologia *ActiveX* tais como *Excel* e *Word.*

O ambiente *Arena* foi desenvolvido usando a linguagem *C++* , técnicas da orientação a objetos e recursos da *Microsoft Fundation Classes* (*MFC*). Isto possibilitou a construção de uma *interface* gráfica amigável para o ambiente contendo barras de ferramentas, caixas de diálogo, menus, etc. [Wagner, 00].

O *Arena* também gera código da simulação na linguagem *SIMAN* [Pegdrm, 95] o que permite a depuração de erros da lógica do modelo com facilidade [Takus 97]. Este ambiente permite combinar a facilidade de uso encontrada nos simuladores de alto nível com a flexibilidade da linguagem de simulação *SIMAN* e além disso, utilizar códigos de linguagens

de propósito geral, tais como *Visual Basic*, *FORTRAN* e *C/C++* [Wagner, 00]. Rotinas desenvolvidas em *C++* ou em *FORTRAN* podem ser acopladas ao ambiente, o que possibilita o desenvolvimento de tarefas complexas que não possam ou não sejam interessantes de serem representadas usando recursos de alto nível ou da linguagem *SIMAN*.

O *Arena* apresenta uma estrutura hierárquica baseada em *templates* que é um módulo composto por um conjunto de painéis que agrupam componentes. Os componentes, pertencentes a um mesmo *template* ou a *templates* distintos, podem ser combinados entre si, permitindo construir uma variedade maior de modelos [Kelton, 98].

O ambiente *Arena* encontra-se disponível em duas versões: Profissional e Acadêmica. A versão acadêmica é gratuita e se diferencia da profissional por não poder utilizar mais de 150 elementos em um modelo, além de não permitir a criação de novos *templates*. A versão profissional apresenta dois ambientes de trabalho [Wagner, 00]: o *Professional Edition* e o *Standard Edition*. O primeiro, permite a construção de novos *templates* pelo usuário. O segundo, é designado à construção, simulação e análise dos modelos do usuário.

Pode-se utilizar componentes tanto na construção de aplicações, como também na construção de outros componentes. O *Arena* apresenta um grande número de alternativas verticais para a construção de modelos, possibilitando ao usuário trabalhar com componentes específicos de alto nível, juntamente com a elaboração de rotinas em linguagens de propósito geral. O usuário também tem acesso a importantes recursos de simulação (suporte à decisão, animação gráfica e lógica de modelagem).

O trabalho apresentado nesta Dissertação diferencia-se dos demais trabalhos mostrados nesta seção por não se tratar de um ambiente de simulação específico e sim de componentes de software que poderão ser utilizados na construção de qualquer ambiente de simulação de modelos de sistemas que apresentam contenção de recursos (i.e., que podem apresentar filas), como é o caso das redes de computadores *TCP/IP*, *ATM*, sem fio, ou de outra tecnologia, ou mesmos sistemas que representam bancos, supermecados, sistemas de tráfego em geral, entre outros. A construção de um ambiente pode ser feita de forma rápida e simples, explorando os benefícios do uso de componentes. Adicionalmente, componentes necessários à construção de um modelo de rede *TCP/IP* também foram especificados e

implementados permitindo a construção de um ambiente de simulação de redes *TCP/IP*, validando assim os componentes propostos nesta Dissertação.

Capítulo 3

JavaBeans

Neste capítulo são descritos os principais aspectos da tecnologia utilizada na implementação dos componentes, o modelo de componentes *JavaBeans.* Ele também apresenta uma breve descrição da técnica da programação visual e de componentes de software de um modo geral. A leitura deste capítulo fornece uma base de conhecimento acerca do que consistem os componentes da tecnologia *JavaBeans*.

3.1 Introdução

A natureza portável da linguagem de programação *Java* e sua biblioteca de classes padrão possibilitam o desenvolvimento de uma aplicação que possa ser executada em qualquer sistema compatível com *Java,* sem a necessidade de recompilar ou adicionar lógica especial [Renshaw, 98]. Com isso, *Java* é um ambiente de execução independente de plataforma permitindo que uma aplicação seja criada uma vez e executada em qualquer lugar, possibilitando a reutilização de software. A reutilização de código atinge um alto nível de abstração com o uso de componentes de software reutilizáveis [Freire, 00] que podem ser vistos como objetos inteiros já instanciados que podem ser conectados a uma aplicação qualquer [Lula, 01].

Componentes de softwares reutilizáveis são projetados para aplicar o poder e o beneficio da reutilização vistos nas indústrias (principalmente na eletrônica) no campo da

construção de software. Com componentes, as aplicações são construídas visualmente através de ambientes ou ferramentas visuais que fornecem aos programadores flexibilidade ao permitir que eles reutilizem e integrem diferentes componentes existentes, que em muitos casos, não foram concebidos para serem utilizados em conjunto.

Os *JavaBeans* são componentes escritos em *Java*. Eles permitem que os desenvolvedores colham os benefícios do desenvolvimento rápido de aplicações em *Java* montando componentes predefinidos para criar aplicativos e *applets* poderosos [Deitel, 01]. Os *JavaBeans* são independentes de plataforma e portável uma vez que tiram proveito das vantagens que a linguagem *Java* oferece [Eckel, 00].

O uso de componentes no desenvolvimento de software leva a um novo tipo de programação, a "programação visual" e a um novo tipo de programador, o "montador de componentes".

3.2 Programação Visual

As linguagens de programação orientadas a objeto têm sido muito valiosas no campo da reutilização de software [Eckel, 00]. A classe é a unidade de código mais reaproveitada uma vez que compreende uma unidade coerente de características e comportamentos que podem ser reutilizados diretamente via composição ou através de herança. No entanto, o que realmente se quer são componentes de software que façam exatamente o que se deseja no programa de modo que possamos apenas "plugá-los" dentro da aplicação, assim como um engenheiro eletrônico adiciona chips a um circuito. Este tipo de programação em que componentes pré-definidos são inseridos no código do programa é conhecido como programação visual.

A programação visual através de componentes de software, levou a um novo tipo de programador, "o montador de componentes", que utiliza componentes bem definidos para criar funcionalidades mais robustas. Os montadores de componentes não precisam conhecer os detalhes de implementação de um componente, apenas os serviços fornecidos para que possam interagir com outros componentes, em geral, sem escrever nenhum código. Freqüentemente, os montadores estão mais preocupados com o projeto da *interface* gráfica com o usuário de um aplicativo ou com a funcionalidade que o aplicativo fornece para um usuário [Deitel, 01].

A programação visual, realmente, despontou a partir do surgimento de ferramentas como o *Visual Basic* da *Microsoft* e o *Delphi* da *Borland*. Nesses ambientes os componentes são representados visualmente, o que faz sentido, uma vez que eles geralmente representam algum tipo de componente visual, como um botão, ou um campo de texto, por exemplo. A representação visual, na verdade, é exatamente o modo como o componente aparecerá no programa. Assim, parte do processo neste tipo de programação envolve "puxar" um componente de uma lista e "soltá-lo" no local onde ele deve aparecer no programa. A ferramenta de desenvolvimento faz todo o código criando o componente quando o programa estiver processando.

Apenas "soltar" um componente em um formulário geralmente não é suficiente para completar um programa. Muitas vezes precisamos modificar características como cor, texto e etc. Características que podem ser modificadas em tempo de projeto são referenciadas como propriedades (mais detalhes na seção 3.4.2.c).

Na fase de projeto, os comportamentos de um componente são parcialmente representados pelos eventos, significando: "Aqui está algo que pode acontecer ao componente". Geralmente o montador de componentes decide o que deve acontecer quando um evento é gerado simplesmente "ligando" código ao evento em questão. No entanto é preciso conhecer quais as propriedades e os eventos que um componente suporta para que este possa ser configurado. Assim, a ferramenta de desenvolvimento visual utiliza um processo chamado de introspecção (mais detalhes na seção 3.4.2.d) para responder estas perguntas dinamicamente.

A razão pela qual as ferramentas de programação visual têm sido tão bem sucedidas é que elas dramaticamente aceleram o processo de construir uma aplicação [Eckel, 00].

3.3 Modelo de Componentes

Um modelo de componente especifica o conjunto de regras e de diretrizes que os componentes pertencentes ao modelo devem seguir [Lula, 01]. Ele é composto de uma arquitetura e de uma *API* (*Application Programing Interface*) [Englander, 97] que fornecem uma estrutura na qual os componentes podem ser combinados para criar uma aplicação. As características básicas que um modelo de componente devem suportar são: descoberta e

registro, geração e tratamento de eventos, persistência, representação visual e suporte a programação visual. A seguir essas características são explicadas mais detalhadamente.

3.3.1 Descoberta e Registro

O modelo de componentes deve fornecer meios para que se possa determinar as *interfaces* (conjunto de métodos públicos) que um componente suporta em tempo de execução [Lula, 01]. Esse processo de descoberta também pode ser feito em tempo de composição em um ambiente que suporte programação visual como o *Jbuilder* da *Borland* [Borland, 02]. O tempo de composição refere-se a etapa em que o "montador de componentes" ou programador está instanciando, através de uma ferramenta visual, os componentes que irá usar e conectando-os através de suas *interfaces*.

O modelo de componente da linguagem *Java*, os *JavaBeans*, realizam essa pesquisa através de um processo chamado de introspecção (mais detalhes na seção 3.4.2.d).

3.3.2 Geração e Tratamento de Eventos

Um evento pode ser descrito como uma perturbação instantânea no ambiente do sistema que provoca mudanças de estado ou simplesmente, um evento é algo importante que acontece em determinado ponto do tempo.

Um evento ocorre devido a uma ação, como o clique do mouse, um *thread* (um fluxo de execução de um programa) que gera um evento ou por outras maneiras. Quando um evento ocorre é preciso que os "interessados" no evento sejam notificados para que as reações pertinentes sejam executadas, assim, o modelo de componentes deve fornecer meios para que os componentes possam enviar notificações para outros objetos interessados. Este processo é descrito no modelo de eventos que o modelo de componentes suporta. Os *JavaBeans,* por exemplo, possuem um modelo de eventos que segue o padrão de projeto chamado *Observer* [Gamma et al., 95]. Mais detalhes sobre o modelo de eventos dos *JavaBeans* são descritos na seção 3.4.2.a.

3.3.3 Persistência

O modelo de componentes deve ser capaz de fornecer meios para salvar as informações correntes em um componente, ou seja, deve ser capaz de salvar o estado do componente para que o mesmo possa ser recriado mais tarde, de acordo com o seu estado

anterior. Portanto, os componentes devem ser capazes de participar de um mecanismo de persistência padrão à aplicação.

Os *JavaBeans* oferecem persistência através de um processo chamado de serialização de objetos. Esse processo é apresentado na seção 3.4.2.f.

3.3.4 Representação Visual

O modelo de componentes deve permitir que componentes individuais escolham os aspectos de sua representação visual, ou seja, a forma como eles serão exibidos na aplicação. A maioria dessas características é representada pelas propriedades do componente.

3.3.5 Suporte de Programação Visual

Programação visual é uma parte chave do modelo de componente [Lula, 01] porque não é necessário escrever nenhuma linha de código, basta interligar (geralmente através de eventos) e configurar os componentes visualmente para obter aplicativos, *applets* ou até mesmos outros componentes mais complexos.

3.4 O Modelo de Componentes JavaBeans

A linguagem de programação *Java* trouxe a criação de componentes ao seu mais avançado estado com a criação do seu modelo de componentes, os *JavaBeans*, porque esses componentes são apenas classes de *Java* [Eckel, 00]. Não é preciso escrever nenhum código extra ou usar extensões especiais da linguagem a fim de fazer algo se tornar um *JavaBean*. A única coisa que se precisa fazer é levemente modificar o modo que se nomeia os métodos (mais detalhes na seção 3.4.2.c). É o nome do método que informa à ferramenta de desenvolvimento se isto é uma propriedade, ou apenas um método comum.

Dessa forma, *JavaBeans* ou simplesmente *Beans* são classes que podem ser manipuladas numa ferramenta de desenvolvimento visual e "conectadas" numa aplicação. Qualquer classe *Java* que segue certas convenções de *interfaces* de eventos e de propriedades pode ser um *JavaBean*. Assim um *JavaBean* é um componente de software reutilizável que funciona com *Java* ou, mais especificamente, é um componente de software reutilizável que pode ser visualmente manipulado numa ferramenta de desenvolvimento [Renshaw, 98]. Simplesmente, é um componente de software que possibilita que desenvolvedores escrevam

componentes reutilizáveis uma vez e os utilizem em qualquer lugar aproveitando a portabilidade que a linguagem *Java* oferece [DeSoto, 97].

A importância dos *JavaBeans* é que alguns modelos de componentes como o *ActiveX* da *Microsoft,* por exemplo, não são independentes de plataforma. Os componentes *ActiveX* além de não serem escritos em *Java* são muito grandes e, em virtude da falta de portabilidade, não cobrem todo o mercado.

3.4.1 Objetivos do Modelo de Componentes JavaBeans

Quando proposto, o modelo de componentes da linguagem *Java* buscava alcançar os seguintes objetivos [Renshaw, 98]:

- ✓ Portável: escrito em *Java* sem nenhum código nativo da plataforma.
- ✓ Peso leve: é possível implementar um componente tão pequeno quanto um botão ou tão grande quanto um processador de texto.
- ✓ Simples de criar: deve ser simples criar um componente *JavaBean* sem a necessidade de implementar inúmeros métodos. A criação deve ser possível com ou sem a necessidade de ferramentas de desenvolvimento. Deve ser simples migrar de um simples *Applet* para um componente *JavaBean*.
- ✓ Aceitável em outros modelos de componentes: deve ser possível usar um componente *JavaBean* como um componente *ActiveX* como se fosse um componente deste modelo. Dessa forma, um componente de uma tabela (*JavaBean)* pode ser inserido num processador de texto para interagir com outros componentes, *ActiveX*, por exemplo, no mesmo aplicativo. O construtor dos *JavaBeans* não precisa fornecer uma lógica especial para lidar com os *JavaBeans* em outro ambiente. Além disso, os *JavaBeans* podem nem saber o que está acontecendo porque toda comunicação e conversão é fornecida por uma "ponte" (*bridge*).
- ✓ Capaz de acessar dados remotos: um componente *Java* pode usar qualquer um dos objetos distribuídos padrão (*Remote Method Invocation* por exemplo) ou mecanismos de dados distribuídos (*JDBC*) para acessar dados remotamente. De fato, os *JavaBeans* podem usar qualquer uma das facilidades dos ambientes padrões.

3.4.2 Principais Características dos JavaBeans

Os principais aspectos do modelo de componentes *JavaBeans* segundo [Szyperski, 99] são: eventos, métodos, propriedades, introspecção e configuração. As seções seguintes descrevem esses aspectos.

a) **Eventos**

Os *JavaBeans* se comunicam através da troca de eventos podendo ser fontes ou consumidores potenciais de eventos [Lula, 01]. A ferramenta de composição é responsável por conectar consumidores a fontes.

O modelo de eventos usado é o mesmo do *Swing* [Deitel, 01] da linguagem *Java*. Nesse modelo, eventos são objetos criados por uma fonte de eventos (*event source*) e propagado para todos os consumidores de eventos (*listeners*) cadastrados. Os consumidores se registram em fontes de eventos para receber notificações de eventos. A comunicação baseada em eventos geralmente é *multicast* ou seja, todos os *listeners* de um determinado evento são notificados quando o evento é gerado. Em uma comunicação *multicast,* para se registrar em uma fonte, os consumidores usam métodos padronizados cujas assinaturas são:

```
public void add<NomeDoEvento>Listener (<NomeDoEvento > Listener l);

public void remove< NomeDoEvento >Listener (<NomeDoEvento > Listener l);
```

Figura 3.1 – Métodos que Permitem o Cadastro/Descadastro de Classes Ouvintes no Componente para o Evento NomeDoEventoEvent (Multicast).

Além dos métodos mostrados na figura 3.1 é necessário fornecer ao componente o método que será responsável em gerar o evento para os *listeners*. A sua assinatura é mostrada na figura 3.2.

```
protected void fire<NomeDoEvento>(<NomeDoEventoEvent> e);
```

Figura 3.2 – Método Responsável pela Notificação das Classes Ouvintes de que o Evento NomeDoEventoEvent Ocorreu.

Uma comunicação *unicast* estabelece que apenas uma classe pode ser ouvinte de um evento. A figura 3.3 mostra as assinaturas de métodos para este tipo de comunicação.

```
public void add<NomeDoEventoListener>(<NomeDoEventoListener> l) throws
 TooManyListenersException;

public void remove<NomeDoEventoListener> (<NomeDoEventoListener> l);
```

Figura 3.3 – Métodos que Definem o Cadastro/Descadastro de Classes Ouvintes para um Evento (Unicast).

O modelo de componentes *JavaBeans* oferece dois tipos de eventos padronizados que são utilizados para informar aos seus *listeners* o estado de uma determinada propriedade: *PropertyChangeEvent* e o *VetoableChangeEvent* que se referem as propriedades amarradas e restritas, respectivamente. Essas propriedades são explicadas na subseção *ii* da seção *c* (Propriedades) apresentada mais adiante.

b) **Métodos**

Os métodos correspondem aos serviços que o "usuário" do componente pode utilizar. Quando uma classe é transformada em um componente *JavaBean* pode-se criar métodos padronizados que serão utilizados para definir as propriedades e/ou tratar eventos de forma que um ambiente de desenvolvimento possa "investigar", através de introspecção (mais detalhes na seção 3.4.2.d), quais são as propriedades, eventos e comportamentos do componente. Nesse caso, nem todos os métodos são métodos "comuns da classe".

c) **Propriedades**

As propriedades são as características (atributos) que podem ser modificadas em tempo de composição e que podem afetar a aparência ou comportamento do *JavaBean*. Quando uma propriedade muda pode ocorrer a geração de um evento resultante da mudança.

Os *JavaBeans* podem ter inúmeras propriedades. Elas são referenciadas por seu nome e podem ter qualquer tipo, incluindo tipos primitivos tal como *int* e tipos de classes ou *interfaces* tal como a classe *java.awt.Color* por exemplo. Propriedades são geralmente parte do estado persistente de um objeto, ou seja, seus valores devem ser salvos usando algum mecanismo de persistência (mais detalhes sobre persistência na seção 3.4.2.f).

O acesso às propriedades dos *JavaBeans* ocorre através de métodos padronizados para o tipo de propriedade em questão. Os métodos usados para acessar as propriedades

seguem o padrão de projeto para propriedades [Englander, 97] que será apresentado nas subseções *i* e *ii* desta seção.

As propriedades dos *JavaBeans* podem ser classificadas de acordo com o tipo do atributo a que elas se referem (simples ou indexadas) ou de acordo com as ações que podem ocorrer quando o valor da propriedade muda (amarradas ou restritas). As subseções *i e ii* a seguir descrevem os tipos de propriedades envolvidas em cada classificação.

i) **Quanto ao Tipo do Atributo**

Propriedades Simples

Uma propriedade simples representa um valor único e pode ser definida com um par de métodos *get/set*. O nome dos métodos é derivado do nome da propriedade. Para uma propriedade chamada *xxx,* deve-se criar dois métodos: *getXxx()* e *setXxx().* A primeira letra após *get* e *set* é automaticamente escrita em maiúscula para produzir o nome da propriedade. Os nomes de métodos *setX* e *getX* indica uma propriedade chamada "*X*". A figura 3.4 mostra o par de métodos *get/set* usados para definir a propriedade *NomeDaPropriedade*.

```
Public void setNomeDaPropriedade( TipoDeDados value );

Public TipoDeDados getNomeDaPropriedade();
```

Figura 3.4 – Par de Métodos que Definem a Propriedade NomeDaPropriedade.

A propriedade simples pode representar um valor único do tipo *boolean.* Nesse caso pode-se usar as regras dos métodos de acesso *get* e *set*, mas pode-se também usar *"is"* em vez de "*get*". Um nome de método *isX* por convenção indica que "*X*" é uma propriedade *booleana*. Se a propriedade é um tipo de dados *boolean*, o par de métodos *set/get* normalmente é definido de acordo com a figura 3.5.

```
Public void setNomeDaPropriedade( boolean value );

Public Boolean isNomeDaPropriedade();
```

Figura 3.5 – Par de Métodos que Definem uma Propriedade Simples Booleana.

Quando uma ferramenta de desenvolvimento examina um *JavaBean*, ela procura nos métodos do componente pares de métodos *set/get* que possam representar propriedades.

Propriedades Indexadas

Uma propriedade indexada representa um *array* de elementos. Os Métodos *set*/*get* referenciam os elementos individuais do *array* por meio de um índice inteiro. A propriedade também permite obter e modificar todos os valores do *array* de uma só vez. Os métodos que permitem esta operação são mostrados na figura 3.6.

```
public void setNomeDaColecao(int index, TipoArmazenado value);

public TipoArmazenado getNomeDaColecao (int index);

public void setPropertyName(TipoArmazenado [] value);

public TipoArmazenado[] getNomeDaColecao();
```

Figura 3.6 – Métodos que Definem uma Propriedade Indexada.

ii) **Quanto as Ações que Podem Ocorrer Resultantes da Mudança do Valor da Propriedade**

Propriedades Amarradas

Uma propriedade amarrada faz com que o objeto que a possui notifique outros objetos quando houver uma alteração no valor dessa propriedade [Deitel, 01]. Cada vez que seu valor muda, a propriedade gera um evento chamado *PropertyChangeEvent* que contém o nome da propriedade, valores antigos e novos. A granularidade da notificação é por *JavaBeans*, não por propriedade. Para suportar esse recurso, o pacote *java.beans* fornece:

- ✓ A *interface PropertyChangeListener* de modo que ouvintes possam ser configurados para receber notificações de alteração na propriedade;
- ✓ a classe *PropertyChangeEvent* para fornecer informações para um *PropertyChangeListener* sobre alterações no valor da propriedade; e
- ✓ a classe *PropertyChangesupport* para fornecer os serviços de registro e notificação de ouvinte.

Propriedades Restritas

Um objeto com este tipo de propriedade permite a outros objetos vetar uma mudança de valor nessa propriedade. As classes ouvintes de eventos registradas para este tipo de propriedade podem vetar uma mudança no valor da propriedade gerando uma exceção (*PropertyVetoException*).

d) **Introspecção**

Para que um componente *JavaBean* possa ser reutilizado nos ambientes de desenvolvimento deve haver um modo de "investigar" o que ele oferece em termos dos métodos que ele suporta e dos tipos de eventos que ele gera e/ou trata. Dessa forma quando um *JavaBean* é inserido na aplicação, a ferramenta de desenvolvimento deve ser capaz de instanciar o componente (que ela pode fazer se houver um construtor padrão) e então, sem ter acesso o seu código fonte, extrair todas as informações relativas às propriedades e eventos suportados pelo mesmo [Eckel, 00].

Na especificação dos *JavaBeans* este processo de "investigação" é nomeado de introspecção e é uma extensão do mecanismo de reflexão de *Java* que permite que todos os métodos públicos disponíveis e variáveis de uma classe anônima sejam descobertos através de pesquisas em tempo de execução [Renshaw, 98]. A reflexão de *Java* torna a linguagem perfeita para descobrir as propriedades e eventos de um componente *JavaBean*. Na verdade, uma das principais razões para que a reflexão fosse adicionada ao *Java* era para dar suporte aos *JavaBeans* (apesar da reflexão também suportar serialização de objetos e *Remote Method Invocation*).

Para os *JavaBeans* que não seguem os padrões de projeto desse tipo de componente ou para aqueles *JavaBeans* em que o programador quer personalizar o conjunto exposto de propriedades, métodos e eventos, o programador pode fornecer uma classe que implementa a *interface BeanInfo* (pacote *java.beans*) [Deitel, 01]. Nestes casos, os *JavaBeans* devem fornecer explicitamente informações sobre suas propriedades, métodos e eventos implementando a interface *java.beans.BeanInfo*. Essa *interface* especifica um conjunto de métodos que podem ser usados para recuperar vários elementos de informação sobre os *JavaBeans*. A classe *BeanInfo* é a classe que implementa a *interface BeanInfo* que é analisada pela ferramenta de desenvolvimento para expor, de acordo com o programador, explicitamente uma seleção dos métodos públicos e das variáveis do *JavaBean*.

Uma classe *BeanInfo* também é útil quando se reutiliza uma parte de código *Java* existente como um componente [Renshaw, 98]. A implementação já existente pode não suportar o *design pattern* esperado de um *JavaBean* e, até mesmo, não ter todas as capacidades necessárias, fazendo com que o mecanismo de introspecção *default* não identifique corretamente a *interface* pública do componente. Assim, o programador pode criar uma classe *BeanInfo* detalhando os nomes dos métodos que serão usados para obter ou mudar as propriedades sobrescrevendo, dessa forma, o esquema de introspecção *default*.

Por *default*, a classe *BeanInfo* tem o mesmo nome que o *JavaBean* e termina com *BeanInfo*. Além disso, ela é incluída no mesmo arquivo *JAR* (mais detalhes sobre o Arquivo *JAR* na seção 3.5.3.) que o *JavaBean* correspondente.

e) **Configuração**

A configuração de componentes *JavaBeans* é feita modificando o valor de suas propriedades em tempo de *design* (tempo de composição) ou seja, no momento em que o montador de componentes estiver instanciando e conectando os componentes através de suas *interfaces*.

f) **Persistência**

O estado de um componente *JavaBean* corresponde às informações que definem sua aparência e comportamento em um dado instante. É necessário que haja um meio de fazer com que um *JavaBean* tenha um comportamento preestabelecido quando este é executado. As propriedades representam algumas dessas informações. *JavaBeans* configurados e conectados necessitam serem salvos para futuro carregamento em tempo de execução da aplicação [Lula, 01].

Um esquema de persistência permite a um componente ser salvo para mais tarde ser recriado [Renshaw, 98]. O esquema de persistência da tecnologia *JavaBeans* usa o mecanismo de serialização de objetos. Desenvolvedores apenas precisam implementar a *interface java.io.Serializable* para tornar o componente persistente. Os campos em qualquer instância de um *JavaBean* que implementa esta *interface* são salvos. Pode-se impedir os campos de serem salvos marcando-os como *transient* ou estáticos; variáveis *static* e *transient* não são salvas. Geralmente os *JavaBeans* devem armazenar o estado de qualquer propriedade exposta [Eckel, 00].

3.5 Preparando uma Classe para ser um JavaBean

JavaBeans são classes *Java* que usam certas convenções para modificar a forma de seus métodos de modo que uma ferramenta visual possa manipulá-las como componentes. No entanto, antes de utilizar uma classe que representa um *JavaBean* ela deve ser empacotada e colocada em um arquivo do tipo *JAR*. Um único arquivo *JAR* pode conter muitos *JavaBeans*.

Dessa forma, partindo de uma classe *Java* as etapas referentes à criação de um *JavaBean* podem ser resumidas em 3 fases principais: (a) convenção de nomes dos métodos que representam as propriedades e os eventos; (b) Empacotamento dos *JavaBeans* e, (c) armazenamento do componente em um arquivo do tipo *JAR*.

3.5.1 Convenção de Nomes dos Métodos que Representam as Propriedades e os Eventos

A convenção utilizada para determinar as assinaturas dos métodos utilizados para definir as propriedades, o cadastro/descadastro de *listeners* e a geração de eventos já foi explicada nas seções 3.4.2.a (eventos) e 3.4.2.c (propriedades), respectivamente.

3.5.2 Empacotamento dos JavaBeans

Antes de colocar os componenetes *JavaBeans* um arquivo do tipo *JAR* eles precisam ser colocados em um pacote. Isto é feito fazendo com que a classe que define o componente contenha a instrução *package* no inicio do arquivo que a define. Essa instrução é seguida pelo nome do pacote onde se quer armazenar os componentes.

3.5.3 Armazenamento do Componente em um Arquivo do tipo JAR

JavaBeans são distribuídos através de arquivos do tipo *JAR* [DeSoto, 97]. O arquivo *JAR* é um arquivo compactado que pode conter arquivos *.class*, *JavaBeans* serializados (*.ser*), arquivos de ajuda em formato *HTML*, editores de customização, a classe *BeanInfo* e outros recursos (imagens, áudio, texto) necessários ao funcionamento do *JavaBean* [Freire, 00]. Se a versão serializada do componente existir no arquivo *JAR*, então ela é usada, ao invés de criar uma nova instância da classe [Renshaw, 98].

O arquivo *JAR* pode conter um arquivo especial, chamado de manifesto, mostrado na figura 3.7, que descreve todo o seu conteúdo. Um arquivo manifesto é simplesmente um

arquivo de texto que segue uma forma particular e deve ser criado antes de empacotar o *JavaBean* no arquivo *JAR*. Ambientes de desenvolvimento devem entender o formato *JAR* e usar as informações contidas no seu arquivo manifesto para identificar os *JavaBeans* contidos dentro de um *JAR*.

De acordo com o arquivo manifesto apresentado na figura 3.7, a primeira linha indica a versão do esquema de manifesto. A segunda linha (linhas vazias são ignoradas) nomeiam o arquivo *BangBean.class* e a terceira diz "isto é um *JavaBean*". Sem a terceira linha, a ferramenta de desenvolvimento não reconhecerá a classe como um *JavaBean*.

```
Manifest-Version: 1.0


Name: bangbean/BangBean.class
Java-Bean: True
```

Figura 3.7 – Formato do Arquivo Manifesto.

É possível ter em um arquivo *JAR* classes que não são *JavaBeans* [Deitel, 01]. Fornecer um arquivo manifesto permite especificar que classes são *JavaBeans* via, a entrada "*Java-Bean*: *True*", nesse arquivo. Caso contrário, os ambientes de desenvolvimento integrados (*IDEs*) não reconhecerão a classe como um *JavaBean*.

Se uma classe contendo o método "*main()"* é incluída em um arquivo *JAR*, essa classe pode ser utilizada pelo interpretador para executar o aplicativo diretamente do arquivo *JAR* especificando a propriedade "*Main-Class*" em uma linha isolada no inicio do arquivo manifesto. O nome completo do pacote e o nome completo da classe devem ser especificados com o ponto normal separando os nomes de pacote e o nome da classe.

Uma vez criado o arquivo *JAR* é possível trabalhar com o *JavaBean* em qualquer ferramenta de desenvolvimento que suporte *JavaBeans*. A *Sun* fornece uma ferramenta de testes totalmente gratuita, o *Bean Development Kit* (*BDK*) chamado de "*Beanbox*".

3.5.4 Beans Development Kit (BDK)

O *BDK* é uma aplicação de *Java* pura que fornece um *container* de teste (o "*BeanBox*" para testar o comportamento dos *JavaBeans*), *JavaBeans* de amostra completos com seu código fonte, a especificação *JavaBeans* e o tutorial [DeSoto, 97].

O BeanBox

O *BeanBox* é um utilitário da tecnologia *JavaBeans* para testar *JavaBeans* [Deitel, 01]. Ele é projetado para permitir que os programadores visualizem como será exibido e manipulado um *JavaBean* que eles criaram em uma ferramenta de desenvolvimento. Entretanto, ele não se destina a ser utilizada como uma ferramenta de desenvolvimento robusta. Ele lida com *JavaBeans* visíveis, aqueles que tem uma representação visual, e invisíveis, aqueles que são objetos puramente computacionais.

3.5.5 Ambiente de Desenvolvimento Integrado Jbuilder

O BeanBox é uma ferramenta direcionada para testes não oferecendo facilidades para o desenvolvimento visual de componentes. No entanto, existem ferramentas de desenvolvimento visual, a exemplo do *Jbuilder* (versão 5.0) da *Borland* [Borland, 02], que apresenta recursos essenciais para a construção e teste de componentes de software reutilizáveis.

Capítulo 4

Especificação dos Componentes

Neste capítulo é apresentada a especificação de componentes de software, mostrando com detalhes as fases de análise e de projeto, construídas utilizando um processo de desenvolvimento de software iterativo e incremental. Essa especificação teve como ponto de partida as especificações apresentadas em [Lula, 01] e em [Wagner, 00].

4.1 Introdução

Como ponto de partida para a implementação dos componentes foram adotadas as especificações apresentadas em [Lula, 01] e em [Wagner, 00]. Conforme mencionado no capítulo 1, o trabalho de [Wagner, 00] apresenta uma especificação de componentes de software que representam as funcionalidades mínimas de elementos para a modelagem de redes *TCP/IP* (*hosts, enlaces,* roteadores etc.), enquanto o trabalho de [Lula, 01], engloba a especificação de componentes apresentada em [Wagner, 00] adicionando novos componentes necessários à construção de um ambiente de simulação (escalonador de eventos, relógio, acumuladores estatísticos, entre outros).

Para viabilizar a fase de implementação foi observado que as fases anteriores do processo de desenvolvimento, a fase de análise e a de projeto nos trabalhos de [Wagner, 00] e de [Lula, 01], necessitavam desdobramentos. Nesse sentido, com base no processo iterativo e incremental descrito em [Larman, 98], as especificações propostas nos trabalhos referenciados

foram revistas e estendidas a fim de possibilitar a implementação satisfatória dos componentes.

É importante ressaltar que o procedimento de refinar artefatos de software, acrescentando novas funcionalidades, corrigindo eventuais problemas ou, até mesmo, simplificando os mesmos na fase de implementação, apesar de consumir bastante tempo é previsto em um processo iterativo e incremental em que o sistema ganha novas funcionalidades a cada iteração. Com isso, o sistema final será mais robusto possibilitando a reutilização não só de código, mas também das fases iniciais de análise e de projeto o que é um dos principais objetivos do presente trabalho.

As seções subseqüentes apresentam uma revisão da especificação dos componentes de software apontando os problemas encontrados a partir da adoção das especificações apresentadas em [Lula, 01] e em [Wagner, 00] bem como as soluções adotadas para resolvê-los. A revisão e conseqüente refinamento dessas especificações serão descritos de forma única como se fossem provenientes de um mesmo trabalho de especificação.

4.2 Especificação dos Componentes de um Ambiente de Simulação de Redes TCP/IP

Em [Lula, 01] foi apresentada uma especificação de componentes que permitem a construção de ambientes de simulação explorando a reutilização de software. O trabalho de especificação foi baseado no processo de desenvolvimento descrito em [Larman, 98] abrangendo as etapas de levantamento de requisitos, análise do domínio do problema e projeto da solução uma vez que o escopo do trabalho era apenas o de especificar os componentes. Como linguagem de modelagem foi utilizada a *UML* [Rumbaugh et al., 99].

Os componentes fornecem o funcionamento básico de uma simulação orientada a eventos, tais como controle do relógio simulado, geradores de valores aleatórios, listas de eventos e etc. Esses componentes atendem às fases clássicas de uma simulação: inicialização, execução e finalização. Um fluxograma do algoritmo de simulação em alto nível é mostrado na figura 4.1:

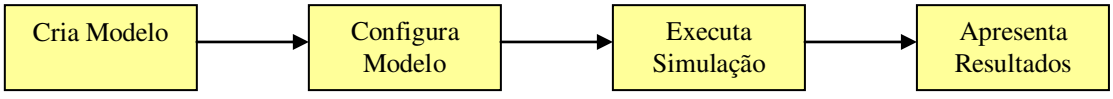

Figura 4.3: Algoritmo de Simulação em Alto Nível.

Na figura 4.1, a fase de inicialização da simulação foi dividida em duas etapas: *Cria Modelo* e *Configura Modelo*. Esta fase corresponde à criação do modelo a ser executado pelo ambiente e a conseqüente configuração dos parâmetros de entrada da simulação (por exemplo, tempos de início e de finalização da simulação). A criação do modelo, bem como a sua configuração pode ser feita através de uma *interface* (preferencialmente gráfica) do ambiente. Esta deve ser capaz de instanciar os componentes do modelo e de configurar as suas propriedades. Uma *interface* gráfica não faz parte do escopo deste trabalho.

A fase correspondente à execução é representada pela etapa *Executa Simulação*. Nessa etapa, eventos são gerados, armazenados em uma lista de eventos, em ordem cronológica, e escalonados segundo a disciplina de escalonamento *First In First Out* (*FIFO*), que significa que o primeiro evento a entrar deve ser o primeiro a sair. Após o escalonamento de um evento, procedimentos referentes às ações pertinentes à ocorrência deste evento, tais como atualização do relógio da simulação, coleta de dados para o cálculo de medidas de desempenho e ações que podem gerar outros eventos são executados. Na figura 4.2 é mostrado o algoritmo da figura 4.1 com a fase Executa simulação expandida.

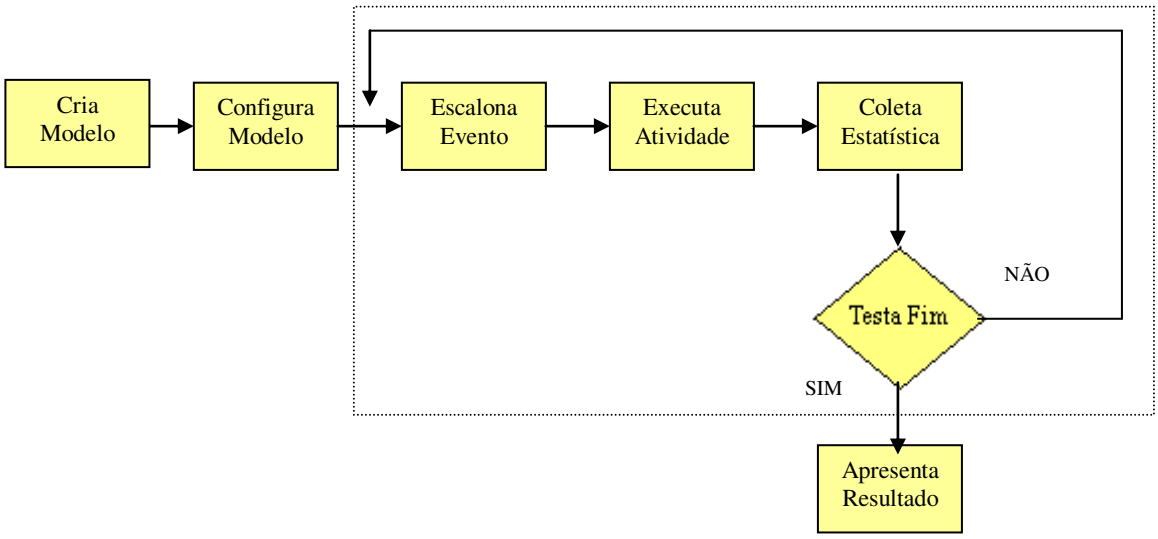

Figura 4.2: Algoritmo de Simulação com a Fase Executa Simulação Expandida.

Por fim, a última etapa do fluxograma da figura 4.1 corresponde à finalização da simulação, que se constitui no cálculo das medidas de desempenho de interesse e na apresentação dos resultados.

4.2.1 Levantamento de Requisitos

Na etapa de levantamento dos requisitos foi feita uma breve descrição do sistema, a identificação das suas metas, dos clientes, dos objetivos, dos requisitos funcionais (o que o sistema deve fazer) e dos não funcionais (características ou dimensões do sistema).

Os requisitos são uma descrição das necessidades ou desejos para um produto [Larman, 98]. Os requisitos do sistema podem ser classificados como Funcionais (o que o sistema deve fazer) e Não-Funcionais (atributos do sistema). Segundo [Larman, 98] as categorias de requisitos funcionais são:

Categoria	Significado
Evidente	Usuário do sistema está ciente de que a função está sendo feita
Escondida	Embora a função seja feita, ela é invisível ao usuário.
Opcional	Funcionalidade opcional; sua adição não afeta outras funções ou o custo de desenvolvimento significativamente

Tabela 4.1: Categoria dos Requisitos Funcionais.

Os Requisitos Funcionais levantados juntamente com sua classificação de acordo com a tabela 4.1 são:

F1 - O sistema deve permitir a construção e a simulação de modelos de redes *TCP/IP* tendo como referência os componentes especificados em [Wagner, 00] e [Lula, 01] (Evidente).

F2 – O sistema deve fornecer um mecanismo de coleta de dados para o cálculo de medidas de desempenho relevantes (Evidente). Essas medidas são definidas pelo ambiente de simulação.

F3 - A simulação é acionada por eventos. Um evento é uma "perturbação" instantânea que modifica o estado do sistema (Escondido). Os módulos do sistema devem responder à ocorrência de eventos. Os eventos devem ser escalonados (processados) segundo sua ordem cronológica de ocorrência.

F4 – Os elementos de modelagem devem ser configuráveis pelo usuário (Evidente).

F5 - O usuário define os parâmetros iniciais da simulação: condição de término (por tempo ou por alguma condição especificada pelo usuário), tempo inicial e número de replicações. (Evidente).

F6 - Um mesmo modelo pode ser simulado mais de uma vez, permitindo a obtenção de amostras de medidas de desempenho relevantes. (Evidente). As medidas de desempenho que devem ser coletadas pelo ambiente durante a simulação são:

- ✓ Tamanho dos pacotes no sistema – indica o tamanho dos pacotes processados pelos componentes *Roteador* e *Host*.
- ✓ Número de pacotes descartados – indica o número de pacotes descartados por cada componente *Roteador*, *Host* e *Enlace* durante a simulação.
- ✓ Fator de utilização – indica o fator de utilização dos componentes *Host, Roteador* e *Enlace*.
- ✓ Número de pacotes gerados – indica o número de pacotes gerados por cada componente *Fonte* durante a simulação.
- ✓ Tempo médio dos pacotes no sistema – indica o tempo médio que os pacotes passaram no sistema. Essa medida é obtida através de um acumulador estatístico que fornece o tempo mínimo, médio e máximo de permanência dos pacotes no sistema.
- ✓ Tempo de transmissão – indica o tempo de transmissão dos pacotes pelo componente *Enlace*. Essa medida é obtida através de um acumulador estatístico que fornece o tempo mínimo, médio e máximo de transmissão de todos os pacotes pelo componente *Enlace*.
- ✓ Atrasos de fila – indica o tempo que os pacotes tiveram de esperar nas filas dos componentes *Roteador* e *Host*. Essa medida é obtida através de um acumulador estatístico que fornece o tempo mínimo, médio e máximo de espera em fila para cada fila do componente.

F7 - O ambiente de simulação deve verificar a consistência do modelo antes de iniciar o processo de simulação propriamente dito (Escondido).

F8 – O ambiente de simulação deve gerar valores aleatórios conforme alguma função de distribuição de probabilidade conhecida (exponencial, uniforme, etc.), gerando automaticamente as sementes necessárias (Escondido).

F9 - O relógio é o mecanismo adotado que representa a evolução do tempo na simulação e representa um tempo "virtual" em que a simulação ocorre. Ele deve avançar de acordo com o tempo do evento sendo processado (Escondido).

F10 – O sistema deve ser capaz de possibilitar perdas de pacotes durante a simulação, devido aos *buffers* utilizados nas redes serem finitos (Evidente).

F11 – O sistema deve fornecer meios para que uma rota possa estar ativa ou não Quando ativa, pacotes são gerados e transitam pela rota. Esta facilidade permite definir um conjunto de rotas (ativas ou não) para um modelo sem a necessidade de remover as rotas inativas (Evidente).

Os requisitos não funcionais, também chamados de atributos do sistema, especificam restrições impostas à solução, como aquelas relacionadas com a adoção de padrões e a integração com outros sistemas: flexibilidade, facilidade de uso, portabilidade, etc [Freire, 00].

Os Requisitos Não-Funcionais são:

NF1 – A utilização de componentes permite o desenvolvimento rápido de novas ferramentas de simulação. A construção dessas ferramentas deve ser realizada visualmente através da composição dos componentes em um editor gráfico, exigindo o mínimo de programação.

NF2 – Os componentes fornecem abstrações de alto nível que são facilmente utilizáveis e extensíveis. Portanto, não deve ser necessário manipular estruturas de dados complexas nem utilizar características de uma linguagem particular que exijam considerável experiência do programador no momento de utilizar ou estender os componentes disponíveis.

NF3 – Os ambientes de simulação construídos a partir dos componentes especificados devem permitir a simulação de qualquer tipo de modelo discreto.

NF4 – Os componentes devem ser transportáveis para os principais ambientes operacionais em uso.

NF5 - Os documentos gerados na especificação dos componentes devem seguir uma linguagem padrão de modelagem (*UML*), permitindo sua reutilização futuramente, e de fácil compreensão.

Seguindo o processo de desenvolvimento adotado chegou-se à construção dos *Use Cases*, que são documentos narrativos que descrevem os processos que se encontram no domínio do problema. Os *Use Cases* são uma excelente forma de explorar e documentar os requisitos funcionais.

Na figura 4.3 é apresentado um Diagrama que ilustra todos os *Use Cases* descobertos, as suas relações, e os seus agentes externos que são chamados de atores. O *use case Executar Componentes* relaciona-se diretamente com o sistema a ser simulado (elementos de uma rede de computadores *TCP/IP*, por exemplo). A descrição detalhada de todos os *Use Cases* da figura 4.3 encontra-se em [Lula, 01].

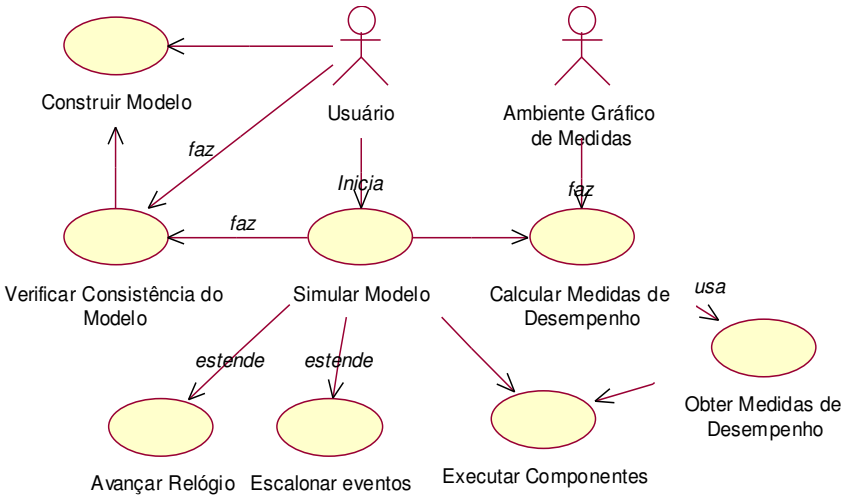

Figura 4.3: Diagrama de Use-Cases.

4.2.2 Fase de Análise

Após a fase de levantamento de requisitos, passou-se à fase de análise. Investigação e análise são freqüentemente caracterizadas por focarem questões do tipo qual – quais são os processos, os conceitos, os eventos e as operações [Larman, 98]. O modelo conceitual descrito no trabalho de [Lula, 01] apresentou-se incompleto necessitando portanto, ser expandido para que o mesmo pudesse corresponder, com maior fidelidade, ao sistema. O modelo resultante

dessa expansão foi dividido em dois sub-modelos a fim de facilitar o seu entendimento. Esses sub-modelos são mostrados nas figuras 4.4 e 4.5. Conforme mencionado anteriormente, os diagramas apresentam a integração dos componentes descritos em [Lula, 01] com os componentes para modelos de redes *TCP/IP* descritos em [Wagner, 00].

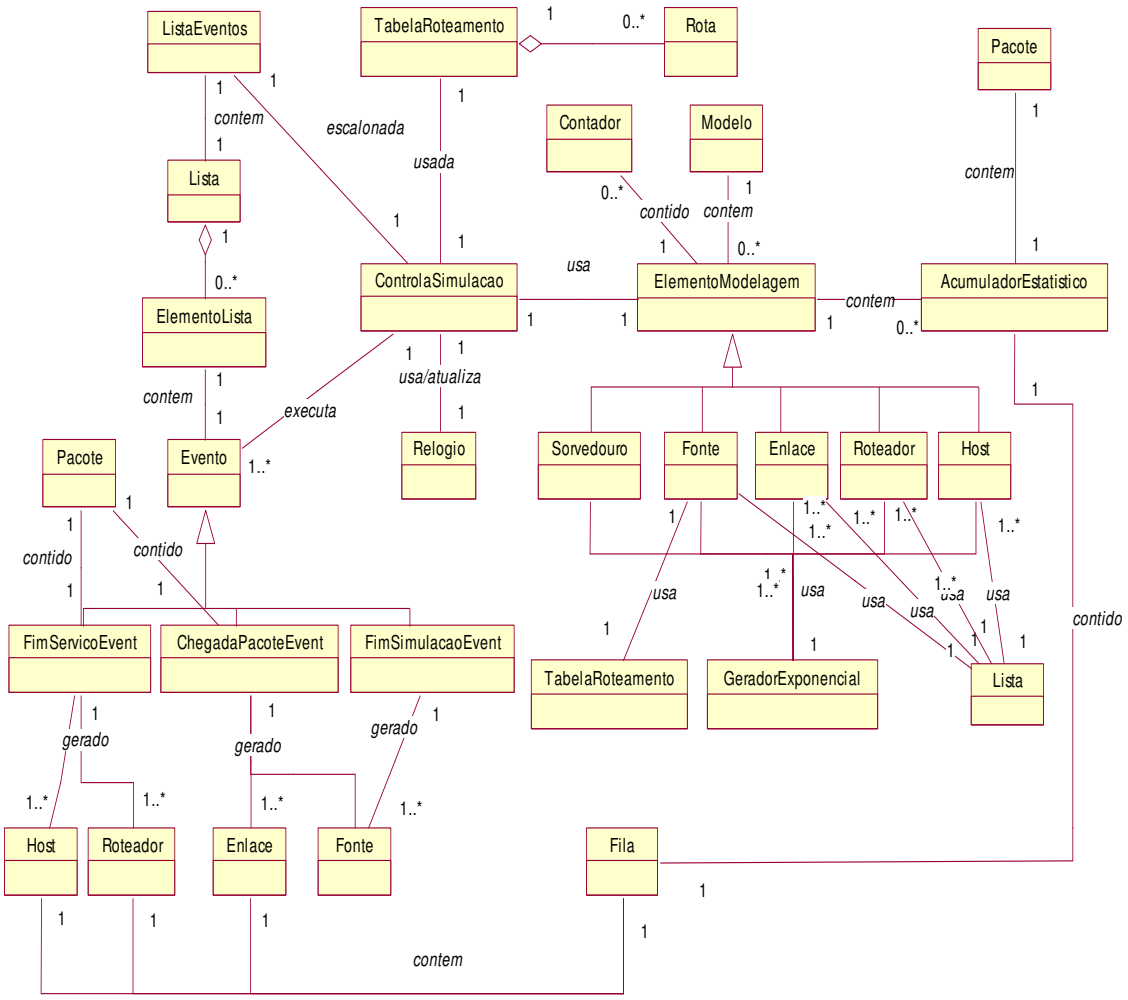

Figura 4.4: Diagrama de Conceitos Expandido – Primeira Parte.

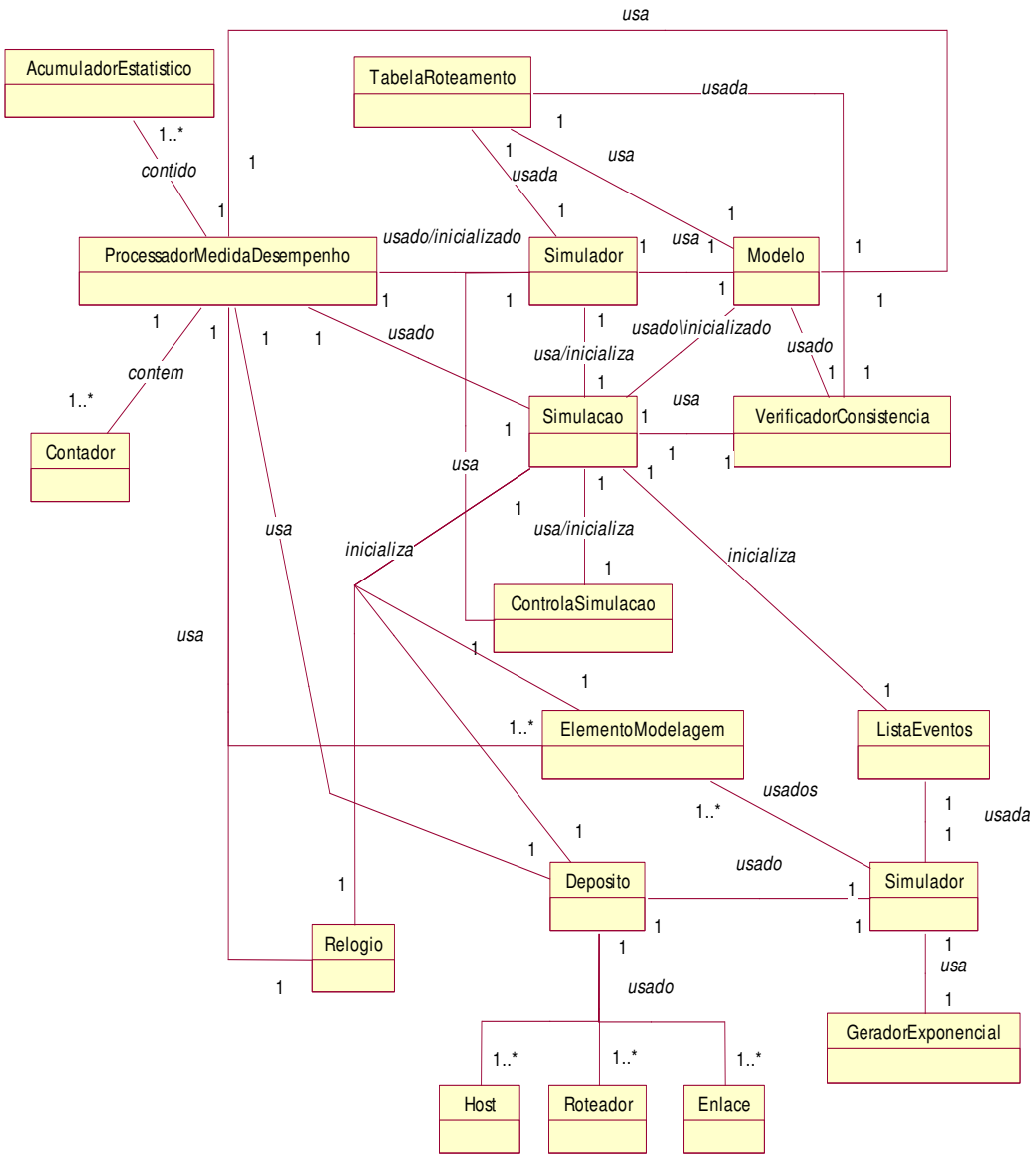

Figura 4.5: Diagrama de Conceitos Expandido – Segunda Parte.

Os diagramas mostrados nas figuras 4.4 e 4.5 não só são mais detalhados como também englobam novos componentes não previstos inicialmente na fase de análise tais como a *TabelaRoteamento*, a *Fila*, a *Lista*, etc. Além disso, outros componentes inicialmente propostos deixaram de existir (componente *Mensagem*) ou evoluíram para outros mais coerentes com sua função como o *GeradorVAs* (gerador de valores aleatórios) que foi substituído pelo *GeradorExponencial* (gerador exponencial).

É importante lembrar que a expansão do diagrama ocorreu na fase de implementação como forma de facilitar o entendimento do sistema uma vez que o diagrama original, conforme [Lula, 01], apresenta-se incompleto em alguns aspectos. Dessa forma, o novo modelo de conceitos, ora apresentado, pode ser considerado um artefato que verdadeiramente representa os componentes e classes do sistema e não apenas "candidatos" aos componentes ou as classes.

Para sistemas dinâmicos somente o modelo conceitual pode não ser suficiente para que o sistema seja realmente compreendido. Para isso, a *UML* dispõe de diagramas de interação cujas finalidades são a de mostrar o comportamento dinâmico do sistema. Esses diagramas ilustram como os objetos interagem através de mensagens (eventos ou métodos) para cumprir suas tarefas. A *UML* define dois tipos de diagrama de interação: (a) diagrama de seqüência e (b) diagrama de colaboração.

Conforme mencionado, os objetos representados através de um diagrama de colaboração se comunicam através de mensagens que podem ser vistas como eventos. No nosso caso, tais objetos são instanciações de componentes que usam o padrão de projeto *Observer* [Gamma et al, 95] para se comunicar. Segundo esse padrão, os eventos são objetos que podem conter informações que serão usadas pelos objetos que irão tratá -los. Com o intuito de facilitar o entendimento dos conceitos inerentes à comunicação de componentes, bem como o entendimento dos próprios diagramas de colaboração, optamos por representar as mensagens através dos nomes dos eventos que os objetos devem tratar e não através dos métodos das *interfaces* ouvintes para o evento em questão. Por exemplo, ao invés de representar a comunicação entre os objetos *Simulador* e *Simulacao* através do método *inicializaSimulacao(),* colocamos o nome do próprio evento que é tratado por este método: o *InicializaSimulacaoEvent*. Esta abordagem tem a vantagem de enfatizar que a comunicação é entre componentes que seguem o modelo de componentes *JavaBeans* (Capítulo 3), onde o evento é um *InicializaSimulacaoEvent* e o objeto que recebe um *InicializaSimulacaoListener*.

Para um melhor entendimento dos diagramas de colaboração gerados, a seguir é apresentada uma descrição de todos os eventos que viabilizam a interação entre os componentes do ambiente e entre estes com os componentes do modelo evidenciando que toda a comunicação é baseada em eventos apesar dos componentes do modelo serem instanciados após a construção e configuração do ambiente.

- ✓ *InicializaSimulacaoEvent* – Evento gerado pelo componente *Simulador* que inicializa o processo da simulação.
- ✓ *VerificaModeloEvent* – Evento gerado pelo componente *Simulacao* que aciona o componente *VerificadorConsistencia* notificando que este verifique a consistência do modelo a ser simulado.
- ✓ *ObterRotasAtivasEvent* – Evento gerado pelo componente *VerificadorConsistencia* para obter as rotas ativas do modelo que devem ser verificadas.
- ✓ *RetornaRotasAtivasEven*t – Evento gerado pelo componente *TabelaRoteamento* em resposta ao evento *ObterRotasAtivasEvent.* Esse evento possui as rotas ativas do modelo.
- ✓ *RetornaModeloEvent* – Evento gerado pelo componente *Modelo* que contém as rotas ativas e os elementos de modelagem que fazem parte do modelo.
- ✓ *ModeloVerificadoEvent* – Evento gerado pelo componente *VerificadorConsistencia* a fim de notificar os componentes interessados que o modelo já foi verificado. Este evento indica se o modelo está correto ou não.
- ✓ *InicializaEvent* – Evento gerado pelo componente *Simulacao* apenas no caso em que o modelo está correto.
- ✓ *ExecutaControlaSimulacaoEvent* – Evento gerado pelo componente *Simulacao* para dar inicio a simulação através da execução do componente *ControlaSimulacao*.
- ✓ *ExecutaFonteEvent* – Evento gerado pelo componente *ControlaSimulacao* para que as fontes ativas do modelo sejam executadas.
- ✓ *ExecutaFonteNovamenteEvent* – Evento gerado pelo componente *Host* (origem) que aciona o componente *ProcessadorMedidasDesempenh*o notificando que a fonte deve ser executada novamente com base no identificador que é passado como parâmetro.
- ✓ *ObterRotaEvent* – Evento gerado pelo componente *Fonte* (subclasse de *ElementoModelagem*) apenas na sua primeira execução para obter a sua rota.

- ✓ *RetornaRotaEvent* – Evento gerado pelo componente *TabelaRoteamento* em resposta ao evento *ObterRotasAtivasEvent*. Este evento possui a rota para qual a fonte deve gerar os pacotes.
- ✓ *InsereRotaEvent* – Evento gerado pelo componente *Simulador* que aciona o componente *TabelaRoteamento* notificando que uma nova rota deve ser inserida na sua lista. Este evento contém a rota a ser inserida.
- ✓ *RemoveRotaEvent* – Evento gerado pelo componente *Simulador* que aciona o componente *TabelaRoteamento* notificando que a rota cujo identificador é passado como parâmetro deve ser removida da sua lista.
- ✓ *TornaRotaAtivaEvent* – Evento gerado pelo componente *Simulador* que aciona o componente *TabelaRoteamento* notificando que este deve tornar uma de suas rotas ativas.
- ✓ *TornaRotaInativaEvent* – Evento gerado pelo componente *Simulador* que aciona o componente *TabelaRoteamento* notificando que este deve tornar uma de suas rotas inativas.
- ✓ *ObterDadosGeradorExponencialEvent* – Evento gerado pelos componentes que necessitam obter amostras de uma função de distribuição exponencial.
- ✓ *RetornaDadosGeradorEvent* – Evento gerado pelo componente *GeradorExponencial* em resposta ao evento *ObterDadosGeradorExponencialEvent.* Este evento possui o resultado dos cálculos produzidos pelo gerador exponencial.
- ✓ *ChegadaPacoteEvent* – Evento gerado pelos componentes *Fonte* e *Enlace* (subclasses de *ElementoModelagem*) para indicar a chegada de um pacote. Este evento contém o pacote que será transmitido para o próximo elemento da rede de acordo com a tabela de roteamento.
- ✓ *FimServicoEvent* – Evento gerado pelos componentes *Host* e *Roteador* (subclasses de *ElementoModelagem*) para indicar o fim de serviço. Este evento contém o pacote que será transmitido para o próximo elemento da rede de acordo com a tabela de roteamento.

- ✓ *FimSimulacaoEvent* – Evento gerado pelo componente *Simulacao* que aciona o componente *Simulador* notificando que este deve receber os resultados coletados durante a simulação.
- ✓ *StatusListaEventosEvent* – Evento gerado pelo componente *ListaEventos* que aciona o componente *ControlaSimulacao* notificando a respeito do número de eventos que essa lista armazena.
- ✓ *EscalonaListaEventoEvent* – Evento gerado pelo componente *ControlaSimulacao* que aciona o componente *ListaEventos* notificando que um evento deve ser escalonado.
- ✓ *RetornaEventoEvent* – Evento gerado pelo componente *ListaEventos* em resposta ao evento *EscalonaListaEventoEvent*. Este evento possui o evento que foi escalonado da lista de eventos do sistema.
- ✓ *AtualizaRelogioEvent* – Evento gerado pelo componente *ControlaSimulacao* que aciona o componente *Relogio* notificando que este atualize o seu tempo de acordo com o tempo que é passado no evento.
- ✓ *RetornaRelogioTempoEvent* – Evento gerado pelo componente *Relogio* em resposta ao evento *AtualizaRelogioEvent.* Ele aciona os componentes *ProcessadorMedidasDesempenho* e o *ControlaSimulacao* notificando-os para que seus tempos de simulação sejam atualizados.
- ✓ *TempoCorrenteRelogioEvent* – Evento gerado pelo componente *ControlaSimulacao* que aciona os componentes *Host, Enlace* e *Roteador* notificando sobre o valor do tempo corrente da simulação.
- ✓ *ObterProximoNoEvent* – Evento gerado pelo componente *ControlaSimulacao* que aciona o componente *TabelaRoteamento* notificando que o identificador do próximo elemento da rede seja retornado.
- ✓ *RetornaProximoNoEvent* – Evento gerado pelo componente *TabelaRoteamento* em resposta ao evento *ObterProximoNoEvent.* Este evento contém o identificador do próximo elemento da rede para o qual o pacote deve ser enviado.

- ✓ *RecebePacoteEvent* – Evento gerado pelo componente *ControlaSimulacao* a fim de acionar os componentes que representam os elementos de rede, *Host* e *Roteador*, de que um pacote deve ser processado.
- ✓ *TransmiteEvent* – Evento gerado pelo componente *ControlaSimulacao* que aciona os componentes que representam os elementos de rede, *Enlace* e *Sorvedouro,* notificando-os de que um pacote deve ser transmitido.
- ✓ *ExecutaFimSimulacaoEvent* – Evento gerado pelo componente *ControlaSimulacao* após o término de uma replicação para que as medidas de desempenho possam ser processadas.
- ✓ *ObterResultadoSimulacaoEvent* – Evento gerado pelo componente *Simulacao* após o fim da simulação para obter os resultados processados pelo componente *ProcessadorMedidasDesempenho.*
- ✓ *RetornaResultadoSimulacaoEvent* – Evento gerado pelo componente *ProcessadorMedidasDesempenho* em resposta ao evento *ObterResultadoSimulacaoEvent.* Este evento contém os resultados coletados durante a simulação.
- ✓ *InsereElementoModelagemEvent* – Evento gerado pelo componente *Simulador* que aciona o componente *Modelo* notificando que um novo elemento de modelagem deve ser incluído na sua lista de elementos.
- ✓ *RemoveElementoModelagemEvent* - Evento gerado pelo componente *Simulador* que aciona o componente *Modelo* notificando que o elemento de modelagem cujo identificador é passado como parâmetro deve ser excluído da sua lista de elementos.
- ✓ *CadastraEvent* – Evento gerado pelo componente *Simulador* para indicar que o elemento de modelagem passado como parâmetro deve ser cadastrado nos componentes do ambiente (*Simulacao, TabelaRoteamento, ProcessadorMedidasDesempenho, GeradorExponencial, ControlaSimulacao, ListaEventos e Deposito*) para que a interação entre eles possa ocorrer.
- ✓ *DescadastraEvent* – Evento gerado pelo componente *Simulador* para indicar que o elemento de modelagem passado como parâmetro deve ser descadastrado

dos componentes do ambiente (*Simulacao, TabelaRoteamento, ProcessadorMedidasDesempenho, GeradorExponencial, ControlaSimulacao, ListaEventos e Deposito*).

- ✓ *DescartaPacoteEvent* – Evento gerado pelos componentes *Host, Roteador* e *Enlace* que aciona o componente *Deposito* notificando que um pacote deve ser descartado.
- ✓ *ObterDadosSorvedouroEvent* – Evento gerado pelo componente *ProcessadorMedidasDesempenho* que aciona o componente *Sorvedouro* notificando que este deve fornecer os seus dados que foram coletados durante a simulação.
- ✓ *RetornaDadosSorvedouroEvent* – Evento gerado pelo componente *Sorvedouro* em resposta ao evento *ObterDadosSorvedouroEvent.* Este evento contém os dados coletados pelo componente *Sorvedouro* durante a simulação.
- ✓ *ObterDadosFonteEvent* - Evento gerado pelo componente *ProcessadorMedidasDesempenho* que aciona o componente *Fonte* notificando que este deve fornecer os seus dados que foram coletados durante a simulação.
- ✓ *RetornaDadosFonteEvent* – Evento gerado pelo componente *Fonte* em resposta ao evento *ObterDadosFonteEvent.* Este evento contém os dados coletados pelo componente *Fonte* durante a simulação.
- ✓ *ObterDadosDepositoEvent* - Evento gerado pelo componente *ProcessadorMedidasDesempenho* que aciona o componente *Deposito* notificando que este deve fornecer os seus dados que foram coletados durante a simulação.
- ✓ *RetornaDadosDepositoEvent* – Evento gerado pelo componente *Deposito* em resposta ao evento *ObterDadosDepositoEvent.* Este evento contém os dados coletados pelo componente *Deposito* durante a simulação.
- ✓ *ObterDadosHostEvent* - Evento gerado pelo componente *ProcessadorMedidasDesempenho* que aciona o componente *Host* notificando que este deve fornecer os seus dados que foram coletados durante a simulação.

- ✓ *RetornaDadosHostEvent* – Evento gerado pelo componente *Host* em resposta ao evento *ObterDadosHostEvent.* Este evento contém os dados coletados pelo componente *Host* durante a simulação.
- ✓ *ObterDadosRoteadorEvent* - Evento gerado pelo componente *ProcessadorMedidasDesempenho* que aciona o componente *Roteador* notificando que este deve fornecer os seus dados que foram coletados durante a simulação.
- ✓ RetornaDadosRoteadorEvent – Evento gerado pelo componente *Roteador* em resposta ao evento *ObterDadosRoteadorEvent.* Este evento contém os dados coletados pelo componente *Roteador* durante a simulação.
- ✓ *ObterDadosEnlaceEvent* - Evento gerado pelo componente *ProcessadorMedidasDesempenho* que aciona o componente *Enlace* notificando que este deve fornecer os seus dados que foram coletados durante a simulação.
- ✓ *RetornaDadosEnlaceEvent* – Evento gerado pelo componente *Enlace* em resposta ao evento *ObterDadosEnlaceEvent.* Este evento contém os dados coletados pelo componente *Enlace* durante a simulação.

A figura 4.6 mostra o diagrama de seqüência para a operação de sistema *executa().* Essa operação de sistema é a mais importante uma vez que compreende a execução da simulação propriamente dita. Essa operação foi gerada pelo sistema em resposta ao evento de sistema *executa()* que faz parte do *Use Case Simular Modelo* descrito no trabalho de [Lula, 01].

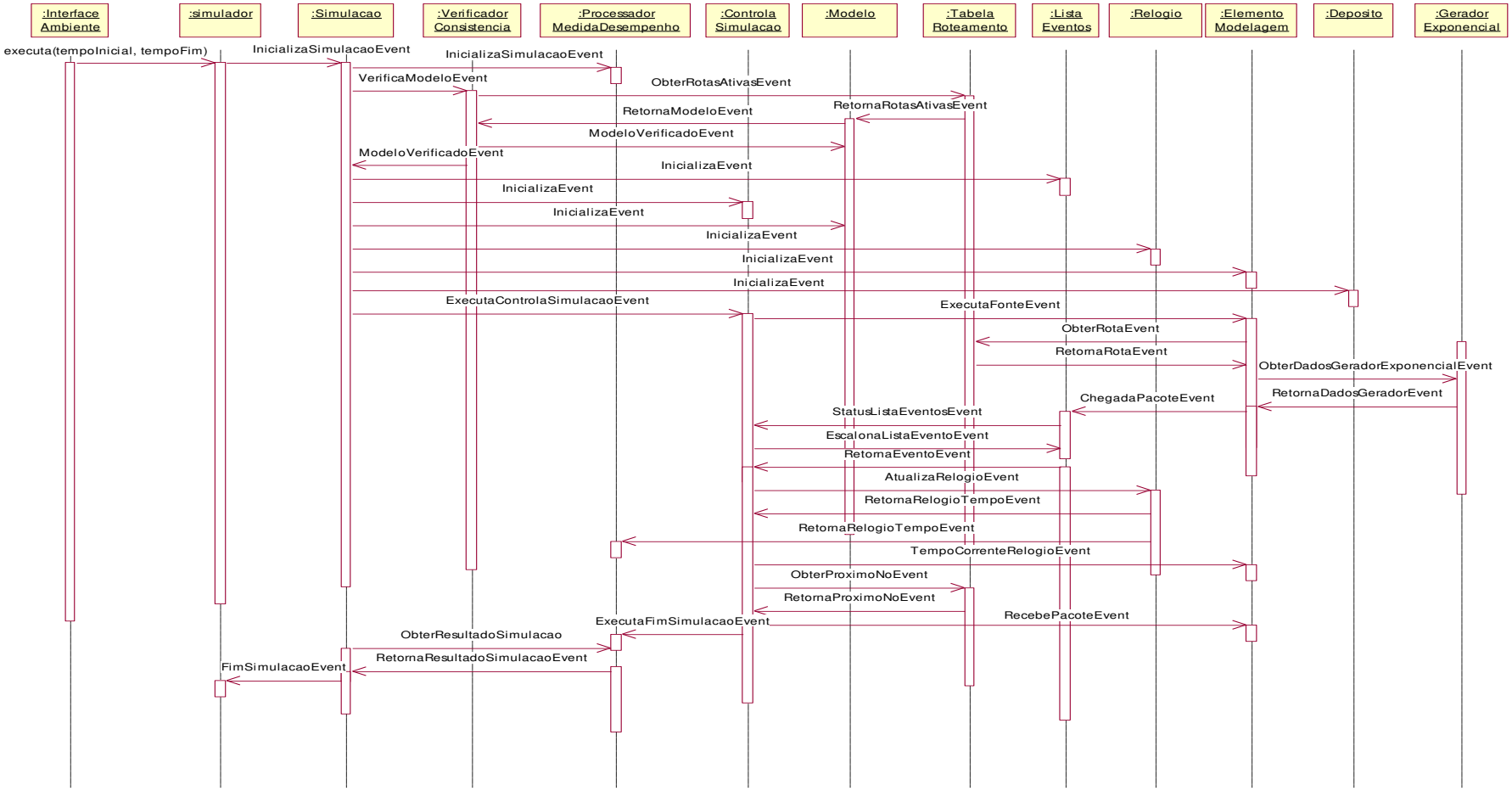

Figura 4.6: Diagrama de Seqüência para a Operação de Sistema executa( ).

É importante destacar, para uma melhor compreensão do leitor, que quando um modelo de redes *TCP/IP* for simulado os componentes do ambiente de simulação tais como o *Relogio*, *ListaEventos* etc., já terão sido instanciados, configurados com os valores iniciais e "conectados" através de eventos gerando, assim, um software pronto que, munido de uma *interface* (preferencialmente gráfica), constituirá a aplicação completa. No entanto, para que o ambiente de simulação funcione é preciso criar um modelo de rede *TCP/IP* através da sua *interface* e executar a simulação do mesmo. Tal procedimento é descrito através do algoritmo do fluxograma da figura 4.1. Nesse caso, o processo de inicialização que ocorre dentro da etapa de execução da simulação, através do evento *InicializaEvent* (figura 4.6), não corresponde a criação do modelo e configuração dos seus elementos de modelagem tais como *hosts*, roteadore*s*, *enlaces* etc., e sim, a inicialização de parâmetros como tempo inicial da simulação, tempo final, número de replicações etc., necessários aos componentes do ambiente para que os mesmos possam realizar as suas funções. Além disso, esse processo de inicialização zera os acumuladores estatísticos e contadores de todos os componentes para que uma nova simulação possa ocorrer. Tal procedimento é importante pois permite que o modelo seja executado várias vezes, não necessitando de intervenção do usuário para configurar os componentes com valores iniciais a cada replicação.

No diagrama de seqüência da figura 4.6 alguns aspectos da dinâmica do sistema foram omitidos em prol da simplificação, mas vale observar os seguintes pontos:

a. Neste momento da simulação, todos os componentes do modelo já foram configurados e conectados aos componentes do ambiente.

b. A simulação inicia de fato quando o componente *Fonte* começa a gerar pacotes de acordo com amostras da função de distribuição de probabilidade fornecida pelo usuário. Essa geração é feita de forma automática durante toda a simulação. No caso de um modelo determinístico o gerador de valores aleatórios (*GeradorExponencial,* por exemplo) não é acionado e o valor do intervalo de tempo fornecido pelo usuário permanece constante durante toda a simulação.

c. Os pacotes transitando pelos elementos da rede também geram eventos que são inseridos na lista de eventos se forem do tipo *FimSimulacaoEvent*, *ChegadaPacoteEvent* ou *FimServicoEvent*. Os pacotes são inseridos dentro dos

eventos *ChegadaPacoteEvent* e *FimServicoEvent* como mostra o diagrama de conceitos da figura 4.4. Vale lembrar que tais eventos são classes em *Java* que podem armazenar informações relevantes ao evento em questão, como o próprio pacote que está sendo transmitido.

d. No diagrama de seqüência só foi mostrado a geração do primeiro pacote pelo componente *Fonte* e a conseqüente execução do evento associado a ele, no caso, o *ChegadaPacoteEvent*. Na segunda vez que o componente *ControlaSimulacao* escalonar um evento este poderá ser um evento gerado por uma fonte (*ChegadaPacoteEvent*), um evento relacionado com o trânsito do pacote no modelo (*FimServicoEvent* por exemplo) ou o evento que indica o fim da simulação (*FimSimulacaoEvent*).

e. Por fim, conforme ressaltado anteriormente, a inicialização dos parâmetros de alguns componentes ocorre imediatamente antes da execução da simulação propriamente dita. Se o modelo precisar de uma nova replicação, a inicialização ocorrerá de forma automática. No entanto, se o usuário executar o modelo novamente, será possível passar novos valores iniciais para os elementos de modelagem.

De acordo com o diagrama de seqüência da figura 4.6, após construir e configurar o modelo, o usuário inicializa a simulação. O componente *ControlaSimulacao* antes de, efetivamente, começar a simular, executa a verificação de consistência do modelo. Estando o modelo correto, inicia-se o processo de escalonamento de eventos, a atualização do relógio simulado e a execução das ações associadas às ocorrências dos eventos. Essas ações também dizem respeito à execução dos elementos do modelo, no caso do presente trabalho, dos elementos que compõem uma rede *TCP/IP*.

Quando um elemento de rede é executado, dados são coletados visando a obtenção de medidas de desempenho de interesse e, eventualmente, são gerados novos eventos que são inseridos na lista de eventos.

Por fim, o escalonamento do evento *FimSimulacaoEvent* indica que a simulação chegou ao seu fim. A próxima etapa consiste nos cálculos das estatísticas utilizando os dados coletados durante a simulação.

4.2.3 Fase de Projeto

A fase de projeto é uma extensão da fase de análise, visando a implementação do sistema em um computador. A fase de projeto concentra-se na questão "como?". Durante a fase de projeto são identificados os componentes que fazem parte do sistema.

4.2.3.1 Projeto Arquitetural ou Projeto de Alto Nível

Um dos artefatos gerados nessa fase do processo de desenvolvimento é o projeto arquitetural. A sua descrição é ilustrada na figura 4.7 através de uma arquitetura de 3 camadas, que são: (i) Camada de Apresentação: define a *interface* com o usuário; (ii) Camada de Aplicação: define a lógica da aplicação, isto é, define as tarefas e regras que governam o processo, e (iii) Camada de Dados: consiste nos mecanismos de armazenamento persistente.

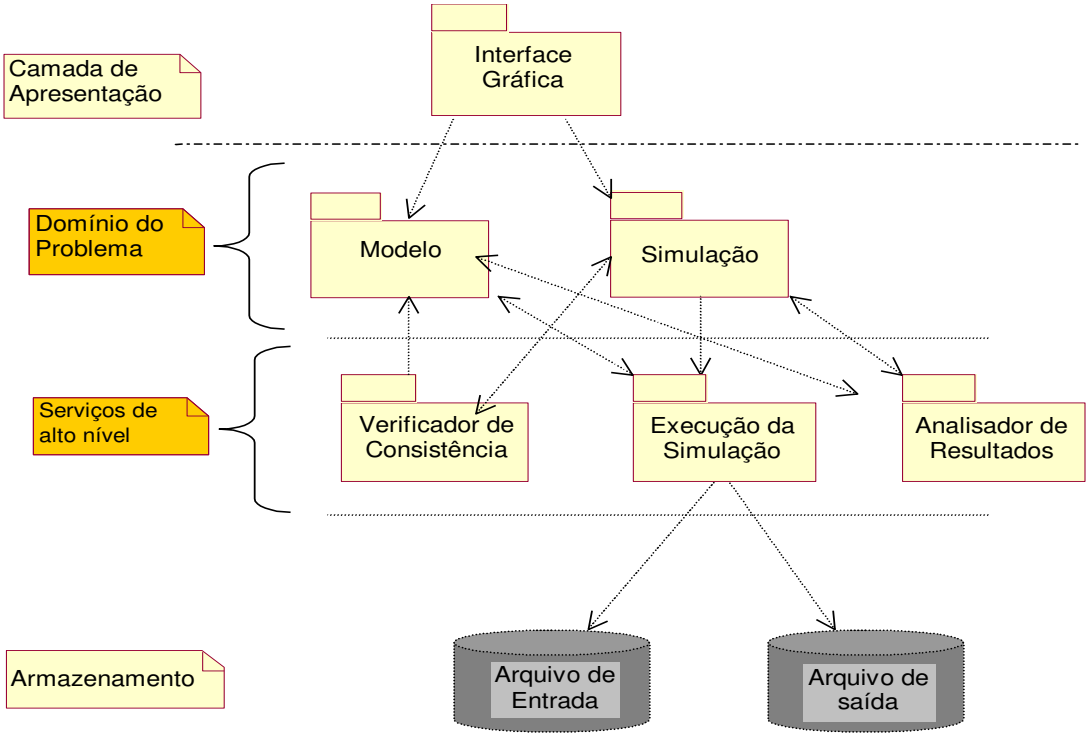

Figura 4.7: Projeto Arquitetural em Camadas.

As camadas de apresentação e de armazenamento não estão contidas dentro do escopo desta Dissertação. Este trabalho está focado na camada da lógica da aplicação que na figura 4.7 está sendo mostrada divida em 2 subcamadas: (i) Domínio do problema: nesta

camada estão inseridos os componentes que são relacionados ao analista de modelagem e dizem respeito à construção do modelo e configuração da simulação. Estes componentes também se relacionam com a apresentação dos resultados e (ii) Serviços de Alto Nível: nesta camada estão inseridos os componentes relativos à execução da simulação, tais como relógio, escalonador de eventos, verificador de consistência, etc.

4.2.3.2 Projeto Detalhado ou Projeto de Baixo Nível

Após a fase de arquitetura do sistema, passa-se à fase de projeto detalhado. Enquanto que a fase de projeto arquitetural consiste numa visão "macro" do sistema, a fase de projeto detalhado visa definir a arquitetura interna da solução lógica, identificando os componentes individuais que compõem o sistema.

Durante essa fase alguns diagramas de colaboração foram elaborados. Os diagramas de colaboração mostram como os componentes devem se comunicar de maneira a atender os requisitos especificados, além de ajudar a atribuir responsabilidades (métodos) a cada componente. Os diagramas de colaboração também têm a capacidade de expressar exceções.

Na expansão e refinamento realizados nos diagramas de colaboração descritos em [Lula, 01] e em [Wagner, 00] foram observados novos componentes bem como novas interações necessitando, portando, de um detalhamento maior desses relacionamentos para que essa fase de desenvolvimento seja perfeitamente compreendida e reaproveitada em trabalhos futuros.

Os diagramas de interação no restante deste capítulo são representados através de diagramas de colaboração em virtude destes possibilitarem maior riqueza de detalhes. Nesses diagramas, os retângulos representam componentes ou classes e as linhas os eventos ou métodos usados na comunicação. Todos os eventos que esses diagramas apresentam foram descritos na seção 4.2.2 (fase de análise).

Para proporcionar uma maior compreensão dos diagramas de colaboração apresentados nesta seção, faz-se necessário uma descrição detalhada da topologia dos modelos de redes *TCP/IP* que serão executados no ambiente. A seção 4.2.3.2.1 apresenta as características da topologia dos modelos considerados no trabalho ao passo que a seção 4.2.3.2.2 descreve todos os diagramas de colaboração criados.

4.2.3.2.1 Aspectos Gerais da Topologia do Modelo da Rede TCP/IP

As topologias das redes que consideramos como estudos de caso contém basicamente *hosts* , roteadores e *enlaces*. Um exemplo dessa topologia é mostrada na figura 4.8.

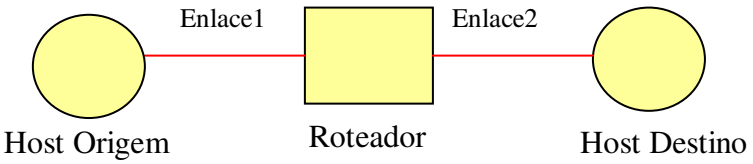

Figura 4.8: Exemplo de uma Topologia com 1 Roteador.

A figura 4.8 poderia corresponder a topologia real de uma rede *TCP/IP*, no entanto, em um processo de desenvolvimento de software, algumas vezes, precisamos fazer simplificações e abstrações no sistema para que a simulação do mesmo seja viável computacionalmente. Dessa forma, foram acrescidos novos componentes para representar determinados aspectos de uma arquitetura *TCP/IP* para que modelos deste tipo de rede possam ser simulados.

A camada de transporte do *host* origem é abstraída através do componente *Fonte* que tem por função gerar mensagens que são enviadas à camada *IP*. No host destino a camada de transporte é representada através do elemento *Sorvedouro* que ao contrário do componente *Fonte,* recebe pacotes da camada *IP* representando, assim, o recebimento dos pacotes. A figura 4.10 mostra esses relacionamentos.

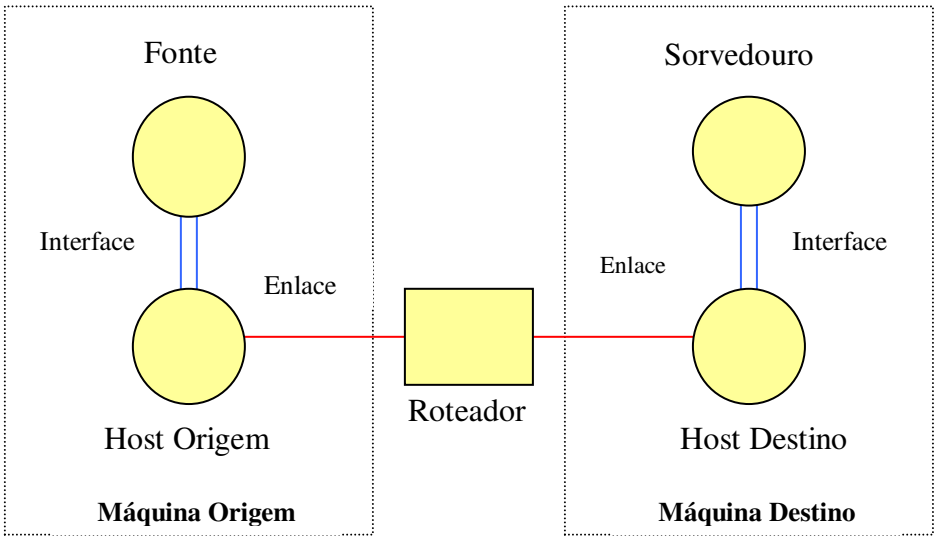

Figura 4.9: Exemplo de um Modelo de Redes que pode ser Simulado.

A seguir, são apresentados diagramas de colaboração para as fases de uma simulação: inicialização, execução e finalização.

4.2.3.2.2 Diagramas de Colaboração para a Fase de Inicialização - Construção do Modelo

Conforme mostra a figura 4.1 as primeiras etapas de um processo de simulação consistem na construção e configuração do modelo a ser simulado. Uma *interface* (preferencialmente gráfica) do ambiente é responsável em instanciamento e configuração dos componentes do modelo. Neste ponto, os componentes do ambiente (lista de eventos, relógio, etc) já terão sido instanciados, configurados e "conectados" através de eventos.

a) **Inserção e Cadastro de Elementos de Modelagem no Ambiente de Simulação**

Na instanciação, uma das tarefas primordiais é a conexão de componentes aos componentes já existentes (componentes do ambiente). Para isso, o ambiente fornece um componente chamado *Simulador* que é responsável em "conectar" os componentes do modelo instanciados pelo usuário aos componentes do ambiente, de modo que estes possam se comunicar.

O diagrama de colaboração da figura 4.10 mostra que para cada componente instanciado pelo usuário para construir o modelo de uma rede *TCP/IP* são gerados dois eventos: (1) o *InsereElementoModelagemEvent* e (2) o *CadastraEvent*. O primeiro faz com que o componente *Modelo* armazene o novo elemento de rede ao passo que o segundo faz a "ligação" deste novo componente aos componentes do ambiente. Este procedimento de cadastro tardio em tempo de execução do ambiente de simulação faz-se necessário porque o ambiente não sabe quais são os componentes do modelo com quem ele irá interagir. A solução para isto foi dotar cada componente que tenha de interagir com os elementos da rede, com métodos que sejam capazes de fazer este cadastro de forma transparente ao usuário do sistema.

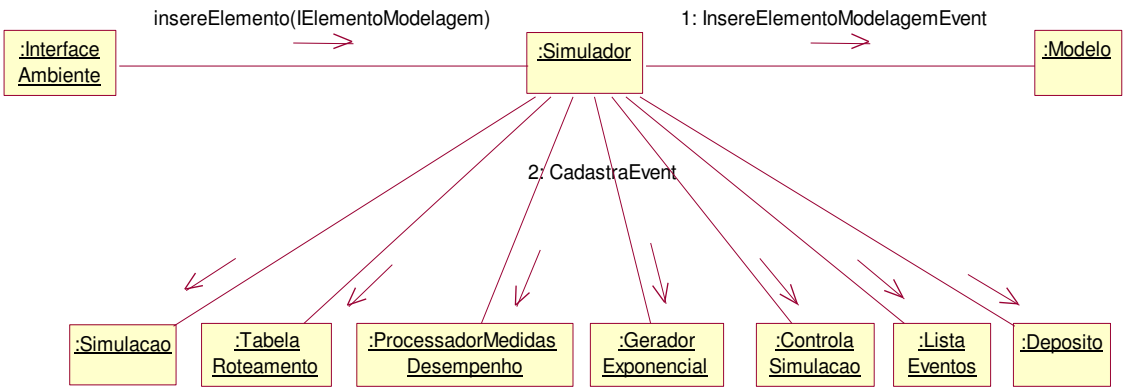

Figura 4.10: Construção do Modelo – Inserção e Cadastro de Componentes.

b) **Remoção e Descadastro de Elementos de Modelagem do Ambiente de Simulação**

De modo inverso à inserção, o ambiente deve ser capaz de fornecer meios de remover componentes que fazem parte do modelo. O diagrama da figura 4.11 mostra essa dinâmica similar ao da figura 4.10, porém realizando tarefa oposta.

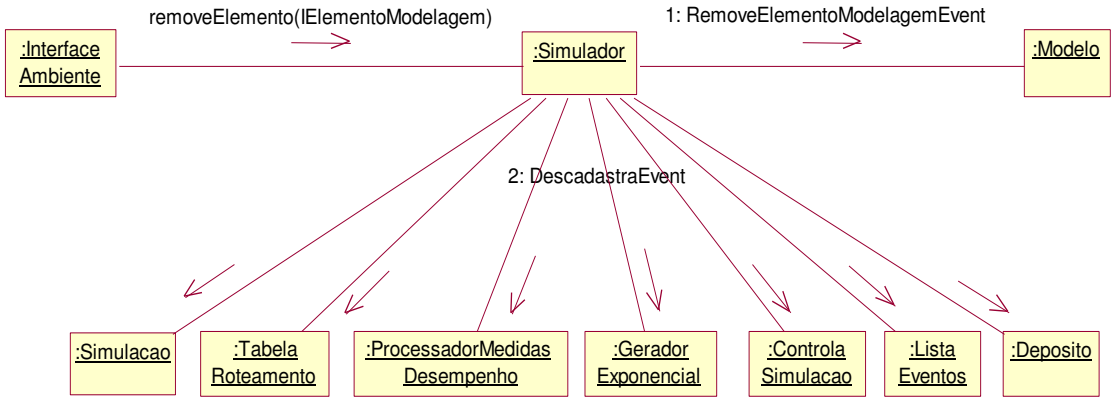

Figura 4.11: Construção do Modelo – Remoção e Descadastro de Componentes.

c) **Inserção de uma Rota no Modelo**

Após a construção do modelo o usuário do ambiente precisa estabelecer rotas por onde os pacotes devem transitar. O componente *TabelaRoteamento* é responsável por armazenar e manipular tais rotas. Toda esta dinâmica é representada na figura 4.12 que representa a inserção de uma nova rota no modelo a ser simulado.

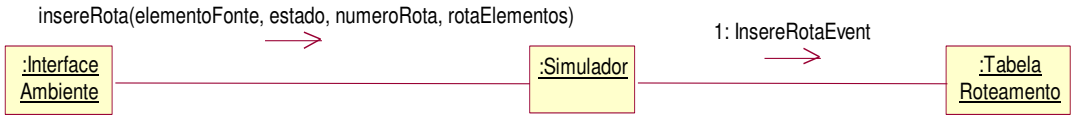

Figura 4.12: Construção do Modelo – Inserção de uma Rota.

d) **Remoção de uma Rota do Modelo**

O ambiente também fornece a possibilidade de remover uma determinada rota da tabela de roteamento. Para isto, é necessário passar para o *Simulador* apenas a identificação da fonte que gera os pacotes da rota e o número da rota. Essa interação é apresentada na figura 4.13.

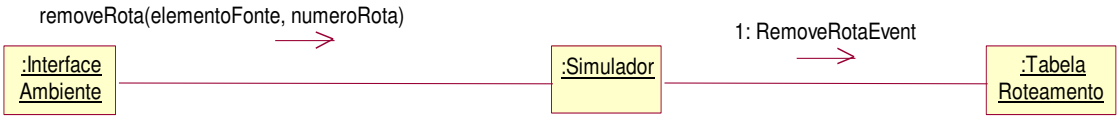

Figura 4.13: Construção do Modelo – Remoção de uma Rota.

e) **Tornar Rota Ativa/Inativa**

Por fim, pode-se tornar uma rota ativa ou não. Quando ativa, pacotes são gerados e transitam pela rota. Esta facilidade permite definir um conjunto de rotas (ativas ou não) para um modelo sem a necessidade de remover as rotas inativas. As figuras 4.14 e 4.15 mostram as colaborações necessárias para tornar uma rota ativa ou inativa.

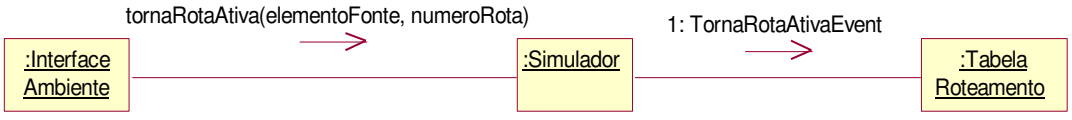

Figura 4.14: Construção do Modelo – Tornar Rota Ativa.

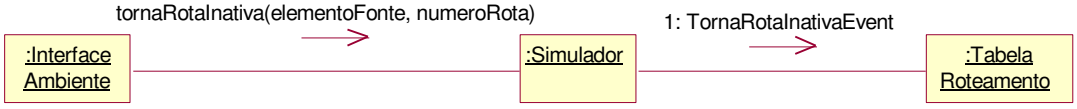

Figura 4.15: Construção do Modelo – Torna Rota Inativa.

4.2.3.2.3 Diagramas de Colaboração para a Fase de Execução

a) **Inicialização dos Componentes (Ambiente e Modelo)**

Uma vez construído e configurado, o modelo já pode ser executado. Mais uma vez, destacamos que esta inicialização não corresponde a criação e configuração do modelo (fase de inicialização de um processo de simulação ilustrada na figura 4.1) e sim a inicialização dos componentes do ambiente para que a execução da simulação possa prosseguir (evento *InicializaEvent* da figura 4.6 ou 4.16).

De acordo com o diagrama de colaboração da figura 4.16, a simulação ocorre quando o usuário usando uma *interface* (preferencialmente gráfica), aqui representada pela classe *InterfaceAmbiente*, invoca o método *executa()* do componente *Simulador*. Este componente é responsável em fornecer uma *interface* orientada a objetos para a comunicação com a classe *InterfaceAmbiente* e uma outra orientada a eventos para os componentes da lógica da aplicação. Dessa forma ao receber a chamada de método *executa(),* o componente *Simulador* gera o evento *InicializaSimulacaoEvent* para que a simulação possa iniciar.

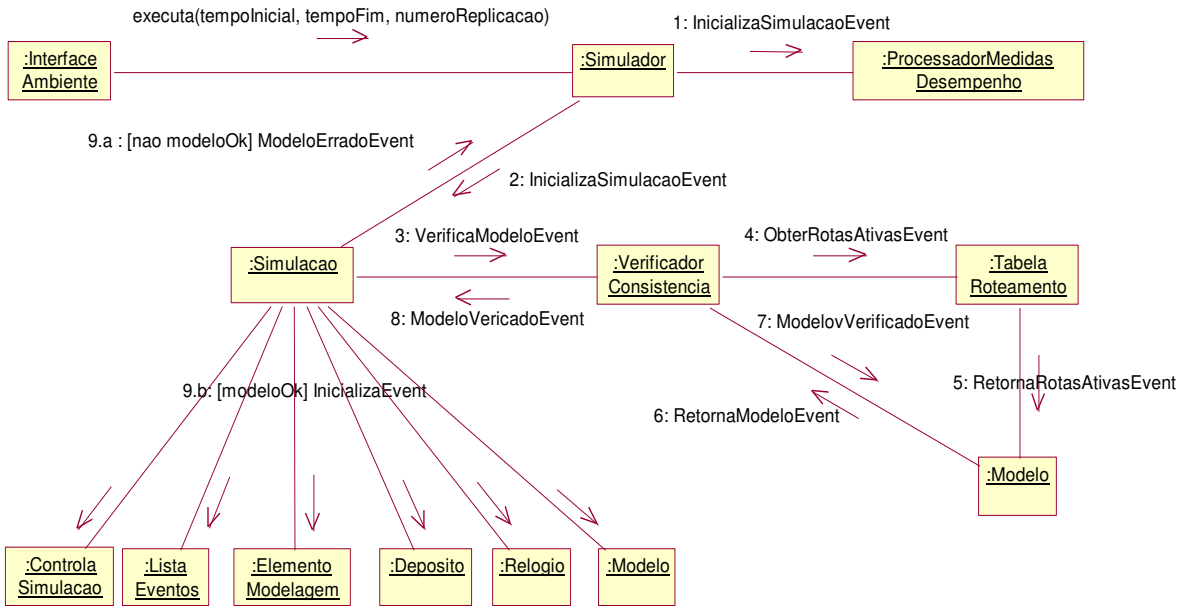

Figura 4.16: Inicialização dos Componentes do Ambiente.

A inicialização dos componentes ocorre depois da verificação do modelo pelo componente *VerificadorConsistencia*. Isto é importante, pois caso seja detectado algum erro no modelo a ser simulado, a inicialização não ocorre, e todo o processo é interrompido à espera da correção por parte do usuário. O diagrama da figura 4.16 mostra o componente *TabelaRoteamento* que é responsável em armazenar todas as rotas por onde os pacotes devem transitar. O verificador de consistência só verifica a consistência das rotas ativas, o que possibilita que o usuário defina várias rotas e possa escolher quais delas deverão ser executadas não necessitando apagar as inativas. No entanto, como mostra a figura 4.16, o componente *TabelaRoteamento* não envia o resultado diretamente para o componente *VerificadorConsistencia* e sim para o componente *Modelo,* que armazena a rota e a repassa, juntamente com uma lista de todos os componentes instanciados, para o componente *VerificadorConsistencia*. Isto é importante, porque a consistência é feita com base nas rotas ativas e nos elementos da rede instanciados.

Caso o modelo esteja correto, o componente *Simulacao* gera o evento *InicializaEvent* para que os componentes sejam inicializados. Um aspecto importante a observar é que a inicialização do componente *ProcessadorMedidasDesempenho,* responsável por coletar as estatísticas, ocorre antes da inicialização do componente *Simulacao*. Tal procedimento faz com que o *ProcessadorMedidasDesempenho* não perca seus dados a cada replicação, caso o usuário deseje que o modelo seja executado mais de uma vez a cada simulação. Dessa forma, para cada simulação, este componente é inicializado apenas uma vez.

O processo de inicialização acionado pelo evento *InicializaEvent* pode ter resultados diferentes dependendo do componente que trata o evento. No caso da lista de eventos (componente *ListaEventos*), há a remoção de todos os eventos em fila e a inserção do primeiro evento, o *FimSimulacaoEvent* com o tempo igual ao tempo final da simulação que foi introduzido pelo usuário através de uma *interface* do ambiente. O componente *Relogio,* por sua vez, apenas inicializa o tempo corrente da simulação com valor zero.

b) **Execução da Simulação**

Imediatamente após a inicialização dos componentes, a execução da simulação prossegue acionando as fontes do modelo. Os diagramas das figuras 4.17 e 4.18 apresentam todas as interações necessárias para a execução de uma simulação. É importante observar que

estes diagramas são apenas uma outra forma de apresentar o diagrama de seqüência mostrado na figura 4.6. No entanto, a sua exibição é relevante porque, através de um diagrama de colaboração, podemos mostrar mais detalhes acerca das interações entre os componentes.

Em virtude do tamanho e da complexidade apresentada nesse diagrama, o mesmo foi dividido em dois sub-diagramas que são mostrados nas figuras 4.17 e 4.18. Alguns elementos estão repetidos a fim de auxiliar a compreensão de cada diagrama.

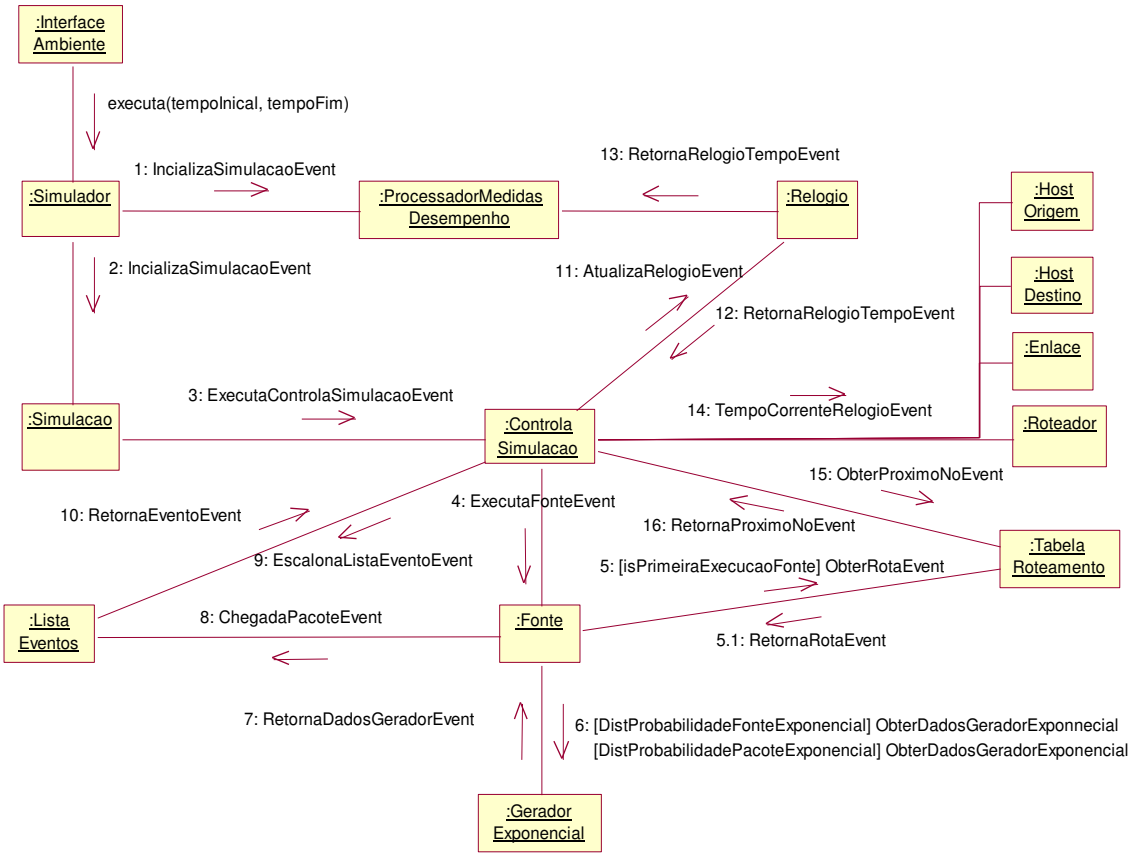

Figura 4.17: Diagrama de Colaboração para a Operação de Sistema Executa( ) – Primeira Parte.

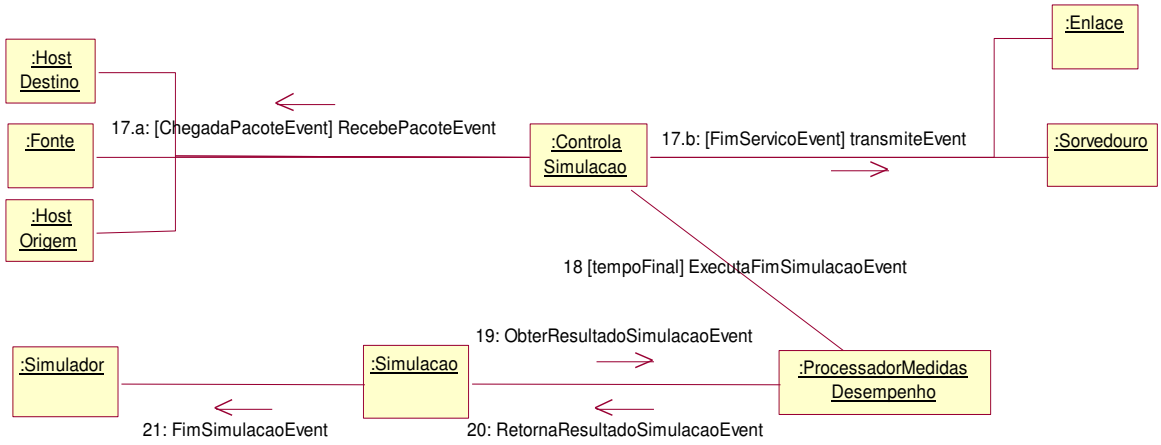

Figura 4.18: Diagrama de Colaboração para a Operação de Sistema Executa( ) – Segunda Parte.

Conforme é mostrado no diagrama da figura 4.17, a fase de execução da simulação começa com a inicialização dos componentes. Em seguida é inicializado a execução propriamente dita através da geração do evento *ExecutaControlaSimulacaoEvent* pelo componente *Simulacao*. Apenas o componente *ControlaSimulacao* trata este tipo de evento. Ao ouvi-lo, ele gera o evento *ExecutaFonteEvent* que é ouvido por todas as fontes do modelo. No entanto, somente as que possuem rotas ativas e estão ativas é que podem gerar pacotes com comprimentos fixos ou conforme amostras obtidas de uma distribuição de probabilidade definida pelo usuário.

Como podemos observar na figura 4.17, no momento em que a fonte é executada pela primeira vez, ela gera um evento que aciona o componente *TabelaRoteamento* para que o mesmo retorne para qual rota a fonte deve gerar os pacotes. De posse dessa informação, ela gera o evento *ChegadaPacoteEvent* que é armazenado na lista de eventos do sistema. Ao ser escalonado pelo componente *ControlaSimulacao,* o pacote encapsulado dentro desse evento é copiado para um outro evento chamado *RecebePacoteEvent* que, por sua vez, aciona o componente que representa o elemento do modelo que deve receber o pacote de acordo com a tabela de roteamento. Nesse caso, cada *ChegadaPacoteEvent* ocasiona a geração de um *RecebePacoteEvent.*

O trânsito dos pacotes pelo ambiente gera um outro tipo de evento: o *FimServicoEvent.* Esse evento aciona o componente *ControlaSimulacao,* que gera o evento *TransmiteEvent*. O último evento a ser escalonado é o *FimSimulacaoEvent* que determina o

fim da simulação. Dessa forma vemos que todo o processo de simulação foi resumido ao armazenado e escalonamento pelo componente *ListaEventos* de apenas três tipos de eventos: *ChegadaPacoteEvent*, *FimServicoEvent* e *FimSimulacaoEvent*. Os eventos *RecebePacoteEvent* e *TransmiteEvent* também são necessários à dinâmica da simulação, no entanto, não são armazenados na lista de eventos. Toda esta dinâmica esta ilustrada nos diagramas das figuras 4.17 e 4.18.

A geração de um pacote pela fonte, e o conseqüente movimento deste pelo modelo, faz gerar eventos que se repetem segundo uma seqüência predeterminada até que o pacote chegue ao elemento *Sorvedouro* do modelo. A figura 4.19 representa esta situação mostrando que um *ChegadaPacoteEvent* implica na geração de um *RecebePacoteEvent* que por sua vez implica na geração de um *FimservicoEvent* que, finalmente faz o sistema gerar um *TransmiteEvent*. Se o elemento que o *TransmiteEvent* acionar for um *Sorvedouro,* as medidas do pacote são coletadas e o mesmo é eliminado do modelo. Caso contrário, este pacote será transmitido por um *Enlace,* ocasionando a repetição da seqüência.

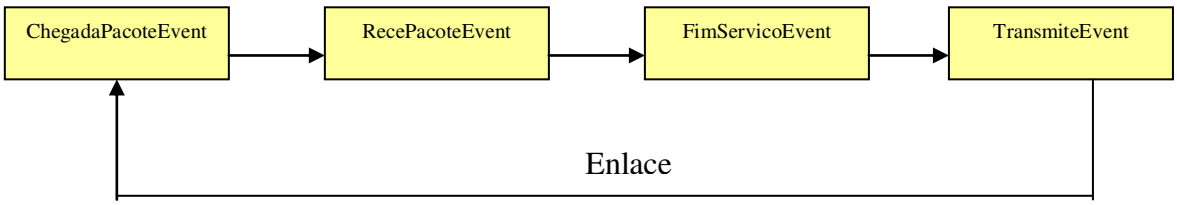

Figura 4.19: Diagrama da Seqüência em que os Eventos Responsáveis pela Dinâmica da Simulação são Gerados no Ambiente.

As ações pertinentes à execução dos eventos *ChegadaPacoteEvent*, *FimServicoEvent* e *FimSimulacaoEvent* são apresentados nas subseções *i*, *ii* e *iii* desta seção, respectivamente.

i) **Processamento do evento ChegadaPacoteEvent pelo componente ControlaSimulacao**

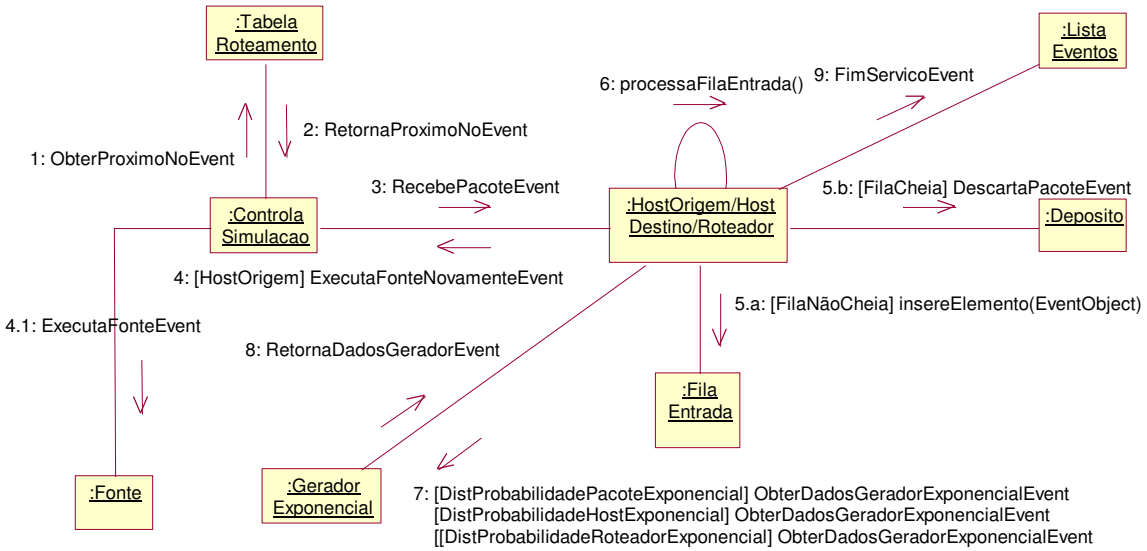

Figura 4.20: Diagrama de Colaboração para o Processamento do Evento ChegadaPacoteEvent pelo Componente ControlaSimulacao.

Quando o evento *ChegadaPacoteEvent* é escalonado pelo componente *ControlaSimulacao,* este componente gera um evento chamado *RecebePacoteEvent* que é ouvido apenas pelos *Hosts* ou *Roteadores*. Observando a figura 4.20 vemos que quando um *RecebePacoteEvent* aciona um componente, a primeira ação do componente é tentar inserir o evento na sua fila de entrada. Se esta estiver cheia o evento é descartado, caso contrário ele é armazenado. Tal procedimento é feito através da verificação do estado do componente que indica se ele está livre (podendo processar pacotes e portanto com fila de entrada vazia) ocupado (processando um pacote e fila de entrada ainda não cheia), congestionado (processando um pacote com sua fila de entrada cheia ocasionando descarte de pacotes) e por fim inativo (quando o elemento encontra-se desativado no ambiente ocasionando descarte de pacotes). A figura 4.21 apresenta o diagrama de estado para os componentes *Host*/*Roteador.*

O diagrama de colaboração da figura 4.20 para a chegada de pacotes no componente *Roteador* é similar ao do *Host* (origem e destino) diferenciando basicamente no fato do *Roteador* não pode gerar o evento *ExecutaFonteNovamenteEvent*. O componente *Roteador,* assim como o componente *Host* (origem), apresenta procedimentos iguais para tratamento dos pacotes nas suas filas, portanto, o componente *Deposito* também interage com

o componente *Roteador*. Por fim, o tratamento do evento *RecebePacoteEvent* pelos *Hosts* ou *Roteadores* resulta na geração do evento *FimServicoEvent*.

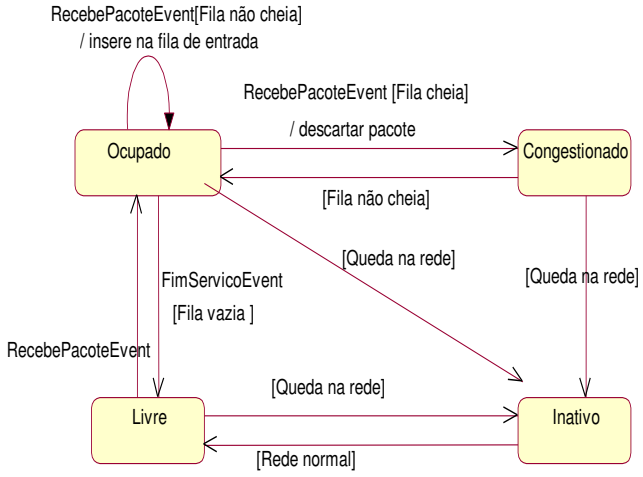

Figura 4.21. Diagrama de Estado dos Componentes Host e Roteador.

ii) **Processamento do Evento FimServicoEvent pelo Componente ControlaSimulacao**

Após o processamento de um pacote em um componente *Host* ou *Roteador,* estes geram um evento chamado *FimServicoEvent* que é inserido na lista de eventos. O escalonamento, e conseqüentemente processamento desse evento, implica na geração de um evento chamado *TransmiteEvent,* que se destina aos componentes *Enlace* e *Sorvedouro*.

A figura 4.22 mostra o diagrama de colaboração referente ao processamento do evento *FimServicoEvent* pelo componente *ControlaSimulacao,* quando o componente que é acionado pelo evento é o *Enlace.*

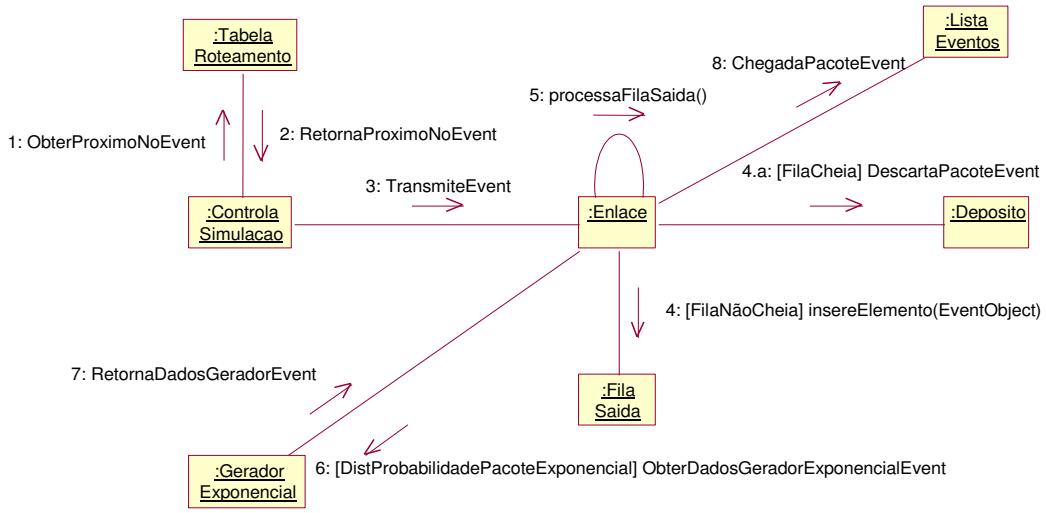

Figura 4.22: Diagrama de Colaboração para o Processamento do Evento FimServicoEvent pelo Componente ControlaSimulacao – Componente Enlace é Acionado pelo Evento TransmiteEvent.

Como se pode observar no diagrama da figura 4.22, as colaborações são praticamente as mesmas apresentadas na figura 4.20, diferenciando na ausência do componente *Fonte* e na substituição da fila de entrada do componente *Host* (origem ou destino) ou *Roteador* pela fila de saída do componente *Enlace*. O componente *Enlace* apresenta uma única fila, chamada FilaSaida, que armazena os pacotes provenientes dos elementos que estão a ele conectados. Dessa forma um *Host* (origem ou destino) ou um *Roteador* apresentam apenas uma única fila de entrada e uma ou várias filas de saídas, dependendo do número de enlaces a eles conectados.

Quando o evento *TransmiteEvent* é ouvido pelo componente *Enlace,* após a transmissão do pacote, é gerado um evento chamado *ChegadaPacoteEvent* para o próximo elemento de modelagem na rota na qual o componente *Enlace* está conectado.

Observando a figura 4.22, vemos que quando um evento *TransmiteEvent* é ouvido por um componente *Enlace*, a primeira ação é tentar inseri-lo na sua fila que representa a fila de saída do componente diretamente a ele conectado. Se esta estiver cheia o evento é descartado, caso contrário ele é armazenado. Tal procedimento é feito através da verificação do estado do componente que indica se ele está livre (podendo processar pacotes e portanto com fila de saída vazia), ocupado (processando um pacote e fila de saída ainda não cheia),

congestionado (processando um pacote com sua fila de saída cheia, ocasionando descarte de pacotes) e, por fim, inativo (quando o elemento encontra-se desativado no ambiente, ocasionando descarte de pacotes). A figura 4.23 apresenta o diagrama de estados para o componente *Enlace*.

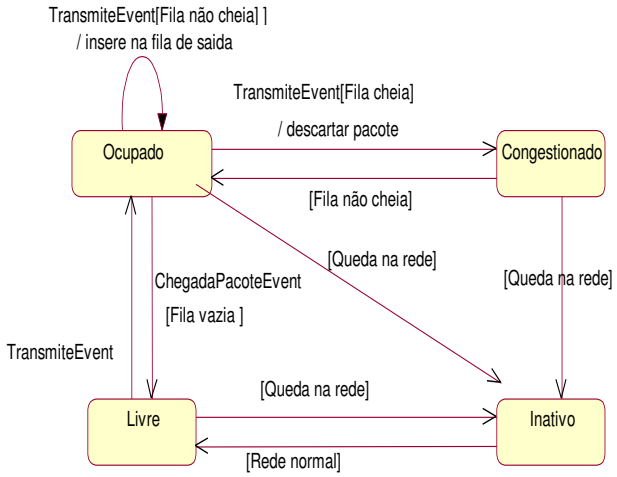

Figura 4.23. Diagrama de Estado do Componente Enlace.

No caso do componente *Sorvedouro* o diagrama de colaboração para o evento *FimServicoEvent* é simples como pode ser observado na figura 4.24. Isto se deve ao fato de que a única função desse componente ao "ouvir" o evento *TransmiteEvent* é a de coletar as estatísticas do pacote e descartá-lo não gerando outros eventos.

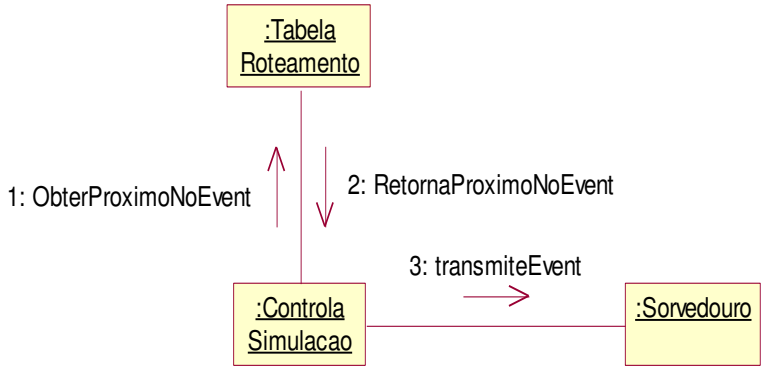

Figura 4.24: Diagrama de Colaboração para o Processamento do Evento FimServicoEvent pelo Componente ControlaSimulacao – Componente Sorvedouro é Acionado pelo Evento TransmiteEvent.

iii) Processamento do Evento FimSimulacaoEvent pelo Componente ControlaSimulacao.

Quando o tempo designado pelo usuário para o fim de Simulação é alcançado, o evento *FimSimulacaoEvent* é escalonado, caso não haja nenhum outro evento com o mesmo tempo de geração. O processamento desse evento pelo componente *ControlaSimulacao* apenas encerra o laço que controla a execução da simulação que corresponde ao do algoritmo apresentado na figura 4.2. Assim, o componente *ControlaSimulacao* gera o evento *ExecutaFimSimulacaoEvent*, que aciona o componente *ProcessadorMedidasDesempenho* para que este obtenha os valores coletados durante a simulação pelos elementos de modelagem. Se o componente *Simulacao* determinar que é preciso uma nova replicação, toda a simulação é reinicializada e, ao término, os novos resultados são armazenados em um *String* de *Java,* para posterior comparação. Se não houver mais replicações, a simulação termina e os resultados das medidas de desempenho armazenados no *String* são apresentados. A figura 4.25 mostra as interações do componente *ProcessadorMedidasDesempenho* quando este é acionado pelo evento *ExecutaFimSimulacaoEvent*.

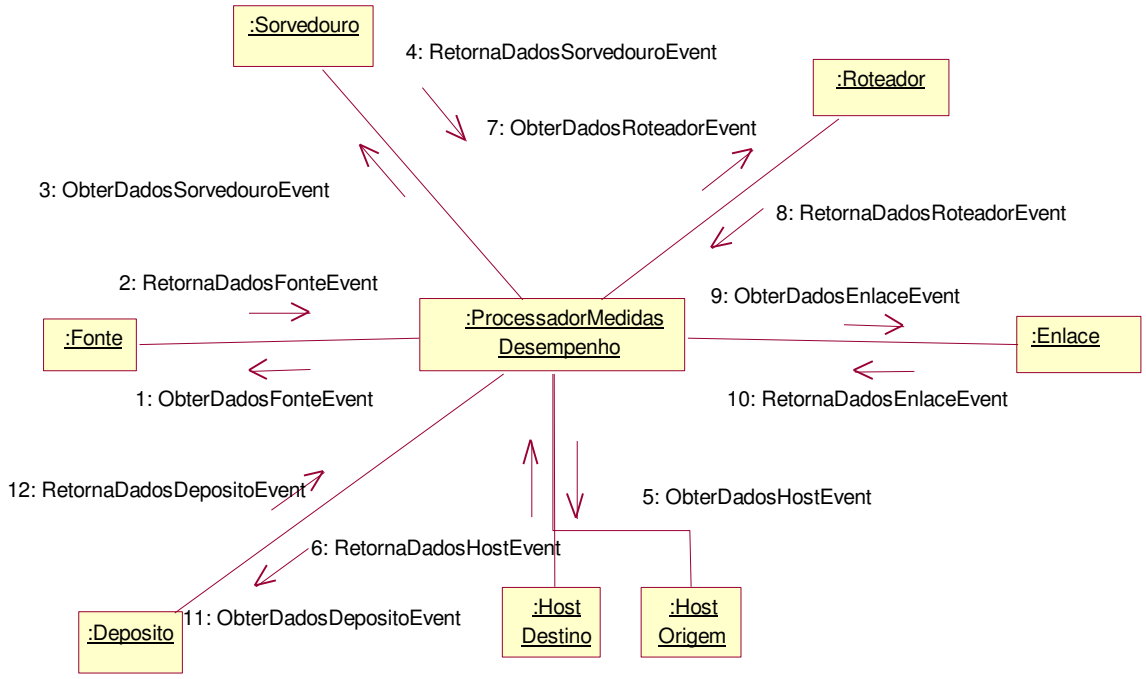

Figura 4.25. Diagrama de Colaboração Resultante do Acionamento do Componente ProcessadorMedidasDesempenho pelo Evento ExecutaFimSimulacaoEvent.

4.2.3.2.4 Descrição dos Componentes Especificados

Finalmente, dos diagramas de interação e do modelo conceitual foram definidos os diagramas de classes que representam os componentes. Cada componente é visto como uma classe, dentro dos conceitos da linguagem *UML* [Lula, 01].

É importante observar que os diagramas de classes apresentados nesta seção não mostram todos os relacionamentos que os componentes ou classes exigem. Isto se deve ao fato de evitar uma complexidade exagerada na apresentação desses diagramas. Portanto, os diagramas aqui apresentados restringem-se apenas a mostrar os relacionamentos mais importantes dos componentes abordados bem como as *interfaces* e classes abstratas que implementam. Para a visualização de todos os relacionamentos, excluindo as *interfaces*, os diagramas de conceitos das figuras 4.4 e 4.5 podem ser utilizados. Nesse caso, lembramos que tais artefatos foram resultados de um refinamento ocorrido na fase de implementação contendo, portanto, todas as classes e componentes do ambiente e não apenas "candidatos" as classes ou aos componentes como o que ocorre normalmente com o diagrama de conceitos gerado na fase de análise.

Uma outra observação importante é que não serão mostrados todos os diagramas de classes produzidos em virtude do seu elevado número, apenas serão mostrados os mais relevantes. No entanto, todos os diagramas e o código *Java* produzido durante a implementação dos mesmos poderão ser vistos no relatório técnico produzido a partir deste trabalho [Rocha, 02].

Por fim, em virtude do grande número de propriedades e de métodos usados para o cadastro de *listeners* em alguns componentes iremos adotar o seguinte procedimento para a explicação dos mesmos:

- ✓ *GetX()* e *setX()* são métodos utilizados para definir a propriedade "x" do componente. Nesse caso, as suas descrições serão feitas de uma única vez, informando apenas que são os métodos que definem a propriedade "x".
- ✓ *Add<NomeDoEvento>Listener* e *remove<NomeDoEvento>Listener* são métodos utilizados para o cadastro de classes que desejam receber o evento *NomeDoEventoEvent*. Portanto, as suas descrições informam apenas que são os métodos que fornecem o cadastro/descadastro de *listeners* para o evento *NomeDoEventoEvent*.

- ✓ Todos os componentes implementam a *interface Serializable* que define os métodos *writeObject()* e *readObject()* para permitir o processo de serialização. A implementação dessa *interface* e os seus métodos não são mostrados nos diagramas deste capítulo
- ✓ Não são mostradas as interações entre os componentes com as classes que representam os seus eventos, *ExecutaFonteEvent* por exemplo. Tais eventos são apenas descritos em detalhes na seção 4.2.2 (fase de análise) como forma de diminuir a complexidade dos diagramas de classes apresentados neste capítulo.

O componente responsável pelo avanço do tempo simulado é o *Relogio* (figura 4.26) cujos atributos e métodos são descritos abaixo:

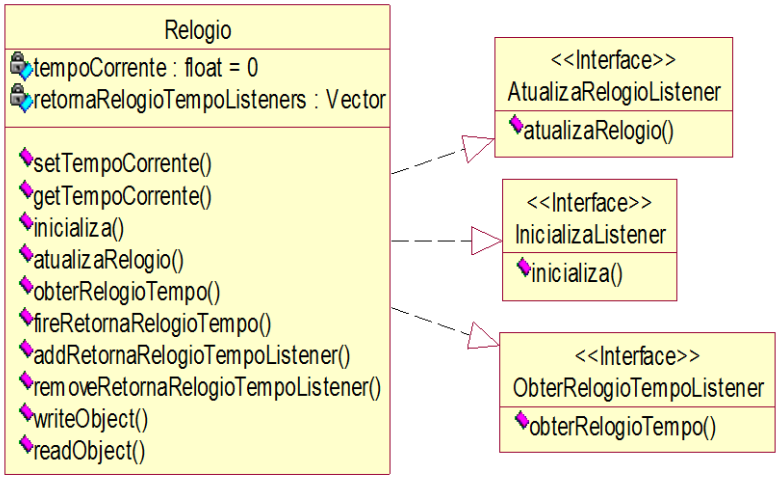

Figura 4.26. Componente Relogio.

Atributos:

- ✓ *tempoCorrente* – Indica o tempo atual em que a simulação se encontra. Por *default* seu valor inicial é zero.
- ✓ *retornaRelogioTempoListeners* – *Vector* em *Java* usado para armazenar os *listeners* do evento *RetornaRelogioTempoEvent.*

Métodos:

- ✓ *setTempoCorrente()* e *getTempoCorrente()* – Definem a propriedade *tempoCorrente*.

- ✓ *inicializa()* –Método da *interface InicializaListener* usado para inicializar o componente *Relogio* com tempo passado pelo evento *InicializaEvent*.
- ✓ *atualizaRelogio()* – Método da *interface AtualizaRelogioListener* usado para atualizar o componente *Relogio* com o tempo passado pelo evento *AtualizaRelogioEvent.*
- ✓ *obterRelogioTempo()* - Método da *interface ObterRelogioListener* usado para obter o tempo corrente do componente *Relogio* que será retornado através do evento *RetornaRelogioTempoEvent*.
- ✓ *fireRetornaRelogioTempo()* – Método usado para disparar o evento *RetornaRelogioTempoEvent*.
- ✓ *addRetornaRelogioTempoListener()* e *removeRetornaRelogioTempoListener()* - Métodos que fornecem o cadastro/descadastro de *listeners* para o evento *RetornaRelogioTempoEvent*.

Durante a fase de implementação foi detectado que havia a necessidade de um componente que manipulasse as rotas criadas pelo sistema. Dessa forma foi criado o componente *TabelaRoteamento* mostrado na figura 4.27.

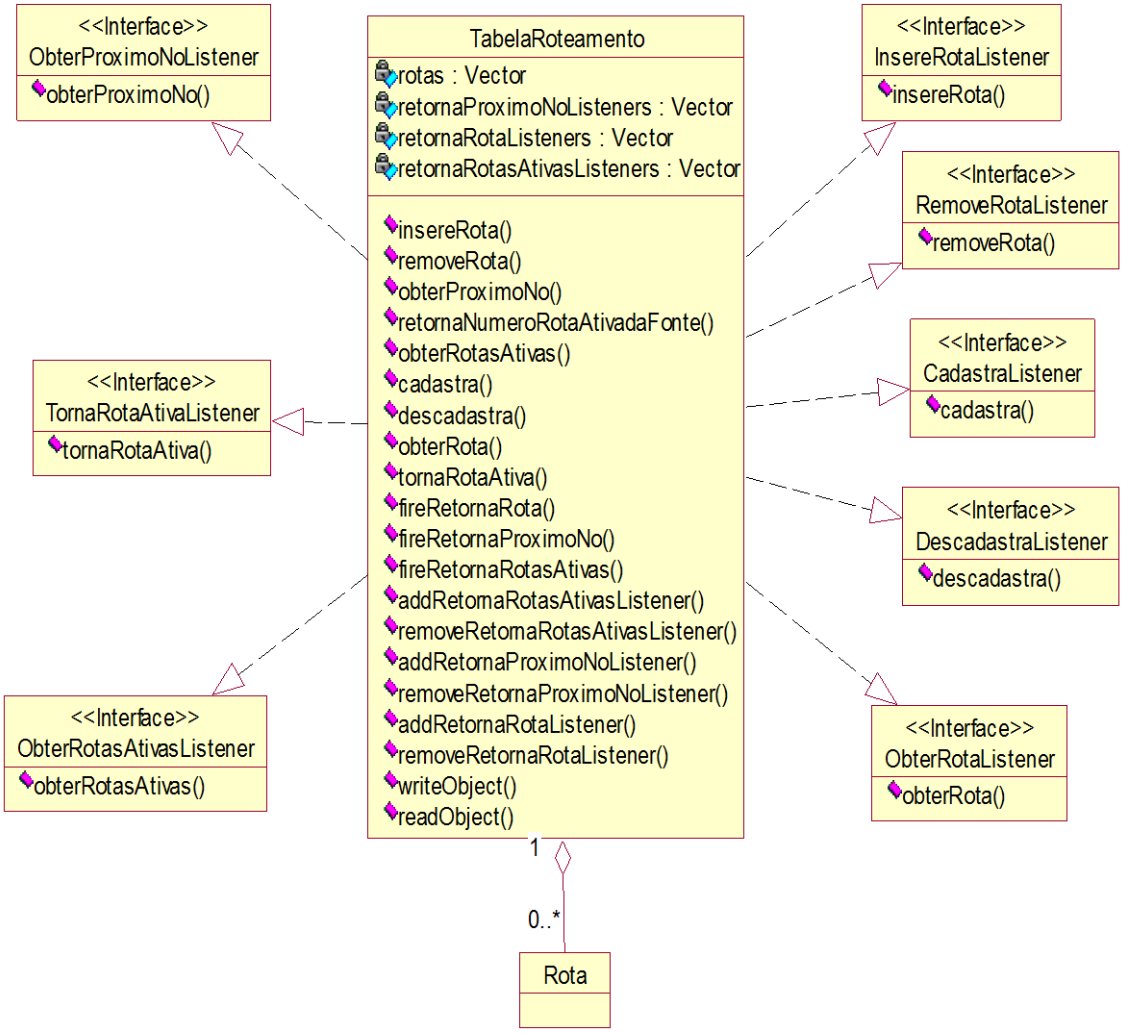

Figura 4.27: Componente TabelaRoteamento.

Atributos:

- ✓ *rotas* – *Vector* em *Java* usado para armazenar todas as rotas ativas ou não.
- ✓ *retornaProximoNoListeners* – *Vector* em *Java* usado para armazenar os *listeners* do evento *RetornaProximoNoEvent.*
- ✓ *retornaRotaListeners* – *Vector* em *Java* usado para armazenar os *listeners* do evento *RetornaRotaEvent.*
- ✓ *retornaRotasAtivasListeners* – *Vector* em *Java* usado para armazenar os *listeners* do evento *RetornaRotasAtivasEvent.*

Métodos:

- ✓ *insereRota()* –. Método da *interface InsereRotaListener* usado para inserir uma nova rota na tabela de roteamento.
- ✓ *removeRota()* –. Método da *interface RemoveRotaListener* usado para remover uma rota da tabela de roteamento.
- ✓ *obterProximoNo()* – Método da *interface ObterProximoNoListener* usado para obter o identificador do próximo elemento de rede para onde o pacote deve ser descartado.
- ✓ *retornaNumeroRotaAtivadaFonte()* – Método usado para retornar o número da rota para qual a fonte deve gerar pacotes.
- ✓ *obterRotasAtivas()* – Método da *interface ObterRotasAtivasListener* usado para retornar todas as rotas que se encontram ativas no modelo.
- ✓ *cadastra()* e *descadastra()* – Métodos da *interface CadastraListener* e *DescadastraListener,* respectivamente, usados para realizar a conexão, método *cadastra()*, ou desconexão, método descadastra(), através de eventos, deste componente com o elemento de modelagem passado como parâmetro. Todos os elementos da rede são cadastrados/descadastrados neste componente.
- ✓ *obterRota()* – Método da *interface ObterRotaListener* usado para retornar o número da rota da fonte. Este método realiza sua função executando o método *retornaNumeroRotaAtivadaFonte()* visto acima.
- ✓ *tornaRotaAtiva()* – Método da *interface TornaRotaAtivaListener* usado para fazer com que uma rota inativa torne-se ativa.
- ✓ *fireRetornaRota()* – Método usado para disparar o evento *RetornaRotaEvent*.
- ✓ *fireRetornaProximoNo()* – Método usado para disparar o evento *RetornaProximoNoEvent*.
- ✓ *fireRetornaRotasAtivas()* – Método usado para disparar o evento *RetornaRotasAtivasEvent*.
- ✓ *addRetornaRotasAtivasListener()* e *removeRetornaRotasAtivasListener()* – Métodos que fornecem o cadastro/descadastro de *listeners* para o evento *RetornaRotasAtivasEvent*.

- ✓ *addRetornaProximoNoListener()* e *removeRetornaProximoNoListener()* – Métodos que fornecem o cadastro/descadastro de *listeners* para o evento *RetornaProximoNoEvent*.
- ✓ *addRetornaRotaListener()* e *removeRetornaRotaListener()* – Métodos que fornecem o cadastro/descadastro de *listeners* para o evento *RetornaRotaEvent*.

Os eventos que controlam o fluxo da simulação são armazenados em uma lista chamada lista de eventos (componente *ListaEventos*). A figura 4.28 mostra o componente *ListaEventos*.

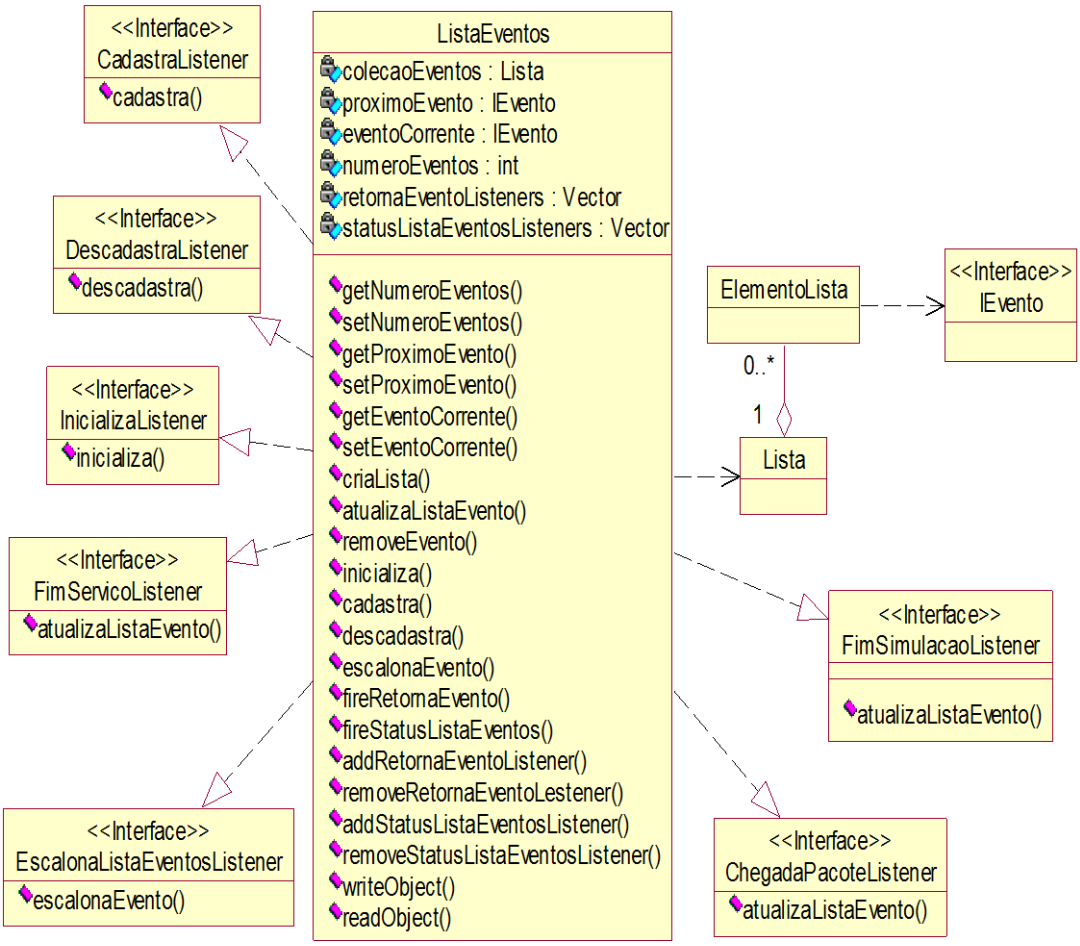

Figura 4.28: Componente ListaEventos.

Atributos:

- ✓ *colecaoEventos* – Atributo do tipo *Lista* que representa a classe usada para armazenar os eventos.
- ✓ *proximoEvento* – Atributo do tipo *IEvento* que representa uma referência ao próximo evento que será escalonado após a remoção do evento corrente.
- ✓ *eventoCorrente* – Atributo do tipo *IEvento* que representa uma referência ao evento corrente que será escalonado.
- ✓ *numeroEventos* – Atributo do tipo *int* que armazena o número de eventos que a lista contém a cada inserção ou remoção.
- ✓ *retornaEventoListeners* – *Vector* em *Java* usado para armazenar os *listeners* do evento *RetornaEventoEvent.*
- ✓ *statusListaEventosListeners* – *Vector* em *Java* usado para armazenar os *listeners* do evento *StatusListaEventosEvent.*

Métodos:

- ✓ *setNumeroEventos()* e *getNumeroEventos()* – Definem a propriedade *numeroEventos*.
- ✓ *setProximoEvento()* e *getProximoEvento()* – Definem a propriedade *proximoEvento*.
- ✓ *setEventoCorrente()* e *getEventoCorrente()* – Definem a propriedade *eventoCorrente*.
- ✓ *criaLista()* – Método usado para criar um objeto do tipo *Lista.*
- ✓ *atualizaListaEvento()* – Método das *interfaces ChegadaPacoteEvent, FimServicoEvent* e *FimSimulacaoEvent* usado para inserir um evento na lista de eventos.
- ✓ *removeEvento()* – Método responsável por remover o primeiro evento da lista de eventos. Ele é executado quando o componente trata o evento *EscalonaListaEventosEvent*.
- ✓ *inicializa()* – Método da *interface InicializaListener* que remove todos os eventos da lista e a inicializa com o evento *FimSimulacaoEvent* com tempo igual ao fim da simulação.

- ✓ *cadastra()* e *descadastra()* – Métodos da *interface CadastraListener* e *DescadastraListener,* respectivamente, usados para realizar a conexão, método *cadastra()*, ou desconexão, método *descadastra()*, através de eventos, deste componente com o elemento de modelagem passado como parâmetro. Todos os elementos da rede exceto o *Sorvedouro* são cadastrados/descadastrados neste componente.
- ✓ *EscalonaEvento()* – Método da *interface EscalonaListaEventosListener* usado para escalonar um evento do componente *ListaEventos.*
- ✓ *fireRetornaEvento()* – Método usado para disparar o evento *RetornaEventoEvent*.
- ✓ *fireStatusListaEventos()* – Método usado para disparar o evento *StatusListaEventosEvent*.
- ✓ *addRetornaEventoListener()* e *removeRetornaEventoListener()* – Métodos que fornecem o cadastro/descadastro de *listeners* para o evento *RetornaEventoEvent*.
- ✓ *addStatusListaEventosListener()* e *removeStatusListaEventosListener()* – Métodos que fornecem o cadastro/descadastro de *listeners* para o evento *StatusListaEventosEvent*.

O diagrama da figura 4.28 mostra que o componente *ListaEventos* instancia a classe *Lista* que por sua vez pode conter vários objetos da classe *ElementoLista.* Cada objeto *ElementoLista* contém uma referência a um evento do tipo *IEvento*.

A figura 4.29 mostra o relacionamento existente entre as classes de eventos do ambiente de simulação. Os eventos exibidos são eventos responsáveis pela dinâmica do processo de simulação não devendo ser confundidos com aqueles outros necessários para fornecer a comunicação entre os componentes.

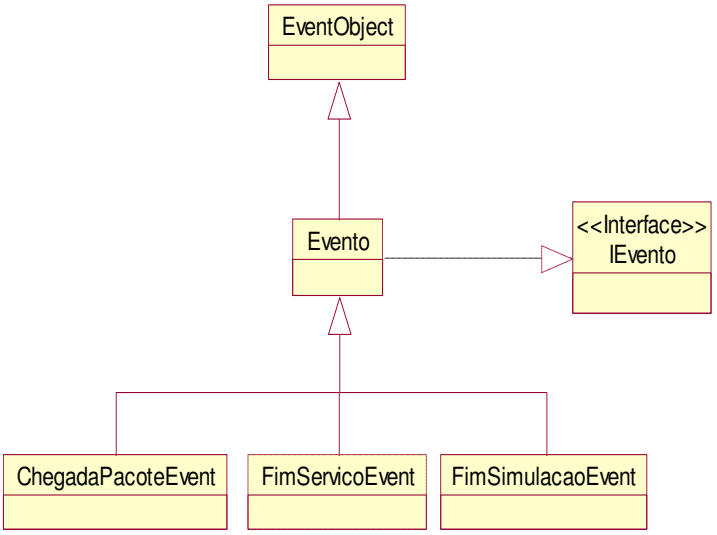

Figura 4.29: Relacionamento dos Eventos do Ambiente.

Pode-se observar na figura 4.29 que os eventos *ChegadaPacoteEvent, FimServicoEvent* e *FimSimulacaoEvent* implementam, através da classe abstrata *Evento,* a *interface Ievento,* e estende a classe *EventObject*. Esta última faz parte do modelo de comunicação do *JavaBeans* e é necessária uma vez que cada evento disparado por um componente precisa ser um *EventObject*. A figura 4.30 apresenta a classe Evento.

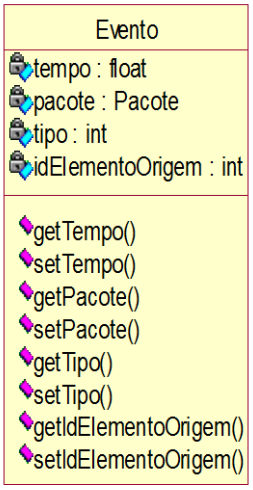

Figura 4.30: Classe Evento.

Atributos

- ✓ *tempo* – Atributo do tipo *float* que representa o tempo de criação do evento.
- ✓ *pacote* –Atributo do tipo *Pacote* que representa uma referência ao pacote de dados armazenado.
- ✓ *tipo* – Atributo do tipo *int* que representa o tipo do evento.
- ✓ *idElementoOrigem* – Atributo do tipo *int* que representa o identificador do elemento de modelagem que criou o evento.

Métodos:

- ✓ *getTempo()* – Método que retorna o tempo de criação do evento.
- ✓ *setPacote()* e *getPacote()* – Definem a propriedade *pacote*.
- ✓ *getTipo() e setTipo()* – Definem a propriedade *tipo*.
- ✓ *getIdElementoOrigem()* e *setElementoOrigem()* – Definem a propriedade *idElementoOrigem*.

A figura 4.31 mostra a classe do evento *ChegadaPacoteEvent*.

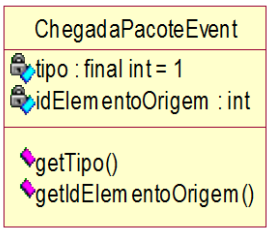

Figura 4.31: Classe ChegadaPacoteEvent.

Atributos

- ✓ *tipo* – Constante do tipo *int* que representa o tipo do evento.
- ✓ *idElementoOrigem* – Atributo do tipo *int* que representa o identificador do elemento de modelagem que criou o evento.

Métodos:

- ✓ *getTipo()* – Método que retorna o tipo do evento.

- ✓ *getIdElementoOrigem* – Método que retorna o identificador do elemento que criou o evento.

Conforme já mencionado os componentes do modelo a serem simulados foram especificados no trabalho de [Wagner, 01] seguindo a mesma metodologia de desenvolvimento adotada no trabalho de [Lula, 01], ou seja, processo de desenvolvimento baseado em Larman e uso da linguagem UML. Na fase de implementação, no entanto, foi verificado que tais componentes apresentavam-se demasiadamente complexos, englobando características não necessárias a simulação, e não forneciam quase nenhuma interação com o ambiente. A figura 4.32 mostra os relacionamentos dos componentes do modelo, obtidos a partir do refinamento das especificações apresentadas nos trabalhos referenciados.

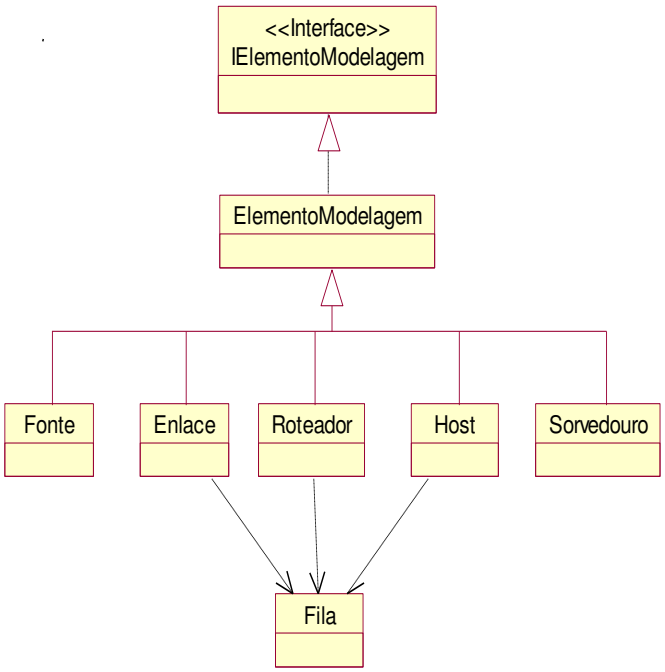

Figura 4.32: Relacionamento dos Componentes do Modelo.

A figura 4.33 apresenta a estrutura do componente *Enlace*.

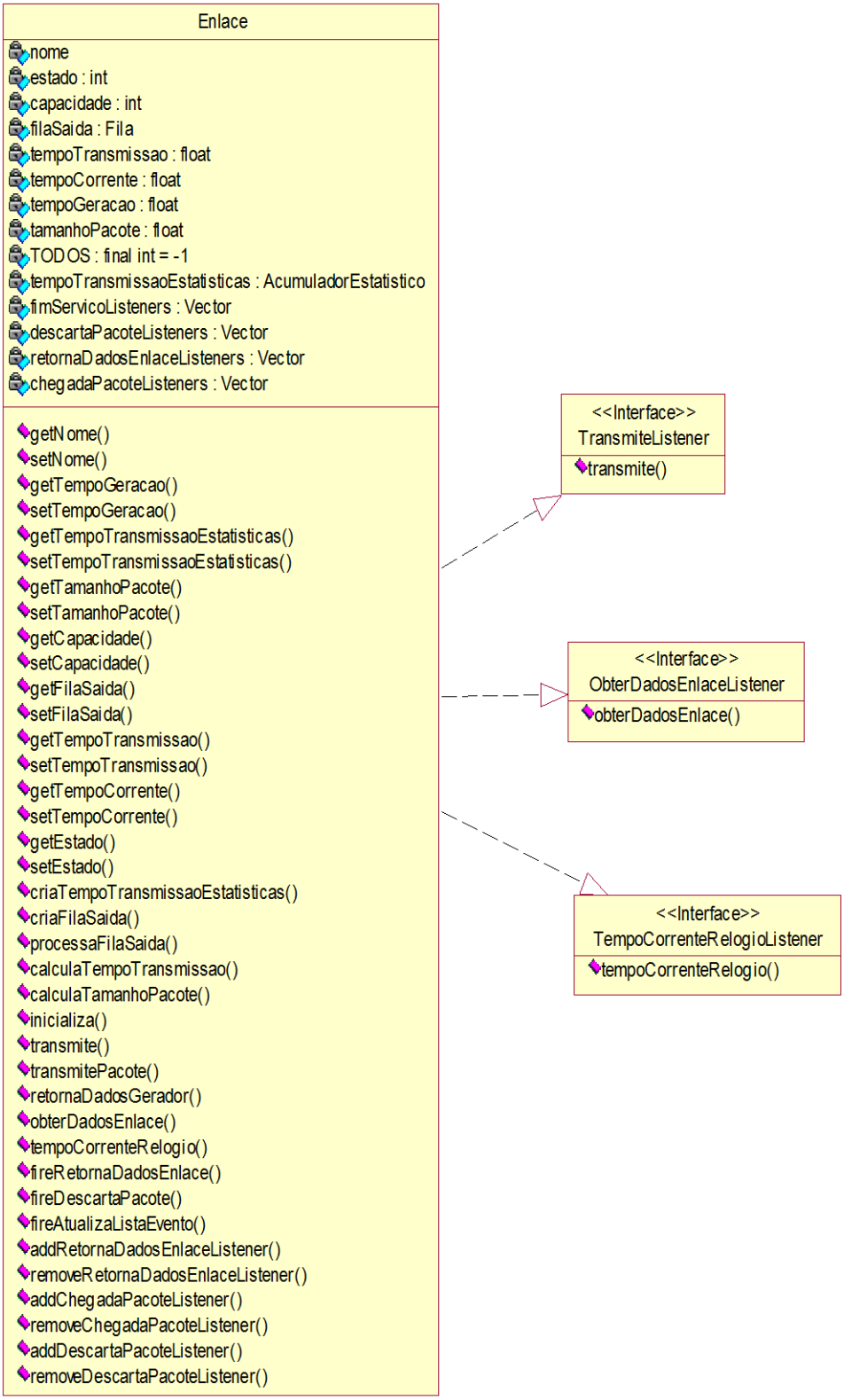

Figura 4.33: Componente Enlace.

Atributos:

- ✓ *nome* – *String* em *Java* utilizado para armazenar o nome do componente.
- ✓ *estado* – Atributo do tipo *int* usado para guardar o número que corresponde ao estado do componente.
- ✓ *capacidade* – Atributo do tipo *int* usado para determinar a capacidade do enlace.
- ✓ *filaSaida* – Atributo do tipo *Fila* que representa a fila de saída do componente *Enlace.* Em um sistema real ela é a fila de saída do *Roteador* ou a fila de saída do *Host.*
- ✓ *tempoTransmissao* – Atributo do tipo *float* usado para armazenar o tempo que o *Enlace* leva para *transmitir o pacote*.
- ✓ *tempoCorrente* – Atributo do tipo *float* usado para armazenar o tempo corrente da simulação.
- ✓ *tempoGeracao* – Atributo do tipo *float* usado para armazenar o tempo em que o pacote foi transmitido. Ele corresponde a soma dos atributos *tempoCorrente* mais *tempoTransmissao*.
- ✓ *tamanhoPacote* – Atributo do tipo *float* usado para armazenar o tamanho do pacote.
- ✓ *TODOS* – Constante do tipo *int* usada para determinar se todos os componentes do tipo *Enlace* devem retornar os seus valores.
- ✓ *tempoTransmissaoEstatisticas* – Atributo do tipo *AcumuladorEstatistico* usado para coletar as estatísticas relacionadas à transmissão dos pacotes.
- ✓ *fimServicoListeners* – *Vector* em *Java* usado para armazenar os *listeners* do evento *FimServicoEvent.*
- ✓ *descartaPacoteListeners* – *Vector* em *Java* usado para armazenar os *listeners* do evento *DescartaPacoteEvent.*
- ✓ *retornaDadosEnlaceListeners* – *Vector* em *Java* usado para armazenar os *listeners* do *RetornaDadosEnlaceEvent.*
- ✓ *chegadaPacoteListeners* – *Vector* em *Java* usado para armazenar os *listeners* do *ChegadaPacoteEvent.*

Métodos:

- ✓ *setNome()* e *getNome()* – Definem a propriedade *nome*.
- ✓ *setTempoFim()* e *getTempoFim()* – Definem a propriedade *tempoFim.*
- ✓ *setTempoGeracao()* e *getTempoGeracao()* – Definem a propriedade *tempoGeracao*.
- ✓ *setTempoTransmissaoEstatisticas()* e *getTempoTransmissaoEstatisticas()* – Definem a propriedade *tempoTransmissaoEstatisticas*.
- ✓ *setTamanhoPacote()* e *getTamanhoPacote()* – Definem a propriedade *tamanhoPacote*.
- ✓ *setCapacidade()* e *getCapacidade()* – Definem a propriedade *capacidade*.
- ✓ *setFilaSaida()* e *getFilaSaida()* – Definem a propriedade *filaSaida*.
- ✓ *setTempoTransmissao()* e *getTempoTransmissao()* – Definem a propriedade *tempoTransmissao*.
- ✓ *setTempoCorrente()* e *getTempoCorrente()* – Definem a propriedade *tempoCorrente*.
- ✓ *setEstado()* e *getEstado()* – Definem a propriedade *estado*.
- ✓ *criaTempoTransmissaoEstatisticas()* – Método utilizado para criar uma instância do acumulador *tempoTransmissaoEstatisticas*.
- ✓ *criaFilaSaida()* – Método utilizado para criar uma instância da classe *Fila*.
- ✓ *processaFilaSaida()* – Método usado para processar os pacotes que se encontram na fila de saída do componente.
- ✓ *calculaTempoServico()* – Método usado para calcular quanto tempo o componente levará para transmitir o pacote pelo *Enlace.*
- ✓ *calculaTamanhoPacote()* – Método usado para calcular o tamanho do pacote no caso do mesmo apresentar uma distribuição de probabilidade para o seu tamanho.
- ✓ *inicializa()* – Método da *interface InicializaListener* que realiza o processo de inicialização do componente. No caso do componente *Enlace* ocorre também a inicialização de sua fila e de seus acumuladores.
- ✓ *transmite()* – Método da *interface TransmiteListener* usado para que o *Enlace* receba o pacote.

- ✓ *transmitePacote()* – Método responsável em transmitir o pacote passado como parâmetro.
- ✓ *retornaDadosGerador()* – Método da *interface retornaDadosListener* usado para retornar o valor obtido da distribuição de probabilidade.
- ✓ *obterDadosEnlace()* – Método da *interface ObterDadosEnlaceListener* usado para retornar os dados coletados pelo componente *Enlace* para o *ProcessadorMedidasDesempenho*.
- ✓ *tempoCorrenteRelogio()* – Método da *interface TempoCorrenteRelogioListener* usado para atualizar o tempo corrente do relógio.
- ✓ *fireRetornaDadosEnlace()* – Método usado para disparar o evento *RetornaDadosEnlaceEvent.*
- ✓ *fireDescartaPacote()* – Método usado para disparar o evento *DescartaPacoteEvent*.
- ✓ *fireAtualizaListaEvento()* – Método usado para disparar o evento *FimServicoEvent*.
- ✓ *addRetornaDadosEnlaceListener()* e *removeRetornaDadosEnlaceListener()* – Métodos que fornecem o cadastro/descadastro de *listeners* para o evento *RetornaDadosEnlaceEvent*.
- ✓ *addChegadaPacoteListener()* e *removeChegadaPacoteListener()* – Métodos que fornecem o cadastro/descadastro de *listeners* para o evento *ChegadaPacoteEvent*.
- ✓ *addDescartaPacoteListener()* e *removeDescartaPacoteListener()* – Métodos que fornecem o cadastro/descadastro de *listeners* para o evento *DescartaPacoteEvent*.

A figura 4.34 apresenta a estrutura do elemento de modelagem *Sorvedouro*.

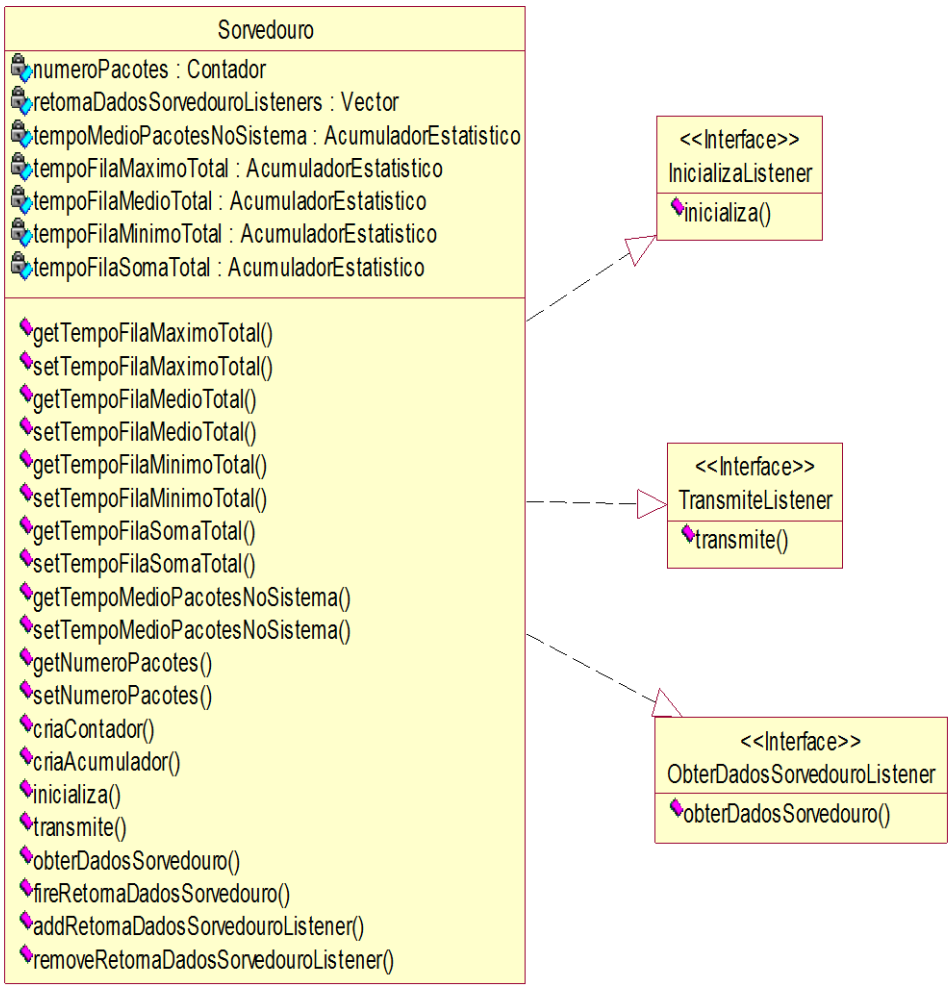

Figura 4.34: Componente Sorvedouro.

Atributos:

- ✓ *numeroPacotes* – Atributo do tipo *Contador* que armazena o número de pacotes descartados.
- ✓ *tempoMedioPacotesNoSistema* – Atributo do tipo *AcumuladorEstatistico* que armazena os tempos de todos os pacotes do sistema.
- ✓ *tempoFilaMinimoTotal* – Atributo do tipo *AcumuladorEstatistico* que armazena os tempos de fila mínimos de todos os pacotes.
- ✓ *tempoFilaMédiaTotal* – Atributo do tipo *AcumuladorEstatistico* que armazena os tempos de fila médios de todos os pacotes.

- ✓ *tempoFilaMaximoTotal* – Atributo do tipo *AcumuladorEstatistico* que armazena os tempos de fila máximos de todos os pacotes.
- ✓ *tempoFilaSomaTotal* – Atributo do tipo *AcumuladorEstatistico* que armazena todas as somas referentes aos tempos de fila de todos os pacotes.
- ✓ *retornaDadosSorvedouroListeners* – *Vector* em *Java* usado para armazenar os *listeners* do *RetornaDadosEnlaceEvent.*

Métodos:

- ✓ *SetTempoFilaMaximoTotal()* e *getTempoFilaMaximoTotal()* – Definem a propriedade *tempoFilaMaximoTotal*.
- ✓ *SetTempoFilaMinimoTotal()* e *getTempoFilaMinimoTotal()* – Definem a propriedade *tempoFilaMinimoTotal*.
- ✓ *SetTempoFilaMediaTotal()* e *getTempoFilaMediaTotal()* – Definem a propriedade *tempoFilaMediaTotal*.
- ✓ *SetTempoFilaSomaTotal()* e *getTempoFilaSomaTotal()* – Definem a propriedade *tempoFilaSomaTotal*.
- ✓ *SetTempoMedioPacotesNoSistema()* e *getTempoMedioPacotesNoSistema()* – Definem a propriedade *tempoMedioPacotesNoSistema*.
- ✓ *setNumeroPacotes()* e *getNumeroPacotes()* – Definem a propriedade *numeroPacotes.*
- ✓ *criaContador()* – Método utilizado para criar uma instância da classe *Contador.*
- ✓ *criaAcumulador()* – Método utilizado para criar uma instância da classe *AcumuladorEstatístico*.
- ✓ *inicializa()* – Método da *interface InicializaListener* que realiza o processo de inicialização do componente. No caso do componente *Sorvedouro* a inicialização consiste em inicializar os seus contadores e acumuladores.
- ✓ *transmite()* – Método da *interface transmiteListener* usado para que o componente *Sorvedouro* receba o pacote.

- ✓ *obterDadosSorvedouro()* – Método da *interface ObterDadosSorvedouroListener* usado para retornar os dados coletados pelo componente *Sorvedouro* para o *ProcessadorMedidasDesempenho*.
- ✓ *fireRetornaDadosSorvedouro()* – Método usado para disparar o evento *RetornaDadosSorvedouroEvent.*

Capítulo 5

Implementação e Testes

Este capítulo apresenta informações relevantes sobre a implementação dos componentes de software e do ambiente de simulação construído (estudo de caso). Também apresenta informações sobre os testes realizados para fins de verificação da implementação.

5.1 Introdução

Seguindo o processo de desenvolvimento orientado a objetos descrito em [Larman, 98], após a fase de Elaboração (análise e projeto do sistema) vem a fase de Construção do Sistema (codificação e testes). Esta etapa do desenvolvimento envolve as tarefas de codificar as classes resultantes das fases anteriores em uma linguagem de programação, preferencialmente orientada a objetos, e de promover testes, a fim de verificar o correto funcionamento das classes e do sistema como um todo.

Dois tipos de componentes foram especificados e implementados: os que permitem a construção de ambientes de simulação orientados a eventos (relógio simulado, lista de eventos, geradores de valores aleatórios, controle da simulação, entre outros), e os que representam elementos essenciais de uma rede de computadores *TCP/IP* (fontes de pacotes, *hosts*, *enlaces* e roteadores). Como estudo de caso, um ambiente de simulação de redes *TCP/IP* foi construído, validando a implementação dos componentes apresentada.

Foi adotado um processo iterativo e incremental de desenvolvimento de software tendo como ponto de partida as especificações, aos níveis de análise e de projeto, apresentados em [Lula, 01] e [Wagner, 00], utilizando *UML*. A tecnologia para a implementação dos componentes escolhida foi *Java* (*JavaBeans*) [Sun, 02] e o Ambiente de Desenvolvimento utilizado foi *Jbuilder* (versão 5.0) da [Borland, 02].

O usuário dos componentes implementados é o desenvolvedor de ambientes de simulação que, através de uma ferramenta visual, poderá construir um ambiente de simulação simplesmente configurando e "conectando" os componentes através de suas *interfaces.*

Durante o processo de implementação descrito neste trabalho, foram realizados alguns testes procurando, assim, garantir a qualidade do produto final. A seção 5.3 descreve mais detalhadamente a verificação de código realizada.

5.2 Fase de Implementação

5.2.1 Linguagem de Programação

A linguagem de programação *Java* é portável e possui uma extensa biblioteca de classes padrão possibilitando um ambiente de execução independente de plataforma, que permite que a aplicação seja criada uma vez e executada em qualquer lugar, favorecendo a reutilização de software. Essas características associadas ao mecanismo de reflexão [Eckel, 00] que a linguagem oferece, tornam *Java* extremamente adequada à criação de componentes reutilizáveis ao possibilitar que os desenvolvedores colham os benefícios do desenvolvimento rápido de aplicações, montando componentes predefinidos para criar aplicativos e *applets* poderosos [Deitel, 01].

5.2.2 Modelo de Componentes

A linguagem *Java* oferece dois modelos distintos para que componentes possam ser desenvolvidos [Sauvé, 00]:

- ✓ *JavaBeans*, para componentes baseados em eventos, freqüentemente, mas não necessariamente com uma representação visual; e
- ✓ *enterprise JavaBeans,* para *Server Components*, ou seja, componentes que executam no lado do servidor.

Na implementação dos componentes foi escolhido o modelo de componentes *JavaBeans* uma vez que se comunicam através de eventos e não serão executados em um servidor.

5.2.3 Modelo de Eventos

Os componentes devem se comunicar entre si para que o sistema funcione. Essa comunicação ocorre através de um modelo de comunicação. No modelo mais simples, conhecido como requisição-resposta, um componente chama uma operação de forma direta em outro componente e espera uma resposta imediata, ou seja, através de uma chamada síncrona. Os componentes participantes são fortemente acoplados o que inviabiliza esse modelo no desenvolvimento de componentes reutilizáveis.

O acoplamento pode ser reduzido pela substituição do modelo requisição-resposta pelo modelo baseado em eventos em que as operações são chamadas de forma indireta. Esse modelo possui fraco acoplamento e permite a utilização de chamadas síncronas e assíncronas.

No modelo baseado em eventos, um componente, conhecido como produtor, gera eventos que são entregues a todos os outros componentes que são acionados por estes eventos. Esses componentes são conhecidos como consumidores. Os eventos podem ser distribuídos através dos modelos: *Push*, *Pull* e *Misto*. No primeiro, todos os consumidores são notificados quando um evento do seu interesse ocorre. No segundo, uma fila de eventos para cada consumidor é criada pelo produtor que escalona cada evento somente quando o consumidor solicita [Neto, 01]. O último é uma combinação dos dois primeiros.

O modelo de comunicação baseado em eventos segue o padrão de projeto chamado *Observer* [Gamma et al, 95]. Este padrão descreve detalhadamente como definir uma dependência entre objetos a fim de que, quando um deles mudar de estado, todos os seus dependentes sejam notificados e atualizados automaticamente. Segundo este padrão, um componente *B* (consumidor de eventos ou *listener*), interessado em receber eventos gerados por um outro *A* (gerador de eventos ou fonte), cadastra-se neste. Então, cada vez que *A* gerar um evento, este é enviado para *B*.

Os componentes implementados neste trabalho comunicam-se através de eventos e utilizam o modelo *Push* [Neto, 01], descrito acima. Essa comunicação apresenta as seguintes características:

- ✓ Quem dispara um evento é uma fonte de eventos.
- ✓ Um objeto interessado neste evento (um *listener*) se cadastra junto a fonte.
- ✓ Todos os *listeners* são avisados do evento (*multicast*) ou apenas um *listener* (*unicast*).
- ✓ O evento pode encapsular ou não informações especiais para os consumidores.
- ✓ Uma referência de quem gerou o evento, a fonte, está encapsulada dentro do evento.

Na especificação feita foram observados três tipos de eventos responsáveis pela dinâmica da simulação: *ChegadaPacoteEvent, FimServicoEvent* e *FimSimulacaoEvent*. Estes são os únicos eventos que entram na lista de eventos do ambiente. No entanto, é importante observar com atenção que os demais eventos são necessários apenas para que os componentes se comuniquem entre si não sendo, portanto, pertinentes à dinâmica de uma simulação orientada a eventos e sim a um modelo de comunicação de componentes que, no nosso caso, é baseado no padrão *Observer*. Dessa forma, temos duas classes de eventos: (a) os eventos relacionados com a dinâmica da simulação, tais como a chegada de pacotes nos hosts e, (b) os eventos provenientes da comunicação entre os componentes (*ExecutaFonteEvent* por exemplo).

5.2.4 Ambiente de Desenvolvimento

O uso de componentes no desenvolvimento de software leva a um novo tipo de programação: a programação visual [Eckel, 00]. Sistemas complexos podem ser criados usando componentes reutilizáveis pré-fabricados que são configurados e interconectados usando uma ferramenta de desenvolvimento visual. A razão pela qual as ferramentas de programação visual têm sido tão bem sucedidas é que elas dramaticamente aceleram o processo de construir uma aplicação [Eckel, 00].

O desenvolvimento dos componentes descritos e a construção do ambiente de simulação foram feitos através do ambiente de desenvolvimento integrado chamado *Jbuilder* (versão 5.0) da *Borland* [Borland, 02], uma vez que se trata de uma ferramenta de desenvolvimento visual completa e adequada aos nossos interesses. A figura 5.1 apresenta a *interface* do *Jbuilder*.

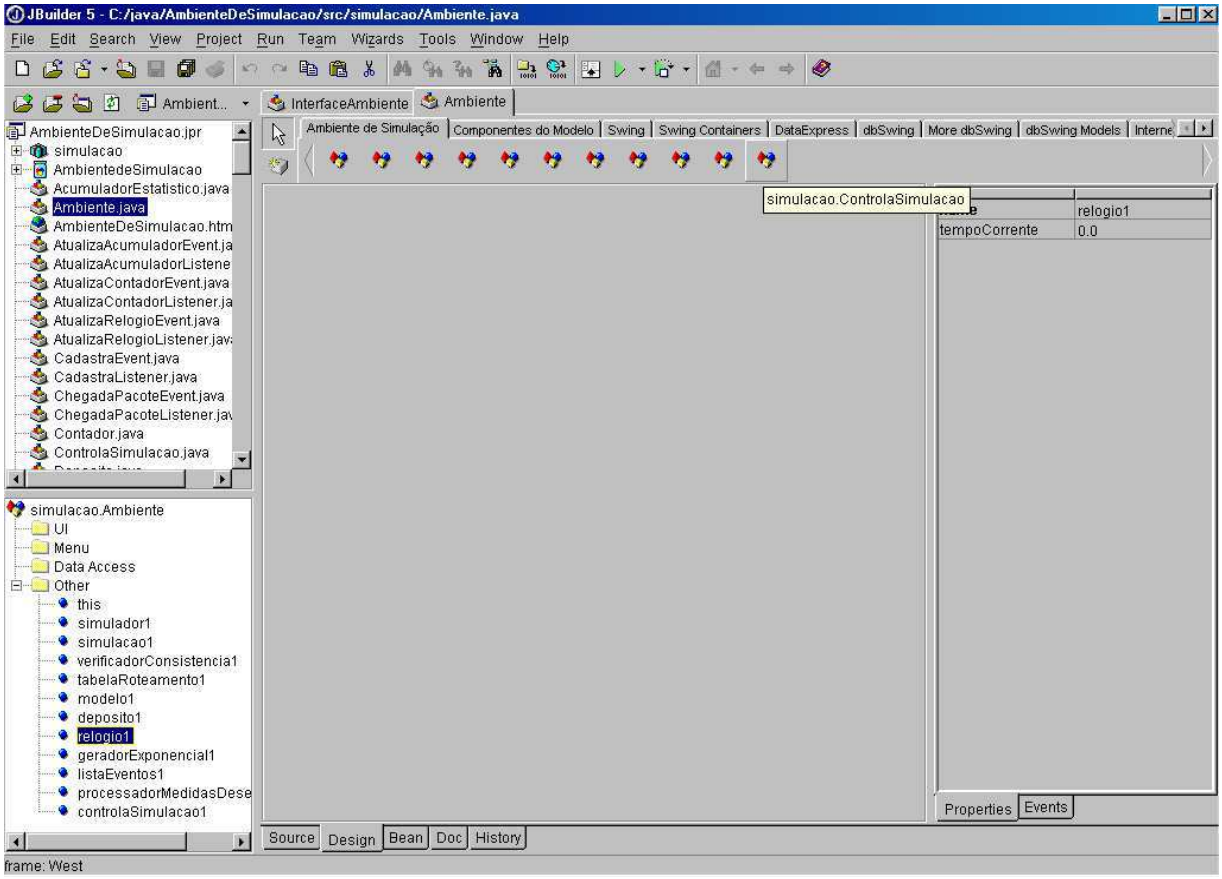

Figura 5.1: Interface do Jbuilder.

Na figura 5.1, pode-se observar a paleta de componentes chamada *Ambiente de Simulação,* onde se encontram todos os componentes necessários ao desenvolvedor do ambiente para que este possa "montar" a sua aplicação. O *Jbuilder* fornece total suporte a programação visual o que facilitou bastante, não só a implementação dos componentes, mas também a construção do ambiente.

5.2.5 Uso de Interfaces

Algumas das classes implementadas herdam de classes abstratas. Entretanto, estas classes não são "puramente abstratas" uma vez que elas já fornecem implementação para alguns de seus métodos. Herança deste tipo é conhecida como herança de classe ou de implementação. Em um projeto de software, a herança de implementação é bastante interessante na construção de uma hierarquia de classes, pois permite a reutilização de código já escrito [Santos, 01].

Essas classes abstratas foram usadas para fatorar o código comum das suas subclasses. No entanto, o polimorfismo é feito fazendo com que a classe abstrata implemente a *interface* adequada (*IEvento e IElementoModelagem*), como pode ser visto nas figuras 4.38 e 4.51. Nesse caso, a herança de *interface* resulta em um projeto de software mais extensível e com muito mais flexibilidade porque uma *interface* define tipo fazendo com que qualquer classe que a implemente tenha o seu tipo. Dessa forma, independente de qual é a classe, ela é tratada como se fosse do mesmo tipo da interface que implementa. Este é apenas uma das situações em que o polimorfismo pode ser empregado.

Outros tipos de *interfaces* presentes na implementação dizem respeito àquelas que seguem o modelo de comunicação dos *JavaBeans* (*NomedoEventoListener*). Por exemplo, na geração de um evento (*NomeDoEventoEvent*), o evento precisa ser tratado na classe ouvinte (*NomeDoEventoListener*) através da chamada do método definido por essa *interface,* que é responsável por tratar o evento em questão. Uma vez que a fonte de eventos observa todas as classes consumidoras de um evento como sendo do mesmo tipo (implementam a mesma *interface* ouvinte de eventos para o evento), a chamada do método é associada dinamicamente à implementação correspondente (Polimorfismo).

Portanto, na especificação feita foram definidas as *interfaces* para que os componentes pudessem se comunicar segundo o modelo de componentes *JavaBeans* e apenas duas outras *interfaces*, *IEvento* e *IElementoModelagem,* foram projetadas para obter polimorfismo. Essas duas *interfaces* contemplam situações passiveis de aproveitar os recursos do polimorfismo (a) no tratamento dos eventos que controlam a dinâmica da simulação e, (b) no tratamento dos elementos de modelagem. Vale observar que, para as *interfaces IEvento* e *IElementoModelagem,* foram definidas classes abstratas para fatorar o código comum das subclasses.

5.2.6 Aspectos Gerais da Implementação do Ambiente de Simulação

Além da implementação dos componentes, investiu-se na implementação de um ambiente de simulação como forma de validar os componentes implementados (um ambiente de simulação de modelos de redes *TCP/IP*). A fase correspondente à implementação desse ambiente foi feita de forma simples através da criação de uma classe *Java* chamada *Ambiente.java* que permite interligar os componentes. A figura 5.2 mostra um trecho de código dessa classe.

```
...

public class Ambiente{

//Instanciação dos componentes do ambiente de simulação
Simulador simulador1 = new Simulador();
Simulacao simulacao1 = new Simulacao();
VerificadorConsistencia verificadorConsistencia1 = new VerificadorConsistencia();
TabelaRoteamento tabelaRoteamento1 = new TabelaRoteamento();
Modelo modelo1 = new Modelo();
Deposito deposito1 = new Deposito();
Relogio relogio1 = new Relogio();
GeradorExponencial geradorExponencial1 = new GeradorExponencial();
ListaEventos listaEventos1 = new ListaEventos();
ProcessadorMedidasDesempenho processadorMedidasDesempenho1 = new ProcessadorMedidasDesempenho();
ControlaSimulacao controlaSimulacao1 = new ControlaSimulacao();
. . .

private void jbInit() throws Exception {

  //Cadastro dos Componentes

  //SIMULADOR

  //Cadastro para o evento CadastraEvent
  simulador1.addCadastraListener( simulacao1 );
  simulador1.addCadastraListener( tabelaRoteamento1 );
  simulador1.addCadastraListener( processadorMedidasDesempenho1 );
  simulador1.addCadastraListener( geradorExponencial1 );
  simulador1.addCadastraListener( controlaSimulacao1 );
  simulador1.addCadastraListener( listaEventos1 );
  simulador1.addCadastraListener( deposito1 );
```

Figura 5.2: Trecho de Código da Classe Ambiente.java.

Na figura 5.2 vemos que os componentes do ambiente são instanciados normalmente com um *new* de *Java* (feito pelo *Jbuilder* no momento da instanciação do componente). Já a ligação dos eventos foi feita de forma manual indicando "quem" seria "*listener* de quem" no código. Pode-se observar que não foi preciso configurar nenhum componente com valores iniciais, uma vez que isto ocorre de forma automática quando o ambiente esta executando um modelo.

Com o ambiente implementado através da classe *Ambiente.java*, mostrada na figura 5.2, temos um ambiente de simulação "quase" pronto, necessitando apenas dos valores iniciais para realizar a sua execução. Isto é feito por meio de uma *interface* gráfica que no caso não faz parte do escopo do presente trabalho. No entanto, como forma de viabilizar uma simulação foi criada a classe *InterfaceAmbiente.java* que interage com o componente *Simulador* realizando as ações de uma *interface* para o ambiente. Um trecho do código dessa classe é mostrado na figura 5.3.

```
  . . .
public class InterfaceAmbiente {

  . . .

  //Instanciação da clase Ambiente que representa o nosso simulador
  Ambiente ambiente = new Ambiente();

  //Instanciação pelo usuário dos componentes do modelo
  Fonte fonte1 = new Fonte();
  Host host1 = new Host();
  Enlace enlace1 = new Enlace();
  Roteador roteador1 = new Roteador();
  Sorvedouro sorvedouro1 = new Sorvedouro();
  Host host2 = new Host();
  Enlace enlace2 = new Enlace();
  Roteador roteador2 = new Roteador();

  . . .
  //Construtor da classe
  public InterfaceAmbiente() {

    //Tempo inicio, tempo final e número da replicação da simulação
     tempoInicio = new Float( "0.0 " ).floatValue();
     tempoFim = new Float( "25" ).floatValue();
     numeroReplicacao = 2;
    . . .
  }
  . . .

  public void construcaodoModelo(){

    //Inserção dos elementos da rede no componente Modelo do ambiente
    ambiente.simulador1.insereElemento( fonte1 );
    ambiente.simulador1.insereElemento( host1 );
    ambiente.simulador1.insereElemento( enlace1 );
    ambiente.simulador1.insereElemento( roteador1 );
    ambiente.simulador1.insereElemento( enlace2 );
    ambiente.simulador1.insereElemento( host2 );
    ambiente.simulador1.insereElemento( sorvedouro1 );
   . . .


    //Construção e inserção da rota no elemento tabelaRoteamento
    ambiente.simulador1.insereRota( fonte1.getId(), 1, 1, this.criaRota() );

    . . .


    //Configuração dos componentes

    //FONTE
    fonte1.setTempoPrimeiraCriacao( 0 );
    fonte1.setDistribuicaoProbabilidadeFonte( 1 );
    fonte1.setNumeroMaximoPacotes(2000 );
    fonte1.setDistribuicaoProbabilidadePacote( 1000 );
    . . .

   //Criação da rota
   rotaElementos.addElement( new Integer( fonte1.getId() ) );
   rotaElementos.addElement( new Integer( host1.getId() ) );
   rotaElementos.addElement( new Integer( enlace1.getId() ) );
   rotaElementos.addElement( new Integer( roteador1.getId() ) );
   rotaElementos.addElement( new Integer( enlace2.getId() ) );
   rotaElementos.addElement( new Integer( roteador2.getId() ) );
   rotaElementos.addElement( new Integer( enlace3.getId() ) );
   rotaElementos.addElement( new Integer( host2.getId() ) );
   rotaElementos.addElement( new Integer( sorvedouro1.getId() ) );
```

```
//Método que executa a simulação
public void execucaodaSimulacao(){
  ambiente.simulador1.executa( 0, tempoFim, numeroReplicacao );
}

 ...

//Método principal da classe InterfaceDoAmbiente.java
private void jbInit() throws Exception {

  this.construcaodoModelo();
  this.execucaodaSimulacao();
  this.apresentacaodosResultados();

}
  ...
```

Figura 5.3: Trecho de Código da Classe InterfaceAmbiente.java.

A figura 5.3 mostra o método *jbInit()* que apresenta os métodos *construcaodoModelo()*, *execuçãodaSimulacao()* e *apresentacaodosResultados()* que correspondem às fases clássicas de um processo de simulação: inicialização, execução e finalização, respectivamente.

Conforme observado no capítulo 4, a construção de um modelo corresponde à instanciação e à configuração dos componentes. No código mostrado na figura 5.3, pode-se ver que o método *construcaodoModelo(),* inicialmente, insere os elementos de modelagem no modelo invocando o método *ambiente.simulador1.insereElemento()* do componente *Simulador*. Com o modelo completo, o usuário pode definir a(s) rota(s) do modelo a ser simulado. Isto é feito pela própria *interface* que, uma vez recebendo uma rota definida, a repassa para o ambiente através da chamada do método *ambiente.simulador1.insereRota()* do componente *Simulador*.

A qualquer momento, antes de iniciar o processo de simulação, o usuário pode configurar os elementos de modelagem. No trecho do código mostrado na figura 5.3, pode-se observar como ocorreu a configuração do elemento fonte. Nessa configuração estabelece-se que: o momento da primeira criação do pacote deve ocorrer no tempo de simulação igual a 0 (zero), o número máximo de pacotes que deve ser criado é de 2000 (dois mil) e a cada 1 (uma) unidade de tempo simulado (*uts*) é criado 1 (um) pacote e o tamanho deste é de 1000 (mil) bits.

Após a execução do método *construcaodoModelo(),* passa-se a execução do método *execucaodaSimulacao()*. Essa execução é simples, bastando que a *interface* invoque o método *ambiente.simulador1.executa()* passando os parâmetros necessários.

Ao final da simulação o método *apresentacaodosResultados()* obtém do componente *Simulador* as informações colhidas durante a simulação.

5.2.7 Breve Descrição dos Componentes/Classes implementados

A tabela 5.1 apresenta um resumo das principais classes implementadas no trabalho. Pode-se observar que componentes e *interfaces* são apenas classes com funções especiais.

Nome	Descrição
Relogio	**Componente** responsável pelo armazenamento do tempo simulado. Ele implementa as *interfaces AtualizaRelogioListener, InicializaListener* e *ObterRelogioListener*
Deposito	**Componente** que permite o descarte de entidades durante a simulação e a coleta de estatísticas tais como o número de pacotes descartados durante a simulação. Ele implementa as *interfaces InicializaListener, DescadastraListener, CadastraListener* e *DescartaPacoteListener.*
Modelo	**Componente** que contém os elementos do sistema modelado (no caso, elementos de uma rede *TCP/IP*) e as rotas ativas do mesmo. Ele implementa as *interfaces ObterElementoModelagemListener, RetornaRotasAtivasListener, ObterModeloListener, ModeloVerificadoListener, InicializaListener, insereElementoModelagemListener e RemoveElementoModelagemListener.*
GeradorExponencial	**Componente** responsável pela geração de amostras de uma função de distribuição de distribuição de probabilidade exponencial com valor médio fornecido pelo usuário. Ele implementa as *interfaces ObterDadosGeradorExponencialListener,*

	CadastraListener e *DescadastraListener.*
Simulador	**Componente** que representa a *interface* entre os componentes da lógica da aplicação e a interface gráfica do simulador. Ele implementa as *interfaces FimSimulacaoModeloListener* e *ModeloErradoListener.*
Simulacao	**Componente** responsável em gerar o evento que inicializa os componentes, controlar a execução do componente *ControlaSimulacao* e em obter os resultados da simulação. Ele implementa as *interfaces ModeloVerificadoListener, InicializaSimulacaoListener, CadastraListener, DescadastraListener* e *RetornaResultadoSimulacaoListener.*
VerificadorConsistencia	**Componente** responsável pela verificação da consistência do modelo. Ele implementa a *interface VerificaModeloListener* e *RetornaModeloListener,*
TabelaRoteamento	**Componente** responsável pelo armazenamento e conseqüente manipulação das rotas do modelo. Ele implementa as *interfaces InsereRotaListener, RemoveRotaListener, CadastraListener*, *DescadastraListener, ObterRotaListener, ObterProximoNoListener, TornaRotaAtivaListener* e *ObterRotasAtivasListener.*
Rota	**Classe** responsável que contém as rotas do modelo.
ListaEventos	**Componente** responsável por armazenar os eventos que vão ser executados durante a simulação. Ele implementa as *interfaces CadastraListener, DescadastraListener, InicializaListener, FimServicoListener, EscalonaListaEventosListener,*

	ChegadaPacoteListener e *FimSimulacaoListener.*
Lista	**Classe** instanciada no componente *ListaEventos* e que representa a lista de eventos propriamente dita.
ElementoLista	**Classe** que representa um elemento da classe *Lista.* O objeto dessa classe armazena a referência a um evento.
IEvento	**Interface** implementada pela classe abstrata *Evento.*
Evento	**Classe Abstrata** que implementa a interface *IEvento.* Essa classe é estendida pelas classes *ChegadaPacoteEvent, FimServicoEvent* e *FimSimulacaoEvent* que representam os eventos que controlam a dinâmica da simulação.
Pacote	**Classe** que representa o pacote de dados que transita pela rede *TCP/IP*.
ControlaSimulacao	**Componente** responsável pela execução da simulação (escalonamento de eventos, avanço do relógio, etc.). Ele implementa as *interfaces ExecutaControlaSimulacaoListener, StatusListaEventosListener, CadastraListener, DescadastraListener, InicializaListener, RetornaProximoNoListener, RetornaRelogioTempoListener, ExecutaFonteNovamenteListener* e *RetornaEventoListener.*
ProcessadorMedidasDesempenho	**Componente** responsável pelo cálculo das medidas de desempenho. Ele implementa as interfaces *RetornaDadosDepositoListener*, *RetornaDadosFonteListener, RetornaDadosHostListener, RetornaDadosEnlaceListener,*

	RetornaDadosRoteadorListener, RetornaDadosSorvedouroListener, RetornaModeloListener, RetornaRelogioTempoListener, CadastraListener, DescadastraListener, ExecutaFimSimulacaoListener, InicializaSimulacaoListener e *ObterResultadoSimulacaoListener.*
Contador	**Classe** utilizada para contagens de dados. Por exemplo, tipo número de pacotes descartados, criados, etc. Em trabalhos futuros pode ser utilizado como um componente. Observando isso, foi fornecido todo o código para tal fazendo com que o contador implemente as *interfaces inicializaListener, AtualizaContadorListener* e *ObterDadosContadorListener.*
AcumuladorEstatistico	**Classe** responsável por armazenar dados coletados durante o processo de simulação. Da mesma forma que a classe Contador, esta classe também pode ser utilizada como componente em trabalhos futuros. Ela implementa as *interfaces InicializaListener, AtualizaAcumuladorListener* e *ObterDadosAcumuladorListener.*
IElementoModelagem	**Interface** que deve ser implementada por todos os elementos de modelagem que fizerem parte do modelo. Ela é implementada diretamente pela classe abstrata *ElementoModelagem.*
ElementoModelagem	**Classe abstrata** que deve ser estendida por todos os elementos de modelagem do modelo (no caso, os elementos de uma rede TCP/IP). Ela implementa as *interfaces InicializaListener, ObterDadosListener, RetornaDadosGeradorListener, TransmiteListener,*

	RecebePacoteListener e *TempoCorrenteRelogioListener*
Fonte	**Componente** que representa a fonte do modelo. Ele implementa as *interfaces RetornaRotaListener, executaFonteListener e ObterDadosFonteListener.* As demais *interfaces* ele herda da classe *ElementoModelagem.*
Host	**Componente** que representa os hosts (origem e destino) do modelo. Ele implementa as *interfaces ObterDadosHostListener* e *TempoCorrenteRelogioListener.* As demais *interfaces* ele herda da classe *ElementoModelagem.*
Enlace	**Componente** que representa os enlaces do modelo. Ele implementa as *interfaces TransmiteListener, ObterDadosEnlaceListener* e *TempoCorrenteRelogioListener.* As demais *interfaces* ele herda da classe *ElementoModelagem.*
Roteador	**Componente** que representa os roteadores do modelo. Ele implementa as *interfaces ObterDadosRoteadorListene* e *TempoCorrenteRelogioListener.* As demais *interfaces* ele herda da classe *ElementoModelagem.*
Sorvedouro	**Componente** que representa os sorvedouros do modelo. Ele implementa as *interfaces inicializaListener, TransmiteListener e ObterDadosSorvedouroListener.* As demais *interfaces* ele herda da classe *ElementoModelagem.*
Fila	**Classe** cujos objetos correspondem as filas dos elementos de modelagem.

Tabela 5.1: Resumo das Classes Implementadas.

As classes de eventos criadas para fornecer as interações entre os componentes tais como o *ExecutaFonteEvent*, *CadastraEvent* etc., bem como as classes que representam as *interfaces* dos respectivos eventos não serão mostradas nesta Dissertação em virtude do seu grande número. Essas classes são mostradas com detalhes no Relatório Técnico [Rocha, 02]. No total foram 130 classes, correspondendo a 65 classes de eventos e 65 *interfaces*. Todos os eventos são descritos na seção 4.2.2 (fase de análise) ao passo que as *interfaces* podem ser observadas nos diagramas de classe da seção 4.2.3.2.4 (Descrição dos Componentes Especificados). A tabela 5.2 apresenta um resumo dos resultados da implementação.

Descrição	**Quantidade**	**Linhas de Código**
Componentes implementados	16	4649
Classes Concretas	10 (incluindo as classes *Ambiente*, *IntefaceAmbiente* e *InterfaceAmbienteFrame)*	1481
Classes Abstratas	2	316
*Interface*s	2	38
Classes de eventos do tipo *NomeDoEventoEvent*	65	3053
Interfaces do tipo *NomeDoEventoListener*	65	
Classes de testes	4	350
Total	160	9987

Tabela 5.2: Resumo dos Resultados da Implementação (Linhas de Código Produzidas e Quantidade de Classes Implementadas).

5.3 Fase de Testes

Testes devem ser feitos em todas as fases de desenvolvimento de um produto [Santos, 01]. Portanto, testes devem ser feitos gradativamente à medida que o sistema está

sendo desenvolvido de forma a achar erros e eliminá-los, antes que os mesmos se propaguem para as demais fases do desenvolvimento. São vários os tipos de testes que podem ser utilizados na verificação de um software. Os principais são: testes de unidade, de integração, de sistema, funcionais e de regressão [Kaner, 99] [Dustin et al., 99].

Durante todo o processo de implementação dos componentes do ambiente de simulação e do modelo, foi realizada a verificação de código para as classes mais propensas a apresentar erros tais como *ListaEventos, Fila, Contador* e *AcumuladorEstatistico*. Estas classes são orientadas a objetos, possuindo métodos que podem retornar valores errados ou gerar efeitos inesperados no sistema como o estouro de um fila, por exemplo. Os testes dessas classes foram feitos através de uma ferramenta de testes denominada *Junit* [Beck & Gamma, 98].

Para as demais classes do sistema, em virtude de sua simplicidade em alguns casos, não foram realizados testes de unidade específicos. Essas classes, junto com as demais testadas pelo *Junit*, foram verificadas conjuntamente através de testes de sistema, com exemplos que englobam todas os componentes propostos e exploram todas as funcionalidades implementadas. Os testes de unidade e de sistemas realizados comprovaram o sucesso da implementação. O próximo capítulo apresenta os resultados da validação dos modelos simulados que comprovam o sucesso dos testes de sistemas realizados.

Capítulo 6

Validação dos Componentes e do Ambiente de Simulação

Este capítulo mostra a validação dos componentes implementados. Esta validação consistiu na comparação dos resultados de simulações de modelos de redes *TCP/IP* obtidos através do ambiente de simulação desenvolvido nesta Dissertação e através do ambiente de simulação *Arena*, versão 3.1 [Takus, 97]. Os modelos de redes TCP/IP simulados englobam todos os componentes implementados, explorando todas as suas funcionalidades.

6.1 Introdução

Nesta fase do trabalho, o objetivo principal é o de provar que o ambiente de simulação, implementado com os componentes propostos nesta Dissertação, é um "software válido", ou seja, se para um sistema de rede *TCP/IP,* é possível com a utilização desse ambiente, construir e simular modelos desse tipo de rede atendendo aos requisitos desejados.

Os modelos escolhidos como estudos de caso para validar o ambiente de simulação e, conseqüentemente, os componentes implementados, são de dois tipos básicos: determinísticos e estocásticos. Em um modelo determinístico cada entrada válida tem um efeito preciso e determinado no estado do modelo. Nesse caso, tal comportamento é simulado atribuindo valores fixos para os parâmetros de entrada dos elementos de modelagem tais

como: tamanho do pacote, tempos de serviço dos *hosts* e dos roteadores, capacidades dos *enlaces,* etc. Nos modelos estocásticos, para uma entrada válida, podemos ter várias mudanças no estado do modelo devido aos processos randômicos existentes. Nesses modelos, pode-se atribuir aos parâmetros de entrada do modelo, valores de amostras de uma função de distribuição de probabilidade.

Nos modelos construídos para a validação do ambiente de simulação, cada um deles apresenta duas versões: a primeira, usando valores fixos para os parâmetros de entrada do modelo (modelo determinístico) e a segunda versão usando valores aleatórios para esses parâmetros (modelo estocástico). No caso dos modelos determinísticos simulados no ambiente de simulação e no ambiente *Arena*, os seus resultados devem ser iguais. No entanto, para os modelos estocásticos, uma vez que lidam com valores aleatórios, os resultados podem ser diferentes desde que apresentem uma variação de resultados aceitáveis.É desejável que os percentuais de suas diferenças sejam os menores possíveis

A seção seguinte apresenta os modelos simulados no ambiente *Arena* e no ambiente de simulação com os seus respectivos resultados. A limitação do ambiente *Arena,* versão 3.1, usado em sua versão acadêmica, não permite mais de 150 (cento e cinqüenta) entidades no modelo simulado. Essa limitação restringe número de componentes de cada modelo e também pode limitar o tempo de simulação máximo para cada modelo. Essa restrição, contudo, não se aplica ao ambiente de simulação, foco desta Dissertação.

6.2 Estudos de caso

6.2.1 Modelo 1

A topologia do primeiro modelo é mostrada na figura 6.1. Nessa figura, assim como nas demais apresentadas neste capítulo, os quadrados representam os roteadores ao passo que os círculos representam os demais elementos. As linhas duplas representam as *interfaces* de comunicação entre a camada de transporte da arquitetura *TCP/IP* com a camada *IP.* Por fim, as linhas simples representam os *enlaces*.

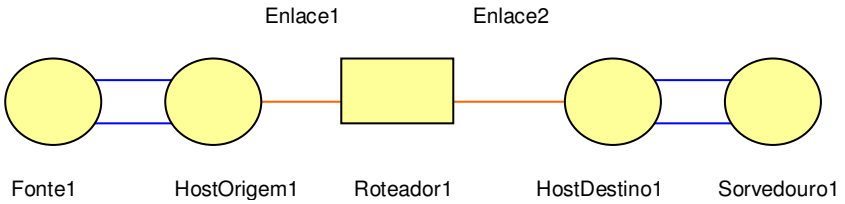

Figura 6.1: Topologia do Modelo 1.

Os modelos foram construídos através da classe *InterfaceAmbiente.java* que representa a *interface* do ambiente. Adotamos a convenção *uts* para unidade de tempo simulado.

O modelo da figura 6.1 apresenta os seguintes valores de entrada: tempo inicial da simulação igual a 0 (zero) *uts*, tempo final da simulação igual 100 (cem) *uts* e o número de replicação igual a 1 (uma). A seguir, apresentamos os resultados obtidos da simulação do modelo da figura 6.1 usando valores determinísticos e valores aleatórios.

a) **Modelo Determinístico**

Os valores dos parâmetros de entrada fornecidos pelo usuário para os elementos do modelo, através da classe *InterfadeAmbiente.Java,* são apresentados na tabela 6.1.

	Tempo da Primeira Criação (*uts*)	Intervalo de Interchegada (*uts*)	Tempo de Serviço (*uts*)	Capacidade (*bits*)	Tamanho do Pacote (*bits*)	N° Max. de Pacotes
Fonte1	0	1	---	---	1000	2000
HostOrigem1	---	---	2	---	---	---
Enlace1	---	---	---	2000	---	---
Roteador1	---	---	4	---	---	---
Enlace2	---	---	---	5000	---	---
HostDestino1	---	---	3	---	---	---

Tabela 6.1: Parâmetros de Entrada dos Elementos de Modelagem do Modelo 1 (Determinístico).

Os elementos de modelagem podem ter qualquer tamanho de fila de entrada ou de saída. No entanto, nos modelos simulados, foi adotado que o tamanho das filas de entrada e de saída dos elementos correspondem a uma capacidade de 100 (cem) posições.

Em seguida, apresenta-se o mesmo modelo simulado no ambiente *Arena* usando os mesmos valores para os parâmetros de entrada. A figura 6.2 mostra a interface do *Arena* com o modelo correspondente ao da figura 6.1.

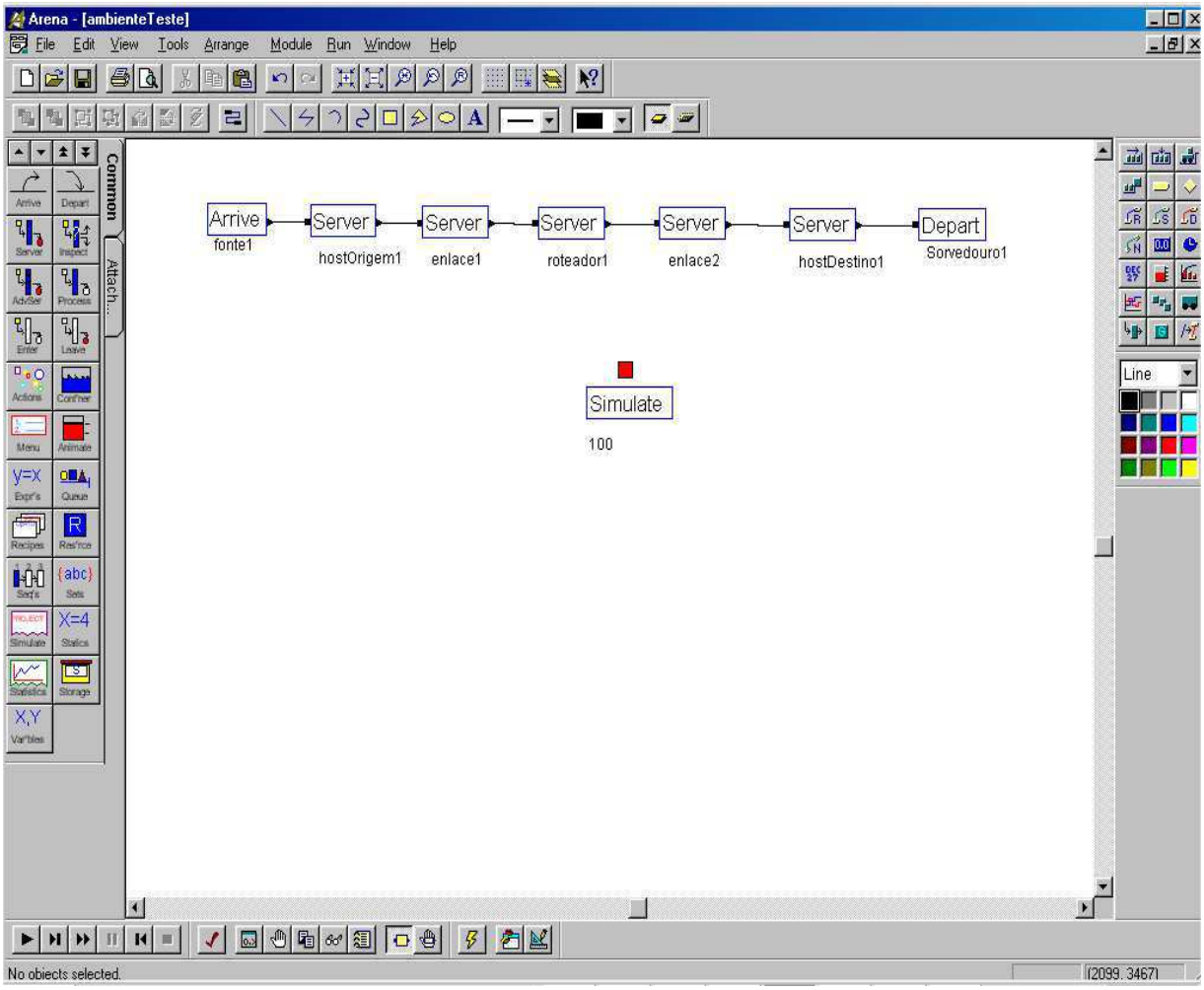

Figura 6.2: Modelo 1 no Ambiente Arena.

As telas de apresentação dos resultados obtidos pelo ambiente *Arena* e dos resultados do ambiente de simulação desenvolvido são mostradas nas figuras 6.3 e 6.4 respectivamente. Como se pode observar, as informações fornecidas pelo ambiente de simulação são superiores em número e em detalhes quando comparadas com aquelas fornecidas pelo ambiente *Arena.*

ARENA Simulation Results

Summary for Replication 1 of 1

Replication ended at time : 100.0

TALLY VARIABLES

Identifier	Average	Half Width	Minimum	Maximum	Observations
Sorvedouro1_Ta	42.700	(Insuf)	9.7000	75.700	23
hostOrigem1_R_Q Queue	25.000	(Insuf)	.00000	50.000	51
enlace1_R_Q Queue Time	00000	(Insuf)	.00000	.00000	50
hostDestino1_R_Q Queue	00000	(Insuf)	.00000	.00000	24
enlace2_R_Q Queue Time	00000	(Insuf)	.00000	.00000	24
roteador1_R_Q Queue Ti	24.000	(Insuf)	.00000	48.000	25

DISCRETE-CHANGE VARIABLES

Identifier	Average	Half Width	Minimum	Maximum	Final Value
# in hostDestino1_R_Q	.00000	(Insuf)	.00000	.00000	.00000
roteador1_R Busy	.97500	(Insuf)	.00000	1.0000	1.0000
hostOrigem1_R Busy	1.0000	(Insuf)	.00000	1.0000	1.0000
# in roteador1_R_Q	11.880	(Insuf)	.00000	25.000	24.000
hostDestino1_R Availab	1.0000	(Insuf)	1.0000	1.0000	1.0000
# in hostOrigem1_R_Q	25.000	(Insuf)	.00000	50.000	50.000
# in enlace2_R_Q	.00000	(Insuf)	.00000	.00000	.00000
enlace2_R Available	1.0000	(Insuf)	1.0000	1.0000	1.0000
roteador1_R Available	1.0000	(Insuf)	1.0000	1.0000	1.0000
# in enlace1_R_Q	.00000	(Insuf)	.00000	.00000	.00000
enlace1_R Available	1.0000	(Insuf)	1.0000	1.0000	1.0000
hostOrigem1_R Availabl	1.0000	(Insuf)	1.0000	1.0000	1.0000
hostDestino1_R Busy	.70300	(Insuf)	.00000	1.0000	1.0000
enlace2_R Busy	.04800	(Insuf)	.00000	1.0000	.00000
enlace1_R Busy	.24500	(Insuf)	.00000	1.0000	1.0000

COUNTERS

Identifier	Count	Limit
Sorvedouro1_C	23	Infinite

Simulation run time: 0.00 minutes.
Simulation run complete.

Figura 6.3: Resultado da Simulação do Modelo 1 (Determinístico) no Arena.

Resultado da Simulação

Sumário da Replicação 1 de 1

Tempo da Simulacao: 100.0

Resultados dos Acumuladores Estatísticos

Tamanho dos Pacotes no Sistema

Identificador	Mínimo	Média	Máximo	Estado	Processados	Total
HostOrigem1	1000	1000,000	1000	OCUPADO	50	51
HostDestino1	1000	1000,000	1000	OCUPADO	23	24
Roteador1	1000	1000,000	1000	OCUPADO	24	25

Tempos Gerais do Ambiente

Identificador	Mínimo	Média	Máximo	Observações
Tempo médio dos pacotes no sistema:	9,700	42,700	75,700	23
Tempo de fila máximo no sistema	0,000	22,000	44,000	23
Tempo de fila mínimo no sistema:	0,000	0,000	0,000	23
Tempo de fila médio no sistema	0,000	6,600	13,200	23
Soma do tempo de fila sistema:	0,000	33,000	66,000	23

Tempos Coletados nos enlaces

Identificador	Mínimo	Média	Máximo	Estado	Processados	Total
Tempo de transmissão (Enlace1):	0,500	0,500	0,500	OCUPADO	49	50
Tempo de transmissão (Enlace2):	0,200	0,200	0,200	LIVRE	24	24

Tempos dos Atrasos nas Filas

Identificador	Mínimo	Média	Máximo	Pacotes em Fila	Processados	Total
HostOrigem1						
Fila de Entrada :	0,000	1,941	2,000	1	51	52
Fila de Saida(Enlace1)	0,000	0,000	0,000	0	50	50
HostDestino1						
Fila de Entrada :	0.000	0.000	0.000	0	24	24
Roteador1						
Fila de Entrada :	0,000	3,760	4,000	1	25	26
Fila de Saida(Enlace2:)	0,000	0,000	0,000	0	24	24

Resultados dos Contadores

Contadores Gerais

Identificador	Observações
NumeroPacotesGerados(Fonte1)	101
NumeroPacotesDestino(Sorvedouro1)	23

Pacotes Descartados

Identificador	Observações
HostOrigem1	49
Roteador1	23
Total de Pacotes Descartados	72

```
Fator de Utilização

Identificador                                   Observações
-------------------------------------------------------------------------------------------
HostOrigem1                                     100%
HostDestino1                                    70,3%
Roteador1                                       97,5%
Enlace1                                         24,5%
Enlace2                                          4,8%

Simulação 1 executada com êxito
===========================================================================================
```

Figura 6.4: Resultado da Simulação do Modelo 1 (Determinístico) no Ambiente de Simulação.

As tabelas 6.2 e 6.3 mostram um resumo dos principais resultados obtidos. A primeira compara o fator de utilização e o número de pacotes observados. Já a segunda tabela apresenta os resultados dos atrasos nas filas de entrada e de saída dos elementos de modelagem, bem como os atrasos totais de sistema tais como o tempo mínimo, médio e máximo que o pacote permaneceu no sistema. Como se pode observar, o desvio (diferença do valor obtido do ambiente pelo valor obtido do *Arena*) foi igual a zero para todas as medidas de desempenho obtidas, comprovando o funcionamento correto do ambiente.

		Ambiente de Simulação	Ambiente Arena	Desvio (Ambiente - Arena)
Fator de Utilização	HostOrigem1	100,00 %	100,00%	0 (0%)
	Enlace1	24,50%	24,50%	0 (0%)
	Roteador1	97,50%	97,50%	0 (0%)
	Enlace2	4,80%%	4,80%	0 (0%)
	HostDestino1	70,30%	70,30%	0 (0%)
N° de Pacotes Processados ou Gerados	Fonte1	101	---	---
	HostOrigem1	51	51	0 (0%)
	Enlace1	50	50	0 (0%)
	Roteador1	25	25	0 (0%)
	Enlace2	24	24	0 (0%)
	HostDestino1	24	24	0 (0%)
	Sorvedouro1	23	23	0 (0%)

Tabela 6.2: Resultados da Simulação do Modelo 1 (Determinístico) no Ambiente de Simulação e no Ambiente Arena – Primeira Parte.

		Ambiente de Simulação				Ambiente Arena				Desvio (Ambiente -Arena)		
		Mínimo	Média	Máximo	N°	Mínimo	Média	Máximo	N°	Mínimo	Média	Máximo
Atrasos de Fila nos elementos do Sistema	HostOrigem1 (Fila de Entrada)	0,00	25,00	50,00	51	0,00	25,00	50,00	51	0 (0%)	0 (0%)	0 (0%)
	Enlace1	0,00	0,00	0,00	50	0,00	0,00	0,00	50	0 (0%)	0 (0%)	0 (0%)
	Roteador1 (Fila de Entrada)	0,00	24,00	48,00	25	0,00	24,00	48,00	25	0 (0%)	0 (0%)	0 (0%)
	Enlace2	0,00	0,00	0,00	24	0,00	0,00	0,00	24	0 (0%)	0 (0%)	0 (0%)
	HostDestino1 (Fila de Entrada)	0,00	0,00	0,00	24	0,00	0,00	0,00	24	0 (0%)	0 (0%)	0 (0%)
Atrasos no Sistema	Sorvedouro1	9,70	42,70	75,70	23	9,70	42,70	75,70	23	0 (0%)	0 (0%)	0 (0%)

Tabela 6.3: Resultados da Simulação do Modelo 1 (Determinístico) no Ambiente de Simulação e no Ambiente Arena – Segunda Parte.

b) **Modelo Estocástico**

Na configuração desse modelo alguns elementos têm parâmetros de entrada com valores obtidos de uma função de distribuição de probabilidade. O ambiente oferece um componente chamado *GeradorExponencial* que obtém amostras de uma função de distribuição exponencial com valor médio fornecido pelo usuário. É importante ressaltar que, para a utilização de outras distribuições de probabilidade, é suficiente criar os componentes das distribuições desejadas e integrá-los ao ambiente.

A tabela 6.4 mostra os valores fornecidos pelo usuário durante a configuração desse modelo. Nos modelos estocásticos, é atribuído ao intervalo de interchegada da fonte um valor fixo (não há restrições quanto a este intervalo de interchegada apresentar valores aleatórios). Esse valor fixo foi adotado apenas para simplificar a comparação dos resultados porque a cada replicação será criado pela fonte sempre o mesmo número de pacotes. Por fim, nesses modelos os componentes *enlaces* permanecem com a mesma capacidade durante toda a simulação.

	Tempo da Primeira Criação (*uts*)	Intervalo de Interchegada (*uts*)	Tempo de Serviço (*uts*) (Valor Médio)	Capacidade (*bits*)	Tamanho do Pacote (*bits*) (Valor Médio)	N° Max. de Pacotes
Fonte1	0	1	---	---	1000	2000
HostOrigem1	---	---	2	---	---	---
Enlace1	---	---	---	2000	---	---
Roteador1	---	---	4	---	---	---
Enlace2	---	---	---	5000	---	---
HostDestino1	---	---	3	---	---	---

Tabela 6.4: Parâmetros de Entrada dos Elementos de Modelagem do Modelo 1 (Estocástico).

Quanto ao tamanho dos pacotes, o ambiente de simulação permite duas abordagens: (1) a fonte gera pacotes de um mesmo tamanho que fica inalterado durante o deslocamento do mesmo através dos elementos da rede que pertencem a sua rota, ou (2) ela gera pacotes cujos tamanhos podem ser recalculados nesses elementos. No segundo caso, temos duas opções: (a) para cada pacote gerado, o seu tamanho (conforme valor de uma função de distribuição de probabilidade) é atribuído na fonte e permanece inalterado em toda a sua rota e, (b) o seu tamanho não é calculado na fonte e sim em cada elemento de modelagem de sua rota conforme valor de uma distribuição de probabilidade. A última alternativa faz com que o pacote possa apresentar um tamanho diferente para cada elemento de rede em sua rota. Esta opção pode viabilizar a comparação de resultados de modelos simulados no ambiente em validação, com estudos analíticos desses modelos. Em todos os modelos estocásticos apresentados neste capítulo, o tamanho do pacote é determinado apenas pela fonte.

O modelo 1 (estocástico), construído no ambiente *Arena,* é o mesmo já apresentado na figura 6.2, alterando apenas os valores dos parâmetros de entrada dos elementos de modelagem. Em virtude de utilizarem amostras de uma função de distribuição de probabilidade exponencial para determinar o tempo de serviço e o tamanho do pacote, os resultados da simulação do Modelo 1 (estocástico), provavelmente não serão iguais. Assim, devido às aleatoriedades presentes neste modelo, no processo de validação, foram realizadas 10 simulações no software *Arena* e no ambiente de simulação, permitido os cálculos das médias aritméticas das medidas observadas. A limitação do software *Arena*, usado em sua versão acadêmica, em não permitir mais de 150 (cento e cinquenta) entidades no sistema, limitou o tempo de simulação máximo para cada modelo.

O número de pacotes recebidos pelo elemento *Sorvedouro,* durante as dez simulações realizadas, pode ser observado na tabela 6.5. Em ambos os ambientes de simulação o *Sorvedouro* processou em média aproximadamente 22 pacotes.

Ambiente\Execução	1	2	3	4	5	6	7	8	9	10	Média
Ambiente	22	26	30	18	25	23	24	15	17	19	21,90
Arena	24	17	21	27	16	23	25	23	21	23	22,00

Tabela 6.5: Número de Pacotes Recebidos pelo Sorvedouro.

As tabelas 6.6, 6.7 e 6.8 mostram um resumo dos principais resultados obtidos. A primeira tabela compara o fator de utilização. A segunda apresenta os resultados dos atrasos totais de sistema comparando-os com o do *Arena.* Por fim, a última tabela mostra a comparação dos atrasos totais de sistema do ambiente (atrasos fim a fim) com os atrasos observados na tabela 6.3 para o modelo determinístico.

		Ambiente de Simulação	Arena	Modelo Determinístico	Desvio (Amb. Sim. -Arena)	Desvio (Amb. Sim. – Modelo Determinístico)
Fator de Utilização Médio	HostOrigem1	99,75%	99,60%	100,00 %	+0,15 (0,15%)	-0,25 (0,25%)
	Enlace1	25,88%	25,80%	24,50%	+0,08 (0,31%)	+1,38 (5,63%)
	Roteador1	92,62%	94,08%	97,50%	-1,46 (1,55%)	-4,88 (5,00%)
	Enlace2	5,13%	5,79%	4,80%%	-0,66 (11,40%)	0,33 (6,87%)
	HostDestino1	65,98%	75,92%	70,30%	-9,94 (13,09%)	-4,32 (6,14%)

Tabela 6.6: Resultados da Simulação do Modelo 1 (Estocástico) – Fator de Utilização Médio.

		Ambiente de Simulação				Arena			Desvio (Ambiente - Arena)			
		Mínimo	Média	Máximo	N°	Mínimo	Média	Máximo	N°	Mínimo	Média	Máximo
Atraso Médio no Sistema	Sorvedouro 1	6,86	42,96	75,66	21,90	9,60	45,37	76,25	22,00	-2,74 (28,54%)	-2,41 (5,31%)	-0,59 (0,77%)

Tabela 6.7: Resultados da Simulação do Modelo 1 (Estocástico) – Atraso Médio no Sistema.

		Ambiente de Simulação				Determinístico				Desvio (Ambiente - Determinístico)		
		Mínimo	Média	Máximo	N°	Mínimo	Média	Máximo	N°	Mínimo	Média	Máximo
Atraso Médio no Sistema	Sorvedouro 1	6,86	42,96	75,66	21,9	9,70	42,70	75,70	23,0	-2,84 (29,28%)	-0,26 (0,61%)	-0,04 (~0,00%)

Tabela 6.8: Resultados da Simulação do Modelo 1 (Estocástico) – Atraso Médio no Sistema.

Como se pode observar o ambiente de simulação apresentou resultados que comprovam a sua validação.

6.2.2 Modelo 2

O segundo modelo apresentado como estudo de caso tem a topologia mostrada na figura 6.5.

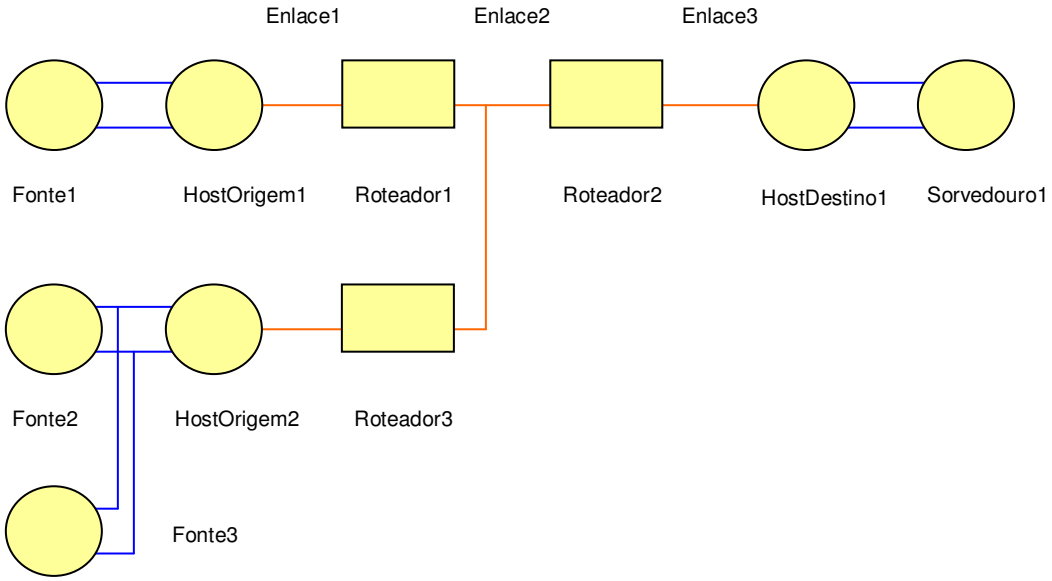

Figura 6.5: Topologia do Modelo 2.

O modelo da figura 6.5 apresenta os seguintes valores para os parâmetros de entrada: tempo inicial da simulação igual a 0 (zero) *uts*, tempo final da simulação igual 50 (cinqüenta) *uts* , número de replicação igual a 1 (uma) e tamanho de todas as filas (entradas e saídas) igual a 100 (cem). A seção *a* apresenta os resultados obtidos da simulação usando

valores determinísticos ao passo que a seção *b* usa valores aleatórios caracterizando um modelo estocástico.

a) **Modelo Determinístico**

Os valores de entrada fornecidos pelo usuário para os elementos do modelo através da classe *InterfadeAmbiente.java* são apresentados na tabela 6.9.

	Tempo da Primeira Criação (*uts*)	Intervalo de Interchegada (*uts*)	Tempo de Serviço (*uts*)	Capacidade (bits)	Tamanho do Pacote (*bits*)	N° Máximo. de Pacotes
Fonte1	0	1	---	---	1000	2000
Fonte1	0	1	---	---	1000	2000
Fonte1	0*s*	1	---	---	1000	2000
HostOrigem1	---	---	1	---	---	---
HostOrigem2	---	---	1	---	---	---
Enlace1	---	---	---	5000	---	---
Enlace2	---	---	---	5000	---	---
Enlace3	---	---	---	5000	---	---
Enlace4	---	---	---	5000	---	---
Roteador1	---	---	1	---	---	---
Roteador2	---	---	1	---	---	---
Roteador3	---	---	1	---	---	---
HostDestino1	---	---	1	---	---	---

Tabela 6.9: Parâmetros de Entrada dos Elementos de Modelagem do Modelo 2 Determinístico.

A figura 6.6 mostra o modelo simulado no ambiente *Arena* correspondente ao Modelo 2 apresentado na figura 6.5.

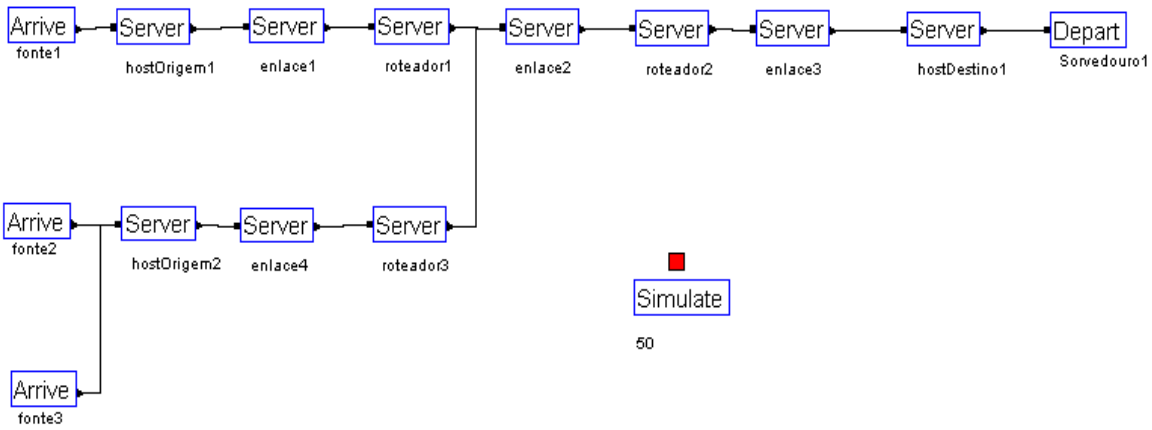

Figura 6.6: Modelo 2 no Ambiente Arena.

As tabelas 6.10 e 6.11 mostram um resumo dos principais resultados obtidos. A primeira tabela compara o fator de utilização. Já a segunda tabela apresenta os resultados dos atrasos nas filas de entrada e de saída dos elementos de modelagem bem como os atrasos totais de sistema tais como o tempo mínimo, médio e máximo que o pacote permaneceu no sistema. Como se pode observar, o desvio foi igual a zero para todas as medidas comprovando o funcionamento correto do ambiente de simulação.

		Ambiente de Simulação	Arena	Desvio (Ambiente -Arena)
Fator de Utilização	HostOrigem1	100,00 %	100,00%	0 (0%)
	HostOrigem2	100,00 %	100,00%	0 (0%)
	Enlace1	19,60%	19,60%	0 (0%)
	Enlace2	38,40%	38,40%	0 (0%)
	Enlace3	18,80%	18,80%	0 (0%)
	Enlace4	19,60%	19,60%	0 (0%)
	Roteador1	97,60%	97,60%	0 (0%)
	Roteador2	95,20%	95,20%	0 (0%)
	Roteador3	97,60%	97,60%	0 (0%)
	HostDestino1	92.80%	92,80%	0 (0%)

Tabela 6.10: Resultados da Simulação do Modelo 2 (Determinístico) no Ambiente de Simulação e no Ambiente Arena – Primeira Parte.

		Ambiente de Simulação				Arena				Desvio (Ambiente -Arena)		
		Mínimo	Média	Máximo	N°	Mínimo	Média	Máximo	N°	Mínimo	Média	Máximo
Atrasos de Fila nos elementos do Sistema	HostOrigem1 (Fila de Entrada)	0,000	0,000	0,000	51	0,000	0,000	0,000	51	0 (0%)	0 (0%)	0 (0%)
	HostOrigem2 (Fila de Entrada)	0,000	12,745	25,000	51	0,000	12,745	25,000	51	0 (0%)	0 (0%)	0 (0%)
	Enlace1	0,000	0,000	0,000	50	0,000	0,000	0,000	50	0 (0%)	0 (0%)	0 (0%)
	Enlace2	0,000	0,100	0,200	96	0,000	0,100	0,200	96	0 (0%)	0 (0%)	0 (0%)
	Enlace3	0,000	0,000	0,000	47	0,000	0,000	0,000	47	0 (0%)	0 (0%)	0 (0%)
	Enlace4	0,000	0,000	0,000	50	0,000	0,000	0,000	50	0 (0%)	0 (0%)	0 (0%)
	Roteador1 (Fila de Entrada)	0,000	0,000	0,000	49	0,000	0,000	0,000	49	0 (0%)	0 (0%)	0 (0%)
	Roteador2 (Fila de Entrada)	0,000	11,900	23,800	48	0,000	11,900	23,800	48	0 (0%)	0 (0%)	0 (0%)
	Roteador3 (Fila de Entrada)	0,000	0,000	0,000	49	0,000	0,000	0,000	49	0 (0%)	0 (0%)	0 (0%)
	HostDestino1 (Fila de Entrada)	0,000	0,000	0,000	47	0,000	0,000	0,000	47	0 (0%)	0 (0%)	0 (0%)
Atrasos no Sistema	Sorvedouro1	4,600	18,970	37,600	46	4.6000	18,970	37,600	46	0 (0%)	0 (0%)	0 (0%)

Tabela 6.11: Resultados da Simulação do Modelo 2 (Determinístico) no Ambiente de Simulação e no Ambiente Arena – Segunda Parte.

b) **Modelo Estocástico**

Na configuração desse modelo alguns elementos utilizam amostras de uma função de distribuição de probabilidade exponencial. A tabela 6.12 mostra os valores dos parâmetros de entrada fornecidos pelo usuário durante a configuração do modelo.

	Tempo da Primeira Criação (*uts*)	Intervalo de Interchegada (*uts*) (Valor Médio)	Tempo de Serviço (*uts*) (Valor Médio)	Capacidade (*bits*)	Tamanho do Pacote (*bits*) (Valor Médio)	N° Máximo. de Pacotes
Fonte1	0	1	---	---	1000	2000
Fonte1	0	1	---	---	1000	2000
Fonte1	0	1	---	---	1000	2000
HostOrigem1	---	---	1	---	---	---
HostOrigem2	---	---	1	---	---	---
Enlace1	---	---	---	5000	---	---
Enlace2	---	---	---	5000	---	---
Enlace3	---	---	---	5000	---	---
Enlace4	---	---	---	5000	---	---
Roteador1	---	---	1	---	---	---
Roteador2	---	---	1	---	---	---
Roteador3	---	---	1	---	---	---
HostDestino1	---	---	1	---	---	---

Tabela 6.12: Parâmetros de Entrada dos Elementos de Modelagem do Modelo 2 (Estocástico).

O modelo criado no ambiente *Arena* é o mesmo já apresentado na figura 6.6 mudando apenas os parâmetros de entrada dos elementos de modelagem.

Como já mencionado na seção 6.2.1 para o modelo 1 (estocástico), em virtude de utilizar amostras de uma função de distribuição de probabilidade exponencial para determinar o tempo de serviço e o tamanho do pacote, os resultados da simulação provavelmente não serão iguais. Assim, o modelo 2 (estocástico) foi simulado 10 (dez) vezes para viabilizar o cálculo das médias aritméticas das medidas de interesse.

O número de pacotes que chegaram ao elemento *Sorvedouro* durante as 10 (dez) simulações realizadas pode ser observado na tabela 6.13. Em ambos os ambientes, *Sorvedouro* processou em média aproximadamente 39,7 (trinta e nove vírgula sete) pacotes.

Ferramenta\Execução	1	2	3	4	5	6	7	8	9	10	Média
Ambiente	40	34	47	35	40	41	45	43	42	36	40,30
Arena	40	37	38	41	39	34	46	25	46	45	39,10

Tabela 6.13: Número de Pacotes Recebidos pelo Sorvedouro.

As tabelas 6.14 e 6.15 permitem comparações dos resultados de medidas de desempenho relevantes (fator de utilização médio e atraso médio no sistema, respectivamente), obtidos pelo ambiente de simulação com aqueles resultados obtidos pelo Ambiente *Arena*. A tabela 6.14 também permite comparações do modelo em questão com resultados obtidos do Modelo 2 (Determinístico). Por fim, a tabela 6.16 permite a comparação de resultados da medida de desempenho, atraso médio no sistema, obtida pelo ambiente de simulação com aquela obtida no Modelo 2 (determinístico), mostrada na tabela 6.11.

		Ambiente de Simulação	Arena	Modelo Determinístico	Desvio (Amb. Sim. -Arena)	Desvio (Amb. Sim. – Modelo Determinístico)
Fator de Utilização Médio	HostOrigem1	93,96%	91,20%	100,00%	+2,76 (3,03%)	-8,64 (8,64%)
	HostOrigem2	99,77%	100,00%	100,00%	-0,23 (0,23%)	-0,23 (0,23%)
	Enlace1	17,77%	17,38%	19,60%	+0,39 (2,24%)	-1,38 (9,33%)
	Enlace2	34,11%	34,53%	38,40%	-0,42 (1,22%)	-4,29 (11,18%)
	Enlace3	19,79%	19,03%	18,80%	+0,76 (4,00%)	+0,99 (5,27%)
	Enlace4	20,63%	19,11%	19,60%	+1,52 (7,95%)	+1,03 (5,25%)
	Roteador1	76,40%	84,13%	97,60%	-7,73 (9,19%)	+21,2 (21,72%)
	Roteador2	94,38%	93,20%	95,20%	+1,18 (1,27%)	-0,82 (0,86%)
	Roteador3	87,52%	83,87%	97,60%	+3,65 (4,35%)	-10,08 (10,32%)
	HostDestino1	74,72%	77,47%	92,80%	-2,75 (3,55%)	-18,08 (19,48%)

Tabela 6.14: Resultados da Simulação do Modelo 2 (Estocástico) – Fator de Utilização.

		Ambiente de Simulação				Arena				Desvio (Ambiente - Arena)		
		Mínimo	Média	Máximo	N°	Mínimo	Média	Máximo	N°	Mínimo	Média	Máximo
Atraso Médio no Sistema	Sorvedouro 1	3,09	19,77	38,30	40,30	2,69	20,93	38,18	39,10	+0,40 (14,87%)	-1,16 (5,54%)	-0,12 (0,34%)

Tabela 6.15: Resultados da Simulação do Modelo 2 (Estocástico) – Atraso Médio no Sistema.

		Ambiente				Determinístico				Desvio (Ambiente - Determinístico)		
		Mínimo	Média	Máximo	N°	Mínimo	Média	Máximo	N°	Mínimo	Média	Máximo
Atraso Médio no Sistema	Sorvedouro 1	3,09	19,77	38,30	40,30	4.60	18,97	37,60	46,00	+1,51 (38,83%)	+0,80 (4,22%)	+0,7 (1,83%)

Tabela 6.16: Resultados da Simulação do Modelo 2 (Estocástico) – Atraso Médio no Sistema.

Como se pode observar, o ambiente de simulação apresentou resultados que comprovam a sua validação.

Capítulo 7

Conclusões e Trabalhos Futuros

Dois tipos de componentes são especificados e implementados neste trabalho: os que permitem a construção de ambientes de simulação orientados a eventos (relógio simulado, lista de eventos, geradores de valores aleatórios e controle da simulação, entre outros), e os que representam os elementos essenciais de uma rede de computadores *TCP/IP* (fontes de pacotes, *hosts*, *enlaces* e roteadores). Como estudo de caso, um ambiente de simulação de redes *TCP/IP* foi construído, validando a implementação dos componentes apresentada.

Foi adotado um processo iterativo e incremental de desenvolvimento de software tendo como ponto de partida as especificações, aos níveis de análise e de projeto, apresentados em [Lula, 01] e [Wagner, 00], utilizando *UML*. A tecnologia para a implementação dos componentes escolhida foi *Java* (*JavaBeans*) [Sun, 02] e o Ambiente de Desenvolvimento utilizado foi *Jbuilder* (versão 5.0) da [Borland, 02].

O usuário dos componentes é o desenvolvedor de ambientes de simulação. Este, através de uma ferramenta visual, poderá construir um ambiente de simulação simplesmente configurando e "conectando" os componentes através de suas *interfaces.* Os componentes podem ser reutilizados na construção de qualquer ambiente de simulação orientado a eventos que se aplique a modelos que apresentam contenção de recursos, tais como aqueles que representam sistemas de computação e sistemas de comunicação de dados.

Na fase de implementação foi observado que os componentes especificados em [Lula, 01] e em [Wagner, 00] necessitavam de novas funcionalidades a fim de garantir uma implementação satisfatória e um software final robusto. Dessa forma, com base no processo iterativo e incremental, no qual se baseia este trabalho, foram refeitas essas especificações possibilitando a reutilização, não só a dos componentes implementados, objetivo maior desta Dissertação, mas também a reutilização das fases de análise e de projeto do processo de desenvolvimento utilizado.

A especificação final exigiu a construção de novos componentes não previstos inicialmente, como o componente *TabelaRoteamento*, e a eliminação de alguns, que não atendiam adequadamente as necessidades do ambiente, como o componente *Mensagem*. Outros componentes tiveram funcionalidades direcionadas, a exemplo do *GeradorVAs* que passou a ser o componente *GeradorExponencial*. Além disso, novas funcionalidades foram acrescentadas para viabilizar a implementação do ambiente de simulação tais como:

- ✓ O ambiente de simulação fornece meios para que uma rota possa estar ativa ou não. Quando ativa, pacotes são gerados e transitam pela rota. Esta facilidade permite definir um conjunto de rotas (ativas ou não) para um modelo sem a necessidade de remover as rotas inativas:
- ✓ Quanto ao tamanho dos pacotes, o ambiente de simulação permite duas abordagens: (1) a fonte gera pacotes de um mesmo tamanho que fica inalterado durante o deslocamento do mesmo através dos elementos da rede que pertencem a sua rota, ou (2) ela gera pacotes cujos tamanhos podem ser recalculados nesses elementos. No segundo caso, temos duas opções: (a) para cada pacote gerado, o seu tamanho (conforme valor de uma função de distribuição de probabilidade) é atribuído na fonte e permanece inalterado em toda a sua rota e, (b) o seu tamanho não é calculado na fonte e sim em cada elemento de modelagem de sua rota conforme valor de uma distribuição de probabilidade. A última alternativa faz com o pacote possa apresentar um tamanho diferente para cada elemento de rede em sua rota. Esta opção pode viabilizar a comparação de resultados de modelos simulados no ambiente em validação, com estudos analíticos desses modelos.

Além dessas novas facilidades oferecidas pelos componentes, outras medidas de desempenho puderam ser obtidas, aumentando a riqueza das informações fornecidas pelo

ambiente de simulação quando um modelo é simulado. Entre as medidas de desempenho que podem ser obtidas, ressaltamos: Atrasos de sistema, fator de utilização dos componentes, estado em que um componente se encontra (livre, ocupado, congestionado ou inativo), numero de entidades processadas, descartadas, ou em fila em cada componente do modelo.

A validação dos componentes implementados foi feita através da construção de um ambiente de simulação para redes *TCP/IP*, completamente funcional, o que resultou em uma contribuição a mais da presente Dissertação. A validação consistiu na comparação dos resultados de simulações de modelos de redes *TCP/IP* obtidos através do ambiente de simulação desenvolvido nesta Dissertação com os resultados obtidos no ambiente *Arena*, versão 3.1 [Takus, 97]. Os modelos de redes *TCP/IP* simulados englobam todos os componentes implementados, explorando todas as suas funcionalidades.

Ao todo foram implementadas 9987 linhas de código, divididas entre 160 classes com um total de 16 componentes. Durante a implementação a verificação de código realizada foi através de testes das classes mais propensas a erros (*Rota*, *Lista*, *Fila* etc). Os componentes e as demais classes foram testados de maneira informal através de um teste de sistema, viabilizado através do ambiente desenvolvido como estudo de caso. Os resultados das simulações dos modelos apresentados no capítulo 6 mostraram que os requisitos funcionais, apresentados na fase de especificação, foram atendidos e que os componentes foram implementados corretamente. Esses resultados também validaram o ambiente de simulação construído.

7.1 Trabalhos futuros

De imediato, visando à continuação dos trabalhos realizados nesta Dissertação, vem a especificação e implementação de uma *interface* gráfica para o ambiente de simulação de redes *TCP/IP* construído como estudo de caso. Esse ambiente gráfico poderá facilitar a construção do modelo a ser simulado e também a análise do desempenho dos elementos de modelagem durante e após a simulação, através de gráficos e outros elementos visuais.

Devido aos componentes propostos serem considerados essenciais a uma simulação, estes podem ser reutilizados na construção de qualquer ambiente de simulação orientada a eventos, independentemente do tipo de modelo, seja este representando redes de

computadores, um sistema de manufatura ou outro sistema discreto com contenção de recursos.

Outro trabalho a ser realizado refere-se à implementação de novas funcionalidades nos componentes desenvolvidos para que ambientes de simulação construídos com estes possam oferecer as seguintes facilidades:

- ✓ Mecanismos para calcular medidas de desempenho não definidas pelo ambiente de simulação a pedido do usuário. Para este requisito, as classes *Contador* e *AcumuladorEstatistico* já implementam recursos que as podem transformar em componentes, caso necessário.
- ✓ O ambiente de simulação, ora construído, permite que um mesmo modelo possa ser simulado mais de uma vez, obtendo amostras de medidas de desempenho para cada simulação realizada. Como extensão dessa facilidade, sugere-se que os ambientes de simulação, a serem construídos, possam calcular automaticamente valores médios para as medidas de desempenho, como também encontrar intervalos de confiança para esses valores.
- ✓ Os ambientes de simulação deverão oferecer novas funções de distribuição de probabilidade para a obtenção de valores aleatórios. A inserção de uma nova função de distribuição terá um efeito mínimo no código do sistema, em virtude de se tratar de componentes de software reutilizáveis.
- ✓ Os ambientes de simulação deverão ser capazes de prover um mecanismo de acompanhamento passo a passo do comportamento do modelo durante a simulação.

Por fim , um *framework* de um ambiente de simulação poderá ser desenvolvido com base nos ambientes de simulação que vierem a ser construídos através destes componentes.

Referências Bibliográficas

[Alaettinoglu, 98] Alaettinoglu, C.; shankar, A.; Dussa-Zieger, K. Matta, I.; *"Design and Implementation of MaRS: A Routing Testbed"*; Journal of Internetworking: Research & Experience"; vol. 5 nº 1, 17-41, 1994.

[Almeida, 99] Almeida, Marcelo J. S. C.; *"ATMLib- Uma biblioteca de classes para construção de Simuladores de Rede ATM"*, UFPB,1999.

[Beck & Gamma, 98] BECK, Kent, GAMMA, Erich. *"Test Infected: Programmers Love Writing Testes"*,. 1998,. Disponível em: http://www.junit.org/.

[Borland, 02] Borland. *"Building Applications with JBuilder"*, 2002, disponível em: http://www.borland.com/techpubs/jbuilder .

[Cabral, 92] Cabral, M. I. C.;*"Um Ambiente de Simulação Inteligente para avaliar o desempenho de Sistemas Distribuídos"*, Projeto de Pesquisa, CNPq/UFPB, Campina Grande, 1992.

[Cadence, 98] Cadense Inc.; *"BONeS Simulator"*; http://www.cadence.com/alta/products/bonesdat.html, 1998.

[Celestino, 90] Celestino, J.; *"Avaliação de Desempenho em Redes Locais Brasileiras"*, Dissertação de Mestrado, CCT, Universidade Federal da Paraíba, Campina Grande, fevereiro de 1992.

[Clocksin, 81] Clocksin, W. F.; Mellish, C. S.; *"Programming in Prolog"*, Berlim: Springer-Verlag, 1981

[D'Souza, 98] D'Souza Desmond, F.; Wills, Alan C.; *"Objects, Components and Frameworks with UML: The Catalysis Approach"*; Addison-Wesley, 1998.

[Damoun, 79] Damoun, F., Kleinrock, L.; *"Stochastic Performance Evaluation of Hierarchical Routing for Large Networks"*, Computer Networks, vol. 3. Pp. 337-353, novembro de 1979.

[Deitel, 01] Deitel, H. M., Deitel, P.J; *"JavaTM Como Programar"*, 3a edição, Porto Alegre, Bookman, 2001.

[DeSoto, 97] DeSoto, Alden; *"Using the Beans Development Kit 1.0"*, A Tutorial, 1997.

[Dias, 92] Dias, Maria Madalena; *"SIMILE - Um Simulador Reutilizável para Avaliação de Desempenho de Redes Locais"*, Dissertação de Mestrado, CCT, Universidade Federal da Paraíba, Campina Grande, abril de 1992.

[Dustin et al., 99] Dustin, E., Rashka, J. S., Paul, J.; *"Automated Software Testing: Introduction, Managemant and Performance"*, Addison-Wesley, Massachussets, USA, 1999.

[Eckel, 00] Eckel, Bruce; *"Thinking in Java"*, 2a edição, Revision 12, New

Jersey, Prentice-Hall, 1978.

[Englander, 97] Englander, Robert; *"Developing JavaBeans"*, Ed O'Reilly, Junho 1997.

[Freire, 00] Freire, Raissa Dantas; *"Especificação de um Framework baseado em Componentes de Software Reutilisáveis para Aplicações de Gerência de Falhas em Redes de Computadores"*, Dissertação de Mestrado, UFPB, 2000.

[Gamma et al., 95] Gamma, E.,Helm R., Johnson R., Vlissides J.; *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison-Wesley, Massachussets, USA, 1995.

[Golmie, 95] Golmie, Nada; Koenig, Alfred and Su, David; *"The NISR – ATM Network Simulator – Operation and Programming"*, National Institute of Standards and Technology, August 1995.

[Howell, 97] Howell, Fred; McNab, Ross; *"Simjava: A Discrete Event Simulation Package for Java with Applications in Computer Systems Modelling"*; Department of Computer Science –The University of Edinburgh, Scotland – UK, 1997.

[Kaner, 99] Kaner, C.; *"Testing Computer Software"*, 2nd Edition, John Wiley, USA, 1995.

[Kelton, 98] Kelton, W. D. et al.; *"Simulation with ARENA"*, McGraw-Hill, 1998.

[Kronbauer, 98] Kronbauer, Artur H.; *"Avaliação de Protocolos Multicast em Redes TCP/IP"*, Dissertação de Mestrado, CCT, Universidade Federal da Paraíba, Campina Grande, junho de 1998.

[Landin, 98] Landin, N., Niklasson, A.; *"Development of Object-Oriented Frameworks"*, Department of Communication Systems, Lund Institute of Technology, Sweden, 1998.

[Larman, 98] Larman, C.; *"Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design"*, Prentice-Hall, 1998.

[Lula, 01] Lula, Juliana C.L.A.; *"Especificação de Componentes para um Ambiente de Simulação de Redes TCP/IP"*, Dissertação de Mestrado, UFPB, 2001.

[ModSim-II, 89] MODSIM-II; *"MODSIM-II: An Object-Oriented Simulation Language for Sequencial and Parallel Processors"*, Proceedings of the 1998 Winter Simulation Conference, Piscataway, NJ, pp 172-189, 1989.

[Neto, 01] Neto, Alberto Costa; *"Projeto e Implementação de um Serviço de Eventos para o Desenvolvimento de Aplicações Baseadas em Componentes"*, Dissertação de Mestrado, UFPB, 2001.

[NS, 98] NS, *"Network Simulator"*; disponível em :http://www.mash.cs.berkeley.edu/ns, 1998.

[Oliveira, 95] Oliveira, S. R DE M.; *"ALLOS - Uma Ferramenta para Solucionar*

Modelos de Redes de Filas Usando Cadeias de Markov"; Dissertação de Mestrado. CCT, Universidade Federal da Paraíba, junho de 1995

[Pegdrm, 95] Pegdrm, C.D, Shannom, R. E.,Sadowski, R; *"Introduction to Simulation Using SIMAN"*; Mc-Graw-Hill, Inc., 1995.

[Renshaw, 98] Renshaw, David S.; *"JavaBeans: The Perfect Roast?",* Hursley Laboratory, IBM, Junho, 1998.

[Roberts, 94] Roberts, Chell; Dessouky, Yasser; *"Na Overview of Object-Oriented Simulation"*; SIMULATION, Number 70, pp. 359-368, june, 1998.

[Rumbaugh et al., 99] Rumbaugh, J., Booch, G., Jacobson, I.; *"The Unified Modeling Language Reference Manual"*, Addison-Wesley, 1999.

[Rumbaugh, 91] Rumbaugh, James; Blaha, Michael; William, Premerlani; Eddy, Fredirck; Lorensen, Willian; *"Object-Oriented Modeling and Design"*, Prentice-Hall, 1991.

[Santos, 01] Santos, Giovanni Almeida; *"Evolução de um Framework para a Construção de Aplicações de Gerência de Falhas em Redes de Computadores",* Dissertação de Mestrado, CCT, Pós-Graduação em Informática, Universidade Federal da Paraíba, Campina Grande, 2001.

[Sauvé, 00] Sauvé, Jacques; "*Análise e Projeto de Sistemas Orientados a Objeto"*, Disciplina do curso de Pós Graduação em Informática; Universidade Federal da Paraíba; Material on-line http://vulcano.dsc.ufpb.br/jacques/cursos/2000.1/apoo/ ; período 1º semestre, ano 2000.

[Silva, 00] Silva, Ricardo Pereira; *"Suporte ao Desenvolvimento e Uso de Frameworks e Componentes"*; Tese de Doutorado, UFRGS/II/PPGC, Porto Alegre: março de 2000.

[Soares, 90] Soares, Luiz Fernando G.; *"Modelagem e Simulação Discreta de Sistemas"*, VII Escola de Computação – São Paulo, 1990.

[Souto, 93] Souto, F.A.C.; *"SAVAD – Sistema de Avaliação de Desempenho de Modelos de Redes de Filas"*, Dissertação de Mestrado, CCT, Universidade Federal da Paraíba, Campina Grande, novembro de 1993.

[Sun, 02] Sun Microsystems Inc; *"Java Beans Specification"*, 1999. Disponível em http://www.sun.com/beans.

[Szyperski, 99] Szyperski, Clemens; *"Component Software Beyond Object-Oriented Programming",* Addison-Wesley, 1999.

[Takus 97] Takus, David A; *"Arena Software Tutorial"*, Pensylvania, System Modeling Corporation, 1997.

[Wagner, 00] Wagner, Marcos V. S.; *"Especificação de Componentes para Simulação de Redes TCP/IP"*, Dissertação de Mestrado, UFPB, 2000.

[Zortech, 90] Zortech Incorporated.; *"C++ Compiler Version 2.1"*, Massachusetts, 1990.

[Rocha, 02] Rocha, Flávio Gonçalves e Cabral, Maria Izabel Cavalcanti; "*Implementação e Validação de Componentes para a Construção de Ambientes de Simulação de Redes TCP/IP*". RT DSC/CCT/UFCG 005/2002.