

Marco Antonio Barros Alves Garcia  
Paulo Henrique Maestrello Assad Oliveira

**Motor de Simulação e Geração de Métricas para Avaliação  
de Desempenho para o Simulador de Grades  
Computacionais iSPD**

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista “Júlio de Mesquita Filho”, como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

São José do Rio Preto  
2010

Marco Antonio Barros Alves Garcia  
Paulo Henrique Maestrello Assad Oliveira

**Motor de Simulação e Geração de Métricas para Avaliação  
de Desempenho para o Simulador de Grades  
Computacionais iSPD**

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista “Júlio de Mesquita Filho”, como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

Orientador:  
Prof. Dr. Aleardo Manacero Júnior

Co-orientadora:  
Profa. Dra. Renata Spolon Lobato

São José do Rio Preto  
2010

Marco Antonio Barros Alves Garcia  
Paulo Henrique Maestrello Assad Oliveira

**Motor de Simulação e Geração de Métricas para Avaliação  
de Desempenho para o Simulador de Grades  
Computacionais iSPD**

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista “Júlio de Mesquita Filho”, como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

Prof. Dr Aleardo Manacero Jr      Marco A. B. A.Garcia      Paulo H. M. A. Oliveira

Banca Examinadora:

Profa. Dra. Adriana Barbosa Santos

Prof. Dr. José Márcio Machado

São José do Rio Preto  
2010

## **Dedicatória**

Marco:

Aos meus pais, Marco e Letícia.

Ao meu irmão Eduardo.

Paulo:

Ao meu avô, José Ribamar.

Aos meus pais, Paulo e Sônia.

Às minhas irmãs, Tatiana e Thaís.

À Juliana.

## **AGRADECIMENTOS**

Agradeço, primeiramente, a Deus, que me deu força e coragem para vencer todos os desafios enfrentados durante o curso.

Aos meus pais, que com sabedoria e paciência, me educaram e me ensinaram a ser o que hoje sou.

Ao meu irmão, que sempre me estimulou e motivou a superar os momentos difíceis.

A todos os tios, tias, primos e primas que com seus conselhos e apoio sempre me ajudaram a seguir em frente.

Aos meus avós, paternos e maternos, que mesmo distante dos olhos, sempre estiveram presentes no meu coração, como exemplo de força e superação.

Aos meus amigos, que próximos ou distantes, participaram de momentos importantes, somando as alegrias e conquistas, dividindo as dificuldades e a saudade daqueles que estavam longe.

Ao Prof. Dr. Aleardo Manacero Júnior, meu orientador, e a Prof. Dr. Renata Spolon Lobato, minha co-orientadora, pela convivência e experiência, como profissionais e amigos, e também aos demais professores, a quem deixo o meu sincero agradecimento.

À FAPESP, que me auxiliou financeiramente com uma bolsa de iniciação científica (processo nº 2009/00183-2) para o desenvolvimento deste projeto e também concedeu uma bolsa de auxílio pesquisa (processo nº 2008/09312-7) para a compra dos computadores do Grupo de Sistemas Paralelos e Distribuídos.

E a todos aqueles que, de alguma forma, contribuíram para que eu chegasse até aqui. Muito Obrigado!

Marco Antonio Barros. A. Garcia.

## AGRADECIMENTOS

| Em primeiro lugar, gostaria de agradecer a Deus e à vida por terem me proporcionado grandes momentos de aprendizado.

Agradeço aos meus pais, Paulo e Sônia por eles nunca terem medido esforços para criar as melhores condições possíveis para eu aprender tudo o que sei. Eles são imprescindíveis em minha vida.

Agradeço aos meus avôs e principalmente ao meu avô José Ribamar por ter sido meu estimulador intelectual desde a época em que eu nem sabia o que era isso.

Agradeço também às minhas irmãs, Tatiana e Thaís, por toda dedicação e apoio.

À minha namorada Juliana e sua família por me incentivarem e, por que não dizer, terem me adotado desde já como membro da família.

| Agradeço também aos professores que passaram pela minha vida e que direcionaram, com maestria, meus esforços intelectuais. Em especial, gostaria de agradecer aos meus orientadores Prof. Dr. Alcardo Manacero Júnior e Profa. Dra. Renata Spolon Lobato, pela orientação, paciência, dedicação e profissionalismo de suas partes.

Aos meus grandes amigos, André, Cauê, Lucas, Paulo, Vinícius, Aldo, Marco e Tiago, verdadeiros doutores em ouvir meus numerosos planos e me aconselhar durante este caminho.

À FAPESP, que me auxiliou financeiramente com uma bolsa de iniciação científica (processo nº 2009/00182-6) para o desenvolvimento deste projeto e também concedeu uma bolsa de auxílio pesquisa (processo nº 2008/09312-7) para a compra dos computadores do Grupo de Sistemas Paralelos e Distribuídos.

Aos integrantes do Grupo de Sistemas Paralelos e Distribuídos, que me auxiliaram no desenvolvimento deste projeto.

Paulo Henrique M. A. Oliveira.

## **Epígrafe**

*“Não é questão de ver quem é mais forte  
ou quem tem mais pontuação acima da nota de corte.  
É questão de objetivo na busca,  
para ser iluminado, porque brilho  
se ofusca”  
Marcus Vinicius Silva (Kamau)*

## **RESUMO**

O uso de sistemas computacionais de alto desempenho tem crescido nos últimos anos, em aplicações científicas e comerciais, com usuários nem sempre especialistas em programação de alto desempenho. Esse uso gera grande demanda de ferramentas para desenvolvimento e para a avaliação de desempenho dos sistemas em uso. Em particular, a avaliação de desempenho pode ser efetuada através do uso de simuladores. Entretanto, as ferramentas de simulação disponíveis nem sempre apresentam facilidade de uso e de geração de modelos, especialmente para o usuário não especialista em computação. Assim, a simulação de grades computacionais carece de simuladores que permitam a variabilidade das características simuladas e interfaces amigáveis. Visando preencher essa lacuna, esse projeto tem como objetivo a especificação e a implementação do motor de simulação de uma plataforma de simulação de grades computacionais, o iSPD, que permita ao usuário descrever com facilidade as características do sistema em estudo e as métricas de desempenho a ele adequadas. A implementação consistiu na representação computacional de um sistema de redes de filas no qual os servidores representam os nós da grade computacional e os elementos pertencentes às filas são as tarefas a serem processadas nesses nós. Além disso, fez-se a implementação das bibliotecas necessárias para os cálculos das métricas de desempenho dos sistemas simulados. Como resultado, um motor de simulação robusto e modular foi obtido.



## **ABSTRACT**

The use of high-performance computing systems has grown in recent years, either in scientific or commercial applications, with users not always experts in high performance programming. This use creates a great demand for development tools and performance evaluation systems. In particular, the performance evaluation can be performed through the use of simulators. However the simulation tools available are not always easy to use or to generate models, especially for non-specialist users. Thus, the simulation of computational grids requires simulators that allow for the variability of the simulated environment and provide user-friendly interfaces. Seeking to fill this gap, this project aims at the specification and implementation of the simulation engine of a simulation platform for grid computing, the iSPD, which allows the user to easily describe the characteristics of the system under study and its appropriate performance metrics. The implementation consisted of the computational representation of a queuing network system in which the servers represent the computational grid nodes and the queue elements are the tasks to be processed on these nodes. The necessary libraries for the calculations of performance metrics of simulated systems were also implemented. As a result, a robust and modular simulation engine was obtained.

# ÍNDICE

<b>LISTA DE FIGURAS .....</b>	<b>III</b>
<b>LISTA DE TABELAS.....</b>	<b>V</b>
<b>LISTA DE ABREVIATURAS E SIGLAS .....</b>	<b>VI</b>
<b>1 INTRODUÇÃO .....</b>	<b>1</b>
1.1 OBJETIVOS.....	2
1.2 ORGANIZAÇÃO DA MONOGRAFIA.....	3
<b>2 FUNDAMENTAÇÃO TEÓRICA .....</b>	<b>4</b>
2.1 <i>CLUSTERS</i> E GRADES COMPUTACIONAIS – COMPUTAÇÃO DE ALTO DESEMPENHO .....	4
2.2 SIMULADORES DE GRADES COMPUTACIONAIS .....	5
2.2.1 <i>SIMGRID</i> .....	6
2.2.2 <i>BRICKS</i> .....	8
2.2.3 <i>OPTORSIM</i> .....	9
2.2.4 <i>GANGSIM</i> .....	10
2.2.5 <i>GRIDSIM</i> .....	10
2.3 TÉCNICAS DE MODELAGEM DE SISTEMAS .....	11
2.3.1 <i>SIMULAÇÃO DE EVENTOS DISCRETOS</i> .....	11
2.4 DISTRIBUIÇÕES DE PROBABILIDADE.....	14
2.4.1 <i>TÉCNICA DA TRANSFORMAÇÃO INVERSA</i> .....	15
2.4.2 <i>TÉCNICA DA SIMULAÇÃO DIRETA</i> .....	16
2.5 TEORIA DAS FILAS.....	16
2.5.1 <i>NOTAÇÃO DE KENDALL</i> .....	17
2.5.2 <i>A FILA M/M/1</i> .....	18
2.5.3 <i>FILAS MULTI-SERVIDOR</i> .....	19
2.5.4 <i>MÚLTIPLAS FILAS COM SERVIDORES ÚNICOS</i> .....	19
2.6 MÉTRICAS DE SAÍDA .....	20
2.6.1 <i>TEMPO TOTAL SIMULADO</i> .....	20
2.6.2 <i>OCIOSIDADE</i> .....	21
2.6.3 <i>EFICIÊNCIA</i> .....	21
2.6.4 <i>SATISFAÇÃO</i> .....	22
2.6.5 <i>MÉTRICAS OBTIDAS DAS TAREFAS</i> .....	23
2.7 CONSIDERAÇÕES FINAIS.....	24
<b>3 DESENVOLVIMENTO DO MOTOR DE SIMULAÇÃO .....</b>	<b>25</b>
3.1 O INTERPRETADOR DE MODELOS E O MODELO SIMULÁVEL .....	26
3.2 OS EVENTOS DO MOTOR DE SIMULAÇÃO .....	27
3.3 O PACOTE SIMULAÇÃO.....	29
3.3.1 <i>A CLASSE SIMULACAO.JAVA</i> .....	29
3.3.2 <i>AS CLASSES LISTAEVENTOSFUTUROS.JAVA, NOLEF.JAVA E RELOGIO.JAVA</i> .....	34

Excluído: ACEITAÇÃO-REJEIÇÃO

Excluído:

3.4 O PACOTE REDES DE FILAS.....	35
3.4.1 A CLASSE REDESDEFILAS.JAVA .....	35
3.4.2 A CLASSE CENTROSDESERVICO.JAVA .....	37
3.4.3 A CLASSE SERVIDORES.JAVA .....	38
3.4.4 AS CLASSES NOFILA.JAVA E FILAS.JAVA .....	39
3.5 O PACOTE NÚMEROS ALEATÓRIOS .....	40
3.5.1 A CLASSE GERACAO NUM ALEATORIOS.JAVA .....	40
3.6 O PACOTE ESTATÍSTICA .....	41
3.6.1 AS CLASSES ESTATISTICA.JAVA E CAIXATEXTOESTATISTICA.JAVA.....	41
3.6.2 AS CLASSES METRICA CS.JAVA, MEDIDAS SERVIDOR.JAVA E NOMEDIDAS SERVIDOR.JAVA .....	43
3.6.3 AS CLASSES TAREFAS FILA.JAVA E NOTAREFAS FILA.JAVA .....	44
3.7 CONSIDERAÇÕES FINAIS .....	45
<b>4 TESTES</b>	<b>46</b>
4.1 TESTES DOS GERADORES DE NÚMEROS ALEATÓRIOS.....	46
4.2 TESTES DO SISTEMA M/M/1 .....	48
4.3 TESTE FINAL.....	49
4.4 CONSIDERAÇÕES FINAIS.....	53
<b>5 CONCLUSÕES</b>	<b>54</b>
5.1 CONTRIBUIÇÕES.....	54
5.2 DIFICULDADES ENCONTRADAS .....	55
5.3 CONCLUSÕES .....	55
5.4 PROPOSTAS PARA TRABALHOS FUTUROS .....	56
<b>REFERÊNCIAS BIBLIOGRÁFICAS</b>	<b>57</b>
<b>APÊNDICE A - DIAGRAMAS DE CLASSES UML DOS PACOTES DO PROJETO</b>	<b>61</b>

## LISTA DE FIGURAS

FIGURA 1.1: DIAGRAMA CONCEITUAL DA PLATAFORMA DE SIMULAÇÃO. ....	2
FIGURA 2.1: COMPONENTES DO SIMGRID .....	6
FIGURA 2.2: <b>M</b> ODELAGEM DO AMBIENTE COMPUTACIONAL POR REDES DE FILAS.....	8
FIGURA 2.3: <b>Q</b> UANTIDADE DE ALGORITMO <i>EVENT SCHEDULING/TIME ADVANCE</i> .....	14
FIGURA 2.4: A REPRESENTAÇÃO GRÁFICA DE UMA FILA M/M/1. ....	19
FIGURA 2.5: <b>A</b> REPRESENTAÇÃO GRÁFICA DE UMA FILA COM VÁRIOS SERVIDORES .....	19
FIGURA 2.6: <b>A</b> REPRESENTAÇÃO GRÁFICA DE VÁRIAS FILAS E VÁRIOS SERVIDORES .....	20
FIGURA 3.1: A INTERFACE GRÁFICA DO SIMULADOR ISPD .....	26
FIGURA 3.2: OS POSSÍVEIS EVENTOS DE UMA GRADE COMPUTACIONAL .....	28
FIGURA 3.3: A MODELAGEM COM REDE DE PETRI DOS EVENTOS DA SIMULAÇÃO. ....	31
FIGURA 3.4: <b>A</b> IMPLEMENTAÇÃO DA CHAMADA DO EVENTO “ENVIO DE TAREFA” .....	33
FIGURA 3.5: O MÉTODO <i>COMPARETO</i> DA CLASSE <i>NOLEF.JAVA</i> . ....	34
FIGURA 3.6: O MÉTODO <i>REMOVEELEMENTOSFILA</i> DA CLASSE CENTROSDESERVICO.JAVA.....	37
FIGURA 3.7: O MÉTODO <i>ATRIBUITAREFASERVIDOR</i> DA CLASSE <i>SERVIDORES.JAVA</i> .....	39
FIGURA 3.8: GERAÇÃO DE NÚMEROS PERTENCENTES À DISTRIBUIÇÃO TWO-STAGE UNIFORM.....	41
FIGURA 3.9: O MÉTODO <i>FIMSIMULACAO</i> DA CLASSE <i>ESTATISTICA.JAVA</i> .....	43
FIGURA 3.10: MÉTODO DA CLASSE <i>METRICACS.JAVA</i> QUE ADICIONA DADOS DOS SERVIDORES AO CENTRO DE SERVIÇO .....	44
FIGURA 3.11: TRECHO DE CÓDIGO RETIRADO DO MOTOR DE SIMULAÇÃO MOSTRANDO A CHAMADA DO MÉTODO <i>ADDNOTAREFASFILA</i> .....	45
FIGURA 4.1: HISTOGRAMA DA DISTRIBUIÇÃO <i>TWO-STAGE UNIFORM</i> .....	47
FIGURA 4.2: <b>H</b> ISTOGRAMA DA DISTRIBUIÇÃO NORMAL.....	47
FIGURA 4.3: <b>H</b> ISTOGRAMA DA DISTRIBUIÇÃO EXPONENCIAL.....	48
FIGURA 4.4: PLATAFORMA CRIADA PARA O TESTE FINAL DO ISPD.....	50
FIGURA 4.5: RESULTADO DO TESTE COM 2 TAREFAS.....	51
FIGURA 4.6: RESULTADO DO TESTE COM 300 TAREFAS.....	52
FIGURA A.1: DIAGRAMA DE CLASSES UML DO PACOTE <i>REDESDEFILA.JAVA</i> .....	61
FIGURA A.2: DIAGRAMA DE CLASSES UML DOS PACOTES <i>SIMULAÇÃO.JAVA</i> E <i>GERACAO NUM ALEATORIOS.JAVA</i> .....	62
FIGURA A.3: DIAGRAMA DE CLASSES UML DAS CLASSES <i>LISTAEVENTOSFUTUROS.JAVA</i> E <i>NOLEF.JAVA</i> .....	63
FIGURA A.4: DIAGRAMA DE CLASSES UML DETALHADO DA CLASSE <i>REDESDEFILAS.JAVA</i> .....	64

Excluído: M

Excluído: O

Excluído: S

Excluído: A

Excluído: A

Excluído: A

Excluído: H

Excluído: H

FIGURA A.5: DIAGRAMA DE CLASSES UML DETALHADO DA CLASSE CENTROSDeSERVICO .....	65
FIGURA A.6: DIAGRAMA DE CLASSES UML DETALHADO DA CLASSE SERVIDORES.JAVA.....	66
FIGURA A.7: DIAGRAMA DE CLASSES UML DETALHADO DA CLASSE NOFILA.JAVA.....	67
FIGURA A.8: DIAGRAMA DE CLASSES UML DETALHADO DAS CLASSES CAIXATEXTO.JAVA E ESTATISTICA.JAVA.....	68
FIGURA A.9: DIAGRAMA DE CLASSES UML DETALHADO DAS CLASSES ESTATISTICA.JAVA, METRICA.CS.JAVA, MEDIDASSERVIDOR.JAVA, NOMEDIDASSERVIDOR.JAVA, TAREFAFILA.JAVA E NOTAREFAFILA.JAVA.....	69

## LISTA DE TABELAS

TABELA 3.1: OS POSSÍVEIS EVENTOS DA SIMULAÇÃO. ....	32
TABELA 4.1: OS PARÂMETROS USADOS PARA GERAR AS DISTRIBUIÇÕES DE PROBABILIDADE.....	47
TABELA 4.2: AS CARACTERÍSTICAS E OS RESULTADOS DO SISTEMA M/M/1 .....	49

## LISTA DE ABREVIATURAS E SIGLAS

AMOK: *Advanced Metacomputing Overlay Kit*  
API: *Application Programming Interface*  
CS: Centro de Serviço  
CPU: Central Processing Unit  
DAG: *Direct Acyclic Grafs*  
FCFS: First Come, First Served  
FIFO: *First In, First Out*  
FE: *Front-End*  
GSPD: Grupo de Sistemas Paralelos e Distribuídos  
GRAS: *Grid Reality And Simulation*  
GUI: *Graphical User Interface*  
ID: Identificador  
IP: *Internet Protocol*  
JDK: *Java Development Kit*  
JVM: *Java Virtual Machine*  
LIFO: *Last in, First out*  
LEF: Lista de Eventos Futuros  
MSG: Meta-SimGrid  
RR: *Round-Robin*  
SG: SimGrid  
SMPI: *SimGrid Message Passing Interface*  
TTL: *Time to Live*  
UML: *Unified Modeling Language*  
VO: *Virtual Organizations*  
XBT: *Extended Bundle of Tools*  
XML: *eXtensible Markup Language*

# CAPÍTULO 1

## Introdução

O uso de sistemas computacionais de alto desempenho tem crescido intensamente nos últimos anos. Isso se dá tanto em aplicações científicas (genoma, bioinformática, física de partículas, química, engenharia, entre outras) como em aplicações comerciais (por exemplo, animação, comércio eletrônico, *business intelligence*), criando um espectro de usuários que, em sua maioria, não podem ser considerados especialistas em programação de alto desempenho. Essa característica cria duas demandas: existência de ferramentas para desenvolvimento de sistemas e para avaliação de desempenho dos sistemas em uso. Essa última decorre da necessidade em se reduzir os custos envolvidos na utilização de sistemas que já são inerentemente caros.

A partir daí, é preciso considerar que a avaliação de desempenho pode ser realizada por três diferentes técnicas: modelagem analítica, simulação e *benchmarking* [Jain et al., (1991)]. A modelagem analítica apresenta limitações quanto à precisão e a confiabilidade do modelo. Por sua vez, *benchmarking* apresenta limitações quanto a sua aplicabilidade (necessidade do sistema físico real) e custos associados. A técnica de simulação apresenta em menor grau os problemas da modelagem analítica sem a presença dos problemas de *benchmarking*, sendo uma técnica bastante usada, mesmo considerando-se seus problemas de precisão. O maior problema com a simulação é que as ferramentas construídas para a sua aplicação são,



em geral, limitadas no que diz respeito à facilidade de uso e de geração de modelos.

Dada a importância do tema e dos problemas listados nesta introdução, dentro deste projeto é feita a proposta de especificação e implementação do motor de uma plataforma de simulação de grades computacionais. Este motor deve ser capaz de prover medidas de desempenho e permitir mudanças nos modelos simulados em tempo de execução.

É necessário citar que este projeto é parte integrante de um projeto maior, o iSPD, e que a descrição das outras partes dele podem ser encontradas em [Aqui e Guerra, (2010)]. Este texto trata somente da especificação do motor de simulação e de suas métricas para avaliação de desempenho. A Figura 1.1 traz o conceito do simulador.

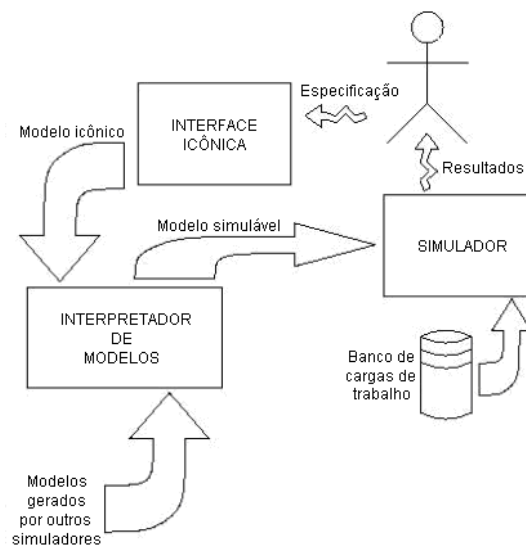


Figura 1.1: Diagrama conceitual da plataforma de simulação.

## 1.1 Objetivos

O objetivo deste trabalho é especificar e implementar o motor de uma ferramenta de simulação de grades computacionais. Assim, podem ser listados os seguintes pontos fundamentais:

1. Identificação de funcionalidades características de grades computacionais

relevantes para a sua simulação;

2. Formulação de um modelo de simulação suficientemente flexível para considerar as características dinâmicas e distribuídas de uma grade;

3. Formulação de um ambiente de simulação capaz de implementar análises do desempenho de aplicações.

## 1.2 Organização da Monografia

No capítulo dois, apresenta-se uma revisão bibliográfica dos assuntos relacionados com o projeto, incluindo um estudo detalhado dos simuladores estado da arte em simulação de *clusters* e *grids*, técnicas de simulação, teoria das filas e um estudo das distribuições de probabilidade estudadas para o desenvolvimento do simulador;

Em sequência, no capítulo três, são apresentados detalhes de implementação do projeto. Alguns resultados importantes, como os modelos de filas utilizados, as distribuições de probabilidade geradas, os parâmetros de entrada e as métricas de saída, também estão descritos;

No capítulo quatro apresentam-se os testes realizados com os geradores de números aleatórios, com o modelo de filas M/M/1, e com o motor do simulador propriamente dito, assim como os resultados obtidos;

No capítulo cinco são oferecidas contribuições, dificuldades encontradas, conclusões e direcionamento para trabalhos futuros.

Finalmente, no apêndice A são apresentados os diagramas de classe UML dos pacotes desenvolvidos ao longo do projeto.

Excluído: ,

## CAPÍTULO 2

### Fundamentação Teórica

Neste capítulo é feita a fundamentação teórica acerca do projeto. Entre os temas abordados estão computação distribuída fracamente acoplada, simuladores de grades computacionais, simulação de eventos discretos, distribuições de probabilidade, teoria das filas e métricas de avaliação de desempenho. A partir dos estudos dos simuladores, foi possível abstrair os requisitos necessários para efetuar uma simulação de grades computacionais. Compreender a simulação de eventos discretos foi imprescindível para entender o funcionamento do motor de simulação. Através das teorias da probabilidade e das filas, foi possível determinar os modelos matemáticos essenciais do projeto. Finalmente, pelo estudo das métricas de avaliação de desempenho, determinou-se as métricas de saída do iSPD.

Excluído: ão

Excluído: probabilidade

Excluído: e

Excluído: Finalmente, a

Excluído: ,

#### 2.1 Clusters e Grades Computacionais – Computação de Alto Desempenho

Sistemas de computação distribuídos, tais como *clusters* e grades, são uma das formas de se fazer computação de alto desempenho. São considerados fracamente acoplados, pois a comunicação entre os nós é feita por redes de computadores.

Enquanto que, nas arquiteturas fortemente acopladas a comunicação é feita pelos barramentos internos.

Um aspecto característico da computação em *cluster* é sua homogeneidade. Na maioria dos casos, os computadores que compõem um *cluster* possuem, em grande parte, as mesmas configurações, todos têm o mesmo sistema operacional e todos estão conectados à mesma rede. Por comparação, sistemas de computação em grade têm alto grau de heterogeneidade, nenhuma premissa é adotada em relação a *hardware*, sistemas operacionais, redes, domínios administrativos, políticas de segurança e assim por diante [Tanenbaum et al.,(2007)].

Formatado: Fonte: Itálico

O desenvolvimento de sistemas computacionais de alto desempenho é estimulado pela necessidade, cada vez mais frequente, de se resolver problemas complexos, grandes ou com um alto volume de dados em tempo hábil.

Mesmo sendo menos dispendiosos financeiramente que as outras formas de se fazer computação de alto desempenho, os sistemas computacionais distribuídos ainda demandam um investimento considerável e a avaliação de desempenho desses sistemas é uma das medidas interessantes para analisar a viabilidade desse investimento.

## 2.2 Simuladores de Grades Computacionais

A simulação de grades computacionais é pertinente, pois a implementação de plataformas reais para estudos é dispendiosa. Aspectos da implementação de uma grade computacional como, por exemplo, seu poder computacional, podem ser estudados antecipadamente com o uso de simuladores.

Nesta seção, o estudo de simuladores de grades computacionais realizado e sua consequente contribuição para o simulador proposto são relatados. São apresentados a seguir os simuladores *SimGrid* [SimGrid Project, (1999)], *Bricks* [Bricks System, (1999)], *Optorsim* [Bell et al., (2003)], *Gangsim* [Dumitrescu et al, (2005)] e *Gridsim* [Buyya et al., (2002)].

### 2.2.1 SimGrid

Excluído: ¶

O *Simgrid* é um simulador orientado a eventos e foi concebido para estudar algoritmos de escalonamento centralizados para aplicações científicas paralelas em plataformas computacionais distribuídas e heterogêneas. É *opensource* e está disponível para os ambientes *Windows*, *Linux* e *MacOS*.

A ferramenta implementa uma API para a linguagem C, denominada SG. Com tal API, é possível especificar a simulação do escalonamento de tarefas em determinados recursos.

Os arquivos XML que modelam os recursos e as tarefas são passados ao *Simgrid* como parâmetros na inicialização e os resultados das simulações são vistos através de mensagens pré-formatadas [Oliveira, (2008)].

Sua arquitetura em camadas é ilustrada na Figura 2.1.

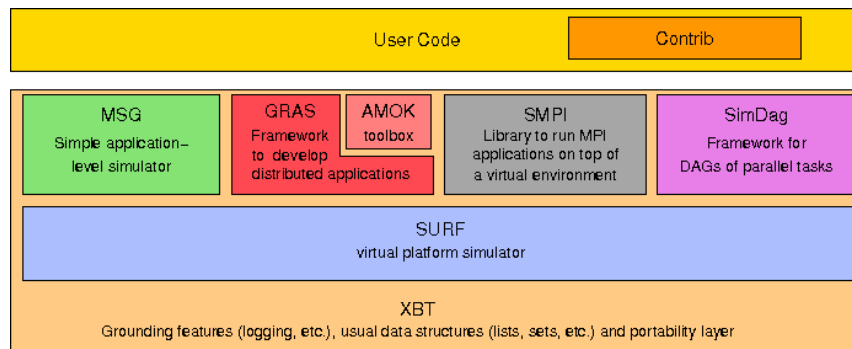


Figura 2.1: Componentes do *Simgrid* [SimGrid Project, (1999)].

#### Módulos do *SimGrid*

##### MSG

O MSG [MSG, (1999)] foi o primeiro ambiente de programação distribuído desenvolvido no *SimGrid* e visa a simulação em termos dos agentes de comunicação, portanto o realismo da simulação não é o principal objetivo deste módulo, sendo que muitos detalhes técnicos da plataforma da grade são omitidos.

Excluído: .

No MSG, um agente é responsável por tomar as decisões de escalonamento e executar uma tarefa (computação ou transferência de dados). Os seguintes passos fazem parte da simulação em MSG: modelagem do problema, modelagem da plataforma física e alocação dos agentes em suas localidades.

## GRAS

Já o GRAS (*Grid Reality and Simulation*) [GRAS, (1999)] é um *framework* para desenvolvimento de aplicações distribuídas orientadas a eventos. Ele possibilita a execução da aplicação juntamente com o simulador e em uma plataforma distribuída real (utilizando duas versões distintas de sua API). Suas vantagens incluem a facilidade de uso e de controle do simulador e a produção automática de código para uma plataforma real.

Juntamente como o GRAS, existe um *toolkit* chamado AMOK (*Advanced Metacomputing Overlay Kit*) que implementa, em alto nível, diversos serviços necessários para várias aplicações distribuídas.

Os outros ambientes são: o SMPI [SMPI, (1999)], *framework* de execução de aplicações MPI não alteradas; o *Simdag* [SimDag, (1999)] que faz simulações de aplicações paralelas por meio do modelo DAG que, por sua vez, possibilita a especificação de relações de dependência entre as tarefas de um programa paralelo; já o *SURF* [SURF, (1999)] provê um conjunto de funcionalidades para simular uma plataforma virtual. Além desses, há o XBT representante do núcleo de ferramentas *SimGrid* e onde há diversas funcionalidades que auxiliam o *SURF* como, por exemplo, estruturas de dados, serviço de *logging* e manipulação de grafos.

Os recursos computacionais que executam tarefas são modelados pelo seu poder computacional, enquanto que os *links* de comunicação são modelados pela sua largura de banda e latência. Por sua vez, as tarefas são modeladas pela sua carga e por sua quantidade.

Formatado: Fonte: Itálico

### 2.2.2 Bricks

Essa ferramenta faz avaliações de desempenho que permitem a análise e a comparação de diferentes estratégias de escalonamento de tarefas em sistemas computacionais de alto desempenho. O simulador *Bricks* é implementado em *Java*, sendo funcional em qualquer sistema operacional com suporte a JVM.

Ele é composto de dois módulos. O ambiente computacional global contém o escalonador de tarefa e o módulo responsável pela monitoração dos recursos. A unidade de escalonamento representa a estrutura física da grade (clientes, servidores e rede).

O *Bricks* utiliza o sistema FCFS, proporcionando maior realidade às simulações, já que sua carga de trabalho pode ser especificada por meio de parâmetros reais. As modelagens, tanto da arquitetura quanto da aplicação, são feitas através de linguagem de *script* própria do *Bricks*.

Um aspecto interessante deste simulador é o emprego de redes de filas como motor da simulação. A Figura 2.2 mostra como o *Bricks* aplica esse conceito. A fila  $Q_{ns}$  representa a rede do cliente ao servidor, a fila  $Q_{nr}$  funciona como a rede do servidor ao cliente, e os servidores propriamente ditos são representados por  $Q_s$ . As taxas de serviços de  $Q_{nr}$  e  $Q_s$  representam, respectivamente, a largura de banda de cada uma das redes e o poder de processamento dos servidores (A e A' denotam um mesmo cliente, mas foram diferenciados para melhor entendimento).

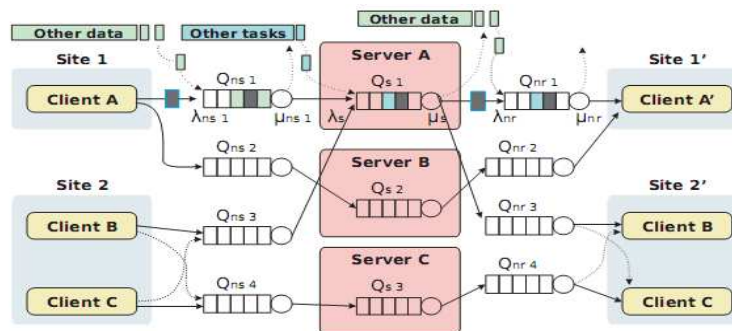


Figura 2.2: Modelagem do ambiente computacional por redes de filas [Bricks System, (1999)].

A rede entre os clientes e servidores é parametrizada por sua largura de banda, congestionamento e sua variação ao longo do tempo. Já os recursos computacionais do sistema distribuído são parametrizados por seu poder computacional, carga de trabalho e variação temporal. E as tarefas são modeladas pela quantidade de dados transmitidos de/para um servidor em função da transmissão da tarefa pela rede e pelo número de operações executadas pela tarefa.

### 2.2.3 *Optorsim*

Com o *Optorsim* é possível realizar o estudo de estratégias de escalonamento baseadas em replicação de dados. Também é implementado em *Java* e possui uma GUI na qual o usuário pode configurar os parâmetros da simulação e analisar seus resultados.

Para tornar as simulações ainda mais reais, adotou-se a arquitetura de grade de dados desenvolvida pelo projeto *EU Data Grid* [EU DataGrid Project, (2001)].

O *Computing Element* é o componente que executa as tarefas propriamente ditas. O *Storage Element* armazena, em arquivos, os dados utilizados nas simulações. O *Replica Optimizer* decide se as réplicas serão eliminadas ou geradas e o *Resource Broker* gerencia o *Computing Element*.

No *Optorsim*, os detalhes da simulação são mostrados em tempo real, e ao final da simulação, estatísticas e informações sobre a grade são mostradas.

Os parâmetros utilizados na modelagem dos recursos computacionais são: número de elementos computacionais do *site*, número de elementos de armazenamento do *site* e tamanho dos elementos de armazenamento em megabytes. Para modelar a rede é usada uma matriz diagonal simétrica *site* versus *site*, em que cada elemento indica a máxima largura de banda do canal que liga os sites correspondentes.

|



#### 2.2.4 Gangsim

O *Gangsim* é um simulador discreto orientado a eventos. É *opensource*, implementado em *Perl* e tem uma interface *Web*. Seu uso é indicado quando se deseja avaliar o impacto das políticas de alocação adotadas por *sites* e organizações virtuais ou quando é necessário simular cargas de trabalho síncronas e assíncronas.

Os componentes dessa ferramenta são os *sites* – constituídos pelos conjuntos de recursos computacionais e de armazenamento, pelos escalonadores e pelo componente para assegurar que a política de alocação de recursos será cumprida – e as VOs que são constituídas pelo conjunto de usuários e pelos componentes de monitoração.

Os elementos de uma grade real simulados no *Gangsim* são a infra-estrutura de submissão de tarefas, a infra-estrutura de monitoração e a política de uso dessa infra-estrutura.

O *Gangsim* implementa diversos algoritmos e estratégias utilizadas para atualizar o estado de diferentes componentes do *framework*. A modelagem é baseada nas políticas de alocação de recursos dos diferentes *sites* e VOs.

Um site é parametrizado pelo número de CPUs, poder computacional de cada CPU, número de discos, tamanho de espaço em disco, canais de comunicação e a largura de banda destes canais. As características da rede instituída entre os sites, a política de escalonamento usada pelo site, a parcela de tempo de CPU, o espaço em disco e a largura de banda que cada VO pode usar são as informações fornecidas para a simulação.

Cada VO utiliza sua própria maneira de escalonamento e as tarefas são postas em conjuntos chamados “carga de trabalho”.

#### 2.2.5 Gridsim

O *Gridsim* [GridSim, (2007)] foi proposto para facilitar a simulação de recursos heterogêneos, usuários, escalonadores e aplicações. Uma aplicação é seu uso quando as estratégias de otimização de replicação nas grades exigem grande volume de dados. Também foi projetado para investigar interações e interferências

nas decisões de escalonamento tomadas pelo escalonador. Escrito em *Java*, é dividido em várias camadas. Essa ferramenta é *opensource*, possui uma extensa documentação e apresenta uma interface gráfica que permite a gravação de arquivos XML para consultas posteriores.

Utiliza o conceito de entidades para representar os usuários, o escalonador com perfil de diferentes usuários, os recursos da grade e o *Grid Information Service*. E, assim como o *Bricks*, faz uso do sistema FCFS.

A modelagem das tarefas, nesse simulador, é feita por um componente chamado *Gridlet*.

## 2.3 Técnicas de Modelagem de Sistemas

As técnicas de modelagem buscam resolver o modelo de um sistema através de solução analítica ou simulação [Chiacchio, (2005)].

- A técnica de solução analítica consiste em definir o problema através de definições lógicas e matemáticas passíveis de resolução. Essa técnica fornece resultado preciso, mas a dificuldade da solução é proporcional à complexidade do sistema e isso pode acarretar num aumento do custo e do tempo necessários para resolver o problema. Essa técnica é recomendada para situações nas quais o modelo matemático é conhecido.
- A técnica de simulação consiste em modelar o sistema para transformá-lo em um programa que represente com fidelidade o sistema real. É uma técnica prognóstica que examina o comportamento do sistema através de várias execuções do modelo. Por essa técnica ser recomendada no uso de sistemas em fase de projeto e de grande complexidade, uma de suas formas - simulação de eventos discretos - foi adotada no projeto e será vista a seguir.

### 2.3.1 Simulação de Eventos Discretos

Na simulação orientada a evento, um sistema é modelado pela definição das mudanças que ocorrem no tempo do evento. Q projetista do sistema deve, então,

Excluído: A

determinar os eventos que podem causar mudanças no estado do sistema para, em seguida, desenvolver a lógica associada com cada tipo de evento. A simulação do sistema é produzida pela execução da lógica associada a cada evento, em uma sequência ordenada pelo tempo [Soares, (1991)].

#### ▪ **Sistemas e Ambientes de Sistema**

Sistema é uma coleção de itens, entre os quais existe uma relação que é objeto de estudo ou de interesse.

Um sistema é frequentemente afetado por mudanças que ocorrem fora dele. Tais mudanças ocorrem no ambiente do sistema. Na modelagem de um sistema é necessário definir claramente os limites entre o sistema e seu ambiente.

Alguns termos precisam ser definidos para uma melhor compreensão sobre um sistema [Banks et al, (2001)]:

Estado do sistema: coleção de variáveis que contêm toda informação necessária para descrever o sistema a qualquer instante.

Entidade: qualquer objeto ou componente no sistema que requer uma representação explícita do modelo. Como por exemplo, servidor, cliente, máquina, etc.

Atributos: propriedade de uma dada entidade. Como por exemplo, prioridade de uso, trajetos definidos, etc.

Lista: coleção de entidades permanentemente ou temporariamente ordenadas segundo algum critério. Como por exemplo, FCFS ou por prioridades.

Evento: uma ocorrência instantânea capaz de mudar o estado do sistema. Como por exemplo, a chegada de um novo cliente.

Notificação de evento: registro de um evento que deve ocorrer no instante atual ou em algum instante futuro, juntamente com quaisquer dados necessários para a execução do evento. Esse registro deve incluir no mínimo o tipo de evento e o tempo do evento.

Lista de eventos futuros (LEF): é uma lista de notificações de eventos ordenada pelo tempo de ocorrência do evento.

Atraso: duração de um tempo cujo tamanho não é previamente conhecido, ou seja, só é especificado após seu término.

Relógio: variável para representar o tempo simulado, servindo para controle do tempo virtual do sistema. Esse tempo não representa o tempo real do sistema.

Sistemas discretos: são aqueles sistemas nos quais os estados das variáveis mudam apenas em um conjunto discreto de pontos ao longo do tempo.

Modelo: é uma descrição de um sistema por meio de equações e relações matemáticas bem como de representações gráficas, baseadas em leis ou princípios físicos que governam o sistema, ou seja, o modelo é uma abstração do sistema original.

Simulação: é o processo de elaboração de um modelo de um sistema real (ou hipotético) e a condução de experimentos com a finalidade de entender o comportamento de um sistema ou avaliar sua operação [Shannon, (1975)].

Em simulação discreta, somente os eventos onde há alterações no sistema são considerados, ou seja, o tempo decorrido entre essas alterações não é relevante para obtenção dos resultados da simulação, embora o tempo nunca pare. Apresenta-se a seguir o algoritmo *Event Scheduling/Time Advance*, que é o mecanismo para o controle do tempo de uma simulação.

- **O Algoritmo *Event Scheduling/Time Advance* [Banks et al., (2001)]**

É o mecanismo para avançar a simulação e garantir que todos os eventos ocorram em uma ordem cronológica correta. É baseado em uma LEF.

O gerenciamento da LEF é chamado de processamento da lista e a eficiência desse processamento é muito dependente do método de ordenação da lista. A remoção, a adição e a remoção por cancelamento são as operações mais realizadas nessa lista.

No contexto de simulações discretas, escalonar um evento futuro consiste em, na eminência desse evento começar, calcular sua duração através de uma distribuição de probabilidade e colocar em uma lista tanto o tempo de duração quanto o instante do seu término.

Esse algoritmo segue basicamente os passos descritos a seguir, ilustrados na Figura 2.3:

Passo 1: Remover o aviso do evento eminente da LEF (evento 3, tempo 1).

Passo 2: Avançar o relógio para o tempo do evento eminente (isto é, avançar o

relógio de  $t$  para  $t_1$ ).

Passo 3: Executar o evento eminente, atualizando o estado do sistema, mudando os atributos dos elementos e alterando o conjunto de membros (se for necessário).

Passo 4: Gerar eventos futuros (se necessário) e colocar o aviso de eventos na LEF ordenando pelo tempo (Exemplo: o evento 4 ocorre no tempo  $t^*$ , e  $t_2 < t^* < t_3$ ).

Passo 5: Atualizar as estatísticas e os contadores.

Clock	System state	Future event list
$t$	(5, 1, 6)	(3, $t_1$ ) – Type 3 event to occur at time $t_1$ (1, $t_2$ ) – Type 1 event to occur at time $t_2$ (1, $t_3$ ) – Type 1 event to occur at time $t_3$ ... (2, $t_n$ ) Type 2 event to occur at time $t_n$
$t_1$	(5, 1, 5)	(1, $t_2$ ) – Type 1 event to occur at time $t_2$ (4, $t^*$ ) – Type 4 event to occur at time $t^*$ (1, $t_3$ ) – Type 1 event to occur at time $t_3$ ... (2, $t_n$ ) Type 2 event to occur at time $t_n$

Figura 2.3: O algoritmo *Event Scheduling/Time Advance* [Banks et al., (2001)].

## 2.4 Distribuições de Probabilidade

Uma característica comum a grande parte dos sistemas contínuos ou discretos é que os eventos simulados são, na maioria dos casos, indeterminísticos, ou seja, não podem ser precisamente determinados a partir do estado atual do sistema. Comportamentos indeterminísticos seguem, em geral, padrões que obedecem a distribuições de probabilidade bem definidas. Assim, sua simulação correta precisa fazer uso de geradores de números aleatórios que correspondam adequadamente ao perfil de tais distribuições.

Portanto, um estudo acerca do assunto foi essencial para o desenvolvimento do projeto. Nesse ínterim, além de detectar variáveis aleatórias pertinentes ao projeto, é importante avaliar suas distribuições de probabilidade, isto é, as probabilidades dos diversos eventos envolvendo tais variáveis.

A seguir, os conceitos de variáveis aleatórias discretas e contínuas são descritos.

Variáveis aleatórias discretas: Seja  $X$  uma variável aleatória. Se o número de possíveis valores para  $X$  for finito ou infinito contável,  $X$  é chamada de variável aleatória discreta.

Variáveis aleatórias contínuas: Seja  $R_x$  a faixa de valores que uma variável aleatória  $X$  pode assumir. Se  $R_x$  for um intervalo ou um conjunto de intervalos,  $X$  é chamada de variável aleatória contínua.

Um modelo estocástico de simulação tem uma ou mais variáveis aleatórias como entrada. Estas entradas aleatórias levam às saídas aleatórias que podem somente ser consideradas como estimativas das características verdadeiras de um modelo. A seguir, serão vistas algumas das técnicas para geração de números aleatórios.

#### 2.4.1 Técnica da Transformação Inversa

Apesar desta técnica de geração de números aleatórios ser, a princípio, aplicável para qualquer distribuição, ela é mais indicada quando a distribuição acumulada é simples o suficiente para a sua função inversa,  $F^{-1}(x)$  ser encontrada facilmente.

O método é baseado no fato de que a distribuição acumulada,  $F(x)$ , tem valor entre 0 e 1, ou seja, no mesmo intervalo de um número aleatório,  $U$ , gerado por um gerador básico. Assim sendo, tendo  $U$ , considera-se esse valor com sendo um valor de  $F(x)$ . Para achar o valor de  $x$ , basta resolver a equação, achando a inversa de  $F(x)$ .

Logo, essa técnica é mais comumente utilizada para gerar amostras das distribuições exponencial, uniforme, Weibull, triangular e empíricas.

Por exemplo, para a geração da distribuição exponencial devem ser executados os seguintes passos para obter a transformação:

- 1) Toma-se  $F(x)$  para distribuição exponencial.  $F(x) = 1 - e^{-\lambda x}$ ;
- 2) Isola-se o  $x$  na lei de  $F(x)$  exposta acima. O resultado dessa operação é o chamado “gerador de variáveis aleatórias”. No caso em questão, ele é dado por:  $x = (-1 / \lambda) * \ln (1 - F(x))$ .
- 3) Basta agora substituir os valores para  $\lambda$  e  $F(x)$ . Onde  $F(x)$  é obtido através de um gerador de números aleatórios. Na linguagem *Java*,

temos a função “*Random*” da biblioteca “*Math*” para geração de números pseudo-aleatórios no intervalo [0, 1].

#### 2.4.2 Técnica da Simulação Direta

Algumas distribuições de probabilidade, como a distribuição normal, possuem funções de densidade probabilística cuja integração analítica é inviável e desta forma não se pode usar o método da transformação inversa para gerá-las. Nestes casos, é possível utilizar o método da simulação direta (Santos, (1999)). Para aplicar esse método à distribuição normal, considera-se o teorema do limite central, pois segundo ele, a soma de  $N$  variáveis aleatórias uniformemente distribuídas entre [0,1] segue uma distribuição normal com  $\mu = N/2$  e  $\sigma = \sqrt{N/12}$ , o que resulta na equação 2.1:

$$Z = \frac{\sum_{i=1}^N U_i - \frac{N}{2}}{\sqrt{\frac{N}{12}}} \quad (2.1)$$

Como a equação 2.1 é válida para  $N > 10$ , fazemos  $N = 12$  para facilitar o procedimento computacional e obtemos  $Z = \sum_{i=1}^{12} U_i - 6$ .

Logo, tem-se um procedimento simples para gerar uma variável aleatória normalmente padronizada. Simplesmente soma-se 12 números aleatórios uniformemente distribuídos em [0,1] e então se subtrai 6, obtendo-se um valor para  $Z$ . Se o objetivo é gerar uma variável normal ( $X$ ) com média  $\mu$  e desvio padrão  $\sigma$ , gera-se primeiro o número  $Z$  e depois gera-se  $X$  com  $X = \mu + \sigma Z$ .

### 2.5 Teoria das Filas

A formação de filas é um fenômeno rotineiro que ocorre sempre que a demanda atual por serviço excede à capacidade atual de fornecimento daquele serviço [Hiller et al., (1988)].

Uma rede de filas é composta por entidades chamadas centros de serviço e um

conjunto de entidades chamadas usuários (ou clientes) que recebem serviços nestes centros. Um centro de serviço é constituído de um ou mais servidores que prestam serviços aos usuários e por uma área de espera (denominada fila) para usuários que esperam pelo serviço requisitado [Santana et al., (1994)].

Para uma melhor compreensão sobre teoria das filas, alguns conceitos serão elencados:

Tempo entre chegadas: define a frequência regular com que os clientes chegam ao sistema [Soares, (1990)].

Tempo de serviço: define o tempo necessário para o servidor executar uma tarefa.

Disciplina de escalonamento: é a regra (algoritmo) utilizada para adicionar e remover os clientes de uma fila. Dentre as regras possíveis de serem utilizadas destacam-se a FIFO, na qual a ordem de atendimento obedece à ordem de chegada; a LIFO, semelhante a uma pilha de pratos onde o último a entrar é o primeiro a sair e RR onde cada cliente é atendido durante certo intervalo de tempo denominado *quantum-time*.

Número de servidores: é possível haver mais de um servidor por centro de serviço. Logo, é necessário determinar qual desses servidores atenderá o próximo cliente. A escolha pode ser feita aleatoriamente, por algum critério de prioridade e até mesmo pelo servidor que está mais (ou menos) tempo desocupado.

Capacidade do sistema: caso a capacidade do sistema não seja infinita, alguns clientes poderão ser perdidos ou descartados [Baliero, (2005)].

### 2.5.1 Notação de Kendall

A notação de Kendall [Kleinrock, (1975)] explicita as características do centro de serviço. Por isso, ela é utilizada para denotar as redes de filas. Esta notação é definida pela quintupla:

|



A/S/c/k/m

Excluído: ¶

Onde:

“A” representa a distribuição estatística da taxa de chegada;

“S” representa a distribuição estatística da taxa de serviço;

“c” representa o número de servidores no centro de serviço;

“k” é o número máximo de usuários que podem estar na fila concomitantemente, ou seja, representa a capacidade da fila;

“m” é o número máximo de usuários na população;

Quando “k” e “m” são infinitos, a notação é reduzida para

A/S/c

As distribuições mais recorrentes nas taxas de chegada e serviço são diretamente representadas por letras específicas, que são:

M: distribuição exponencial;

D: taxa constante;

E<sub>k</sub>: Distribuição k-Erlang;

H<sub>k</sub>: Distribuição hiperexponencial de k estados;

### 2.5.2 A Fila M/M/1

Se o servidor estiver livre, quando o cliente chega ao sistema, ele faz o atendimento imediatamente. Caso contrário, o cliente se une à fila associada aquele servidor e aguarda sua vez. No momento apropriado, o cliente é selecionado para atendimento, que ocorre de acordo com a disciplina da fila. O serviço requisitado é então executado pelo servidor e ao seu término o cliente abandona o sistema.

[Machado, (2008)]. [Esta fila é representada pela Figura 2.4.](#)

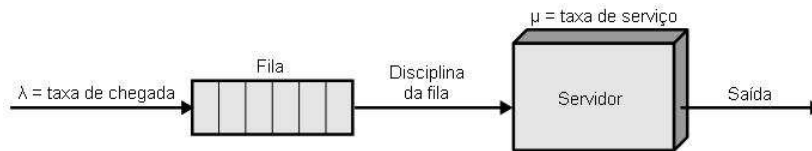


Figura 2.4: A representação gráfica de uma fila M/M/1.

### 2.5.3 Filas Multi-Servidor

No modelo ilustrado na Figura 2.5 vários servidores compartilham uma mesma fila. É comum considerar “N” como o número de servidores e, também, que todos os servidores são iguais. Essa consideração torna indiferente a escolha do servidor para o qual a tarefa será alocada. Se um cliente chega e há, pelo menos, um servidor livre, esse cliente é atendido imediatamente. Mas, se todos os servidores estão ocupados, a fila é formada.

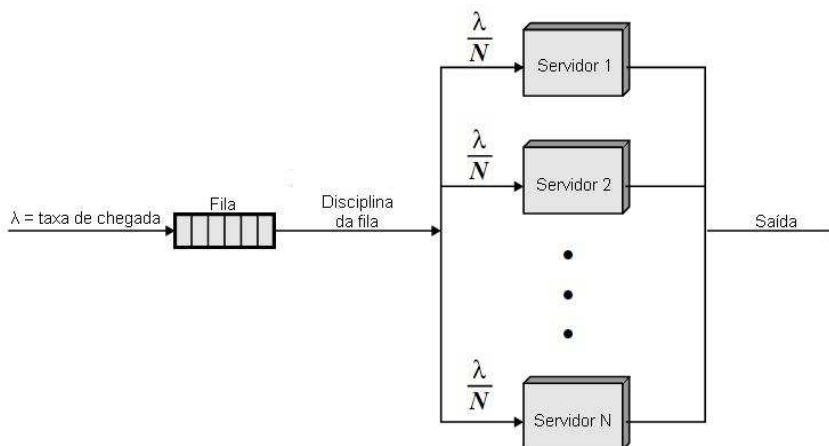


Figura 2.5: A representação gráfica de uma fila com vários servidores.

### 2.5.4 Múltiplas Filas com Servidores Únicos

Em contraste com o modelo anterior, temos o modelo que trabalha com vários servidores, porém cada um possuindo sua própria fila. Este modelo é mostrado na Figura 2.6.

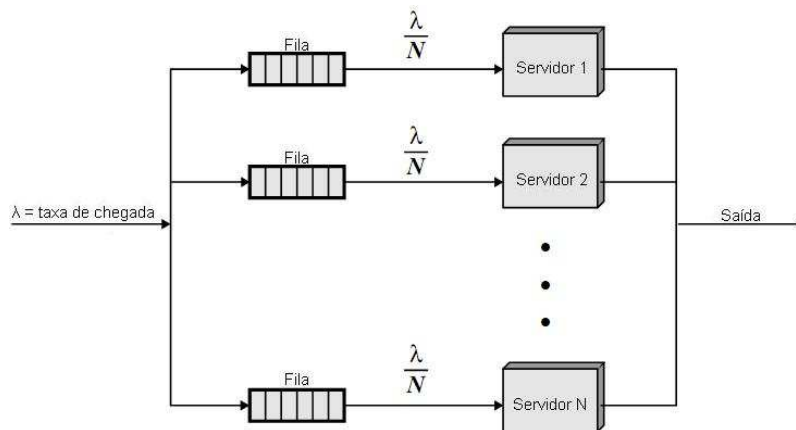


Figura 2.6: A representação gráfica de várias filas e vários servidores.

## 2.6 Métricas de Saída

Métricas são utilizadas para quantificar e mensurar o desempenho de um sistema e através delas os resultados podem ser avaliados com o intuito de tirar conclusões e gerar mudanças para melhorar o sistema [Falavinha Jr, (2009)].

Excluído: d

A execução de uma simulação tem como objetivo a apresentação de resultados relativos a essas métricas de desempenho de interesse do usuário. As métricas de saída são importantes, pois é através delas que o usuário verifica o comportamento do sistema simulado.

### 2.6.1 Tempo Total Simulado

Conforme o nome sugere, essa métrica aponta o tempo total simulado. Isso é feito marcando o instante de tempo em que a simulação começa e o tempo em que ela termina. Após isso, o tempo inicial é subtraído do tempo final conforme a equação 2.2.

Excluído: final

Excluído: inicial

$$\text{Tempo Total Simulado} = \text{Tempo Final} - \text{Tempo Inicial} \quad (2.2)$$

### 2.6.2 Ociosidade

Ociosidade do sistema é o tempo que a plataforma ficou sem uso. Ele é calculado utilizando o tempo em que os servidores ficaram disponíveis, isto é, o tempo em que não havia tarefa sendo executadas em algum servidor.

Então, somando-se o tempo total que um servidor ficou livre e dividindo pelo tempo total da simulação obtemos a ociosidade do servidor. Para a ociosidade total do sistema, fazemos isso para todos os servidores.

Portanto, podemos calcular a ociosidade de um servidor utilizando a equação 2.3 e a ociosidade total do sistema pela equação 2.4.

$$\text{Ociosidade do servidor (\%)} = (T_{\text{livre}} / T_{\text{simulação}}) \times 100 \quad (2.3)$$

$$\text{Ociosidade Total (\%)} = (\sum \text{Ociosidade de todos os servidores}) / \text{número total de servidores} \quad (2.4)$$

### 2.6.3 Eficiência

A eficiência é uma métrica de saída que aponta o quanto o sistema foi eficiente para uma tarefa, ou seja, ela é uma métrica que mede o quão bem o sistema está sendo utilizado. Ela é obtida através da carga exigida pela tarefa e pelo quanto ela recebe de potência computacional do sistema.

Para calcularmos a eficiência do nosso sistema, calculamos a carga computacional da tarefa e dividimos pela capacidade recebida multiplicada pelo tempo de execução. Após isso, obtemos a eficiência média da tarefa (equação 2.5). Para finalizar, efetuamos uma média entre todas as tarefas do sistema (equação 2.6). Se o valor determinado na equação 2.8 for acima de 70% ela é classificada como boa, entre 70% e 40% como média e, abaixo disso como ruim.

$$\text{Eficiência (\%)} = \text{Carga Computacional} / (\text{Capacidade Recebida} \times \text{Tempo Execução}) \quad (2.5)$$

$$\text{Eficiência média das tarefas (\%)} = (\sum \text{Todas as eficiências}) / \text{número total de tarefas} \quad (2.6)$$

### 2.6.4 Satisfação

A satisfação do usuário é uma métrica de desempenho que mostra quão satisfeito está um determinado usuário com o sistema. Satisfação é algo complexo de definir e de medir, uma vez que muitos fatores podem ser considerados como qualidade do serviço, tempo médio de resposta e tempo de entrega (Falavinha Jr., 2009).

Para o nosso simulador, essa métrica é calculada baseando-se no comportamento ideal esperado pelo usuário, que deseja ter o sistema inteiro para executar suas tarefas, e o comportamento real do sistema. De acordo com o tamanho de cada tarefa e o instante em que elas chegam ao sistema é possível determinar o instante final esperado para sua finalização. Quanto mais o tempo real de finalização se aproximar do tempo ideal esperado, mais satisfeito o usuário estará. Essa métrica é calculada da seguinte forma:

- Primeiramente calcula-se a satisfação para cada tarefa desse usuário utilizando a equação 2.7, na qual  $T_{ideal}$  representa o tempo em que a tarefa terminaria, por exemplo, se uma tarefa com carga de 15 unidades de processamento chegasse no instante 10 e o servidor processa a uma taxa de 1 unidade de processamento por u.t.,  $T_{ideal}$  será 25. Já  $T_{real}$  é o instante em que ela realmente acabou de executar. A mudança entre  $T_{ideal}$  e  $T_{real}$  pode acontecer porque a tarefa pode parar de executar. A mudança entre  $T_{ideal}$  e  $T_{real}$  pode acontecer porque a tarefa pode parar de executar.
- Após o cálculo da satisfação de cada tarefa, a média de satisfação é calculada pela equação 2.8 em que a satisfação de todas as tarefas são somadas e o resultado desta operação é dividido pelo número total de tarefas.

$$\text{Satisfação da tarefa (\%)} = ((T_{ideal} - T_{envio}) / (T_{real} - T_{envio})) \times 100 \quad (2.7)$$

$$\text{Satisfação Total (\%)} = (\sum \text{Satisfação de todas as tarefas}) / \text{número total de tarefas} \quad (2.8)$$

### 2.6.5 Métricas Obtidas das Tarefas

Através das tarefas é possível obter várias métricas como o tempo médio de espera, o tempo médio de servidor, tempo médio de sistema e tempo médio entre chegada das tarefas. O simulador implementado mede três tempos que são:

- Tempo médio de fila – é o tempo médio que todas as tarefas esperam para começar sua execução. Ele é medido, primeiramente, calculando a diferença entre o tempo que a tarefa começa sua execução no servidor e o tempo em que ela chegou ao sistema (equação 2.9), conforme a equação 2.10. Depois disso, calcula-se a média dos tempos (equação 2.10).

$$\text{Tempo da tarefa na fila} = T_{\text{entrada no servidor}} - T_{\text{entrada no sistema}} \quad (2.9)$$

$$\text{Tempo médio de espera} = \frac{(\sum \text{Tempo de todas as tarefas na fila})}{\text{número total de tarefas}} \quad (2.10)$$

Formatado: Fonte: 12 pt

- Tempo médio de servidor – essa medida representa o tempo médio que o servidor foi utilizado pelas tarefas de uma fila, ou seja, a média de uso do servidor por uma tarefa. Ele é obtido de forma análoga ao tempo médio de fila conforme as equações 2.11 e 2.12.

$$\text{Tempo da tarefa no servidor} = T_{\text{saída do sistema}} - T_{\text{entrada no servidor}} \quad (2.11)$$

$$\text{Tempo médio de servidor} = \frac{(\sum \text{Tempo das tarefas no servidor})}{\text{número total de tarefas}} \quad (2.12)$$

Formatado: Fonte: 12 pt

- Tempo médio de sistema – a última métrica calcula diz respeito ao tempo médio em que cada tarefa ficou no sistema. Ele é calculado da mesma maneira que as métricas de tempo médio de fila e servidor. Seus cálculos estão detalhados nas equações 2.13 e 2.14.

$$\text{Tempo da tarefa no sistema} = T_{\text{saída do sistema}} - T_{\text{entrada na fila}} \quad (2.13)$$

$$\text{Tempo médio das tarefas no sistema} = \frac{(\sum \text{Tempo de todas as tarefas no sistema})}{\text{número total de tarefas}} \quad (2.14)$$

Formatado: Fonte: 12 pt

Excluído: que o sistema desenvolvido

## 2.7 Considerações Finais

Excluído: ¶  
¶

Este capítulo tratou de todo o embasamento teórico necessário para sustentar o desenvolvimento do projeto. O estudo de computação de alto desempenho e dos simuladores de grades computacionais existentes contribuiu para a identificação dos principais parâmetros envolvidos na construção de um simulador de grades computacionais. Em seguida, foi feita uma breve discussão a respeito de técnicas de modelagem e avaliação de desempenho, mais precisamente, de simulação. Em conjunto com simulação, foram vistos temas como teoria das filas e técnicas para geração de números aleatórios, pois estes tópicos complementam o processo de simulação. Para finalizar, as métricas de saída definidas para o motor de simulação do iSPD foram abordadas possibilitando o desenvolvimento integral do motor de simulação que, por sua vez, será discutido no capítulo seguinte.

## CAPÍTULO 3

### Desenvolvimento do motor de simulação

Neste capítulo apresenta-se todo o processo de desenvolvimento do projeto. O paradigma de programação utilizado para implementar o motor de simulação foi o Orientado a Objetos e a linguagem escolhida foi Java. Aproveitando os benefícios dessas ferramentas, os vários componentes do motor de simulação foram divididos em estruturas do tipo pacote e, para facilitar a compreensão das soluções propostas, os pacotes criados (Simulação, Redes de Filas e Números Aleatórios) são apresentados separadamente, assim como as classes pertencentes a cada um desses pacotes. Premissas de engenharia de *software* consideram a abordagem modular uma boa prática de programação, pois ela facilita o reuso e a extensão do *software*. Além disso, optou-se por também oferecer separadamente o diagrama de classes da modelagem UML desses pacotes para que se tenha uma maior clareza do projeto, o que é feito no Apêndice A. A notação UML utilizada neste texto é a apresentada em [Guedes, (2009)].



### 3.1 O interpretador de modelos e o modelo simulável

Como esse projeto é uma parte de um projeto global, o iSPD, é prudente fazer sua contextualização final, o motor de simulação recebe dados de outras partes da ferramenta para poder executar a simulação, sendo então necessário saber que dados são esses e como eles são recebidos.

No momento em que o usuário utiliza o iSPD, ele interage com a ferramenta através de sua interface gráfica, mostrada na Figura 3.1. Essa interação consiste em inserir os ícones desejados e configurar os parâmetros deles com os valores que representem o seu sistema computacional. Após essa fase, o interpretador de modelos organiza os dados num modelo simulável e passa esse modelo para o motor de simulação.

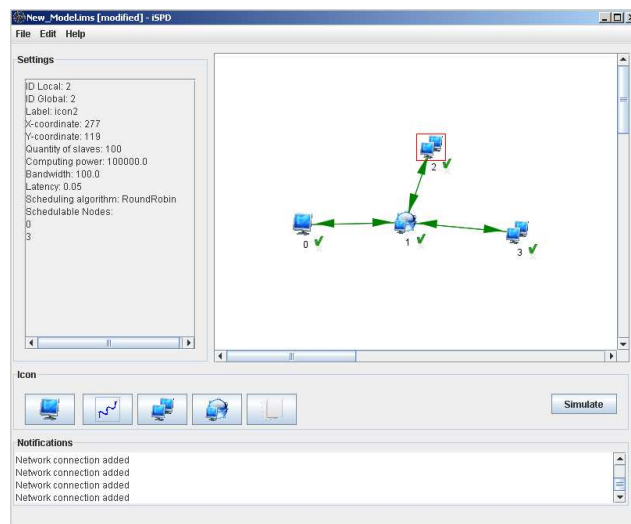


Figura 3.1: A interface gráfica do simulador iSPD.

O modelo simulável é composto por uma descrição funcional do sistema, passada através de um arquivo gerado no interpretador de modelos. Ele contém os dados necessários para o funcionamento correto do motor de simulação. O conjunto dos dados que parametrizam o sistema computacional a ser simulado e que podem pertencer ao modelo simulável são formados pelos seguintes elementos:

- Ícone máquina: poder computacional, taxa de ocupação e um dado que diz se a máquina é mestre ou escravo;
- Ícone cluster: quantidade de nós (incluindo o nó mestre), poder computacional,

largura de banda e latência da rede de interconexão do cluster e algoritmo de escalonamento das tarefas;

- Ícone enlace de rede: largura de banda, latência e taxa de ocupação;
- Ícone de Internet: largura de banda, latência e taxa de ocupação.
- Ícone de tarefas: número de tarefas; tamanho médio, mínimo, máximo e valor de probabilidade de processamento das tarefas; tamanho médio, mínimo, máximo e valor de probabilidade de tamanho de comunicação das tarefas e tempo médio de chegada das tarefas na fila; quantidade de tarefas iniciais de cada nó escalonador.

O conjunto de tarefas é caracterizado pelo tempo médio de chegadas da tarefa na fila, tamanho médio de processamento das tarefas e tamanho médio de comunicação das tarefas. De acordo com resultados anteriores [Lublin et al., (2003)], há determinadas distribuições de probabilidade que representam cada uma dessas grandezas mais fidedignamente. O tempo médio de chegadas na fila é mais bem representado pela distribuição exponencial cujo único parâmetro necessário para se gerar um número que respeite essa distribuição é a média da distribuição. Já os volumes de processamento e de comunicação são mais bem representados por uma distribuição chamada “*two-stage Uniform*” [Lublin et al., (2003)] e, como será visto mais adiante [na seção 3.5.1](#), são demandados quatro parâmetros para se gerar um número dessa distribuição.

Após o usuário inserir esses dados na interface gráfica da ferramenta e o interpretador de modelos preparar o modelo simulável, as operações do motor de simulação podem ser iniciadas.

### 3.2 Os eventos do motor de simulação

A definição de quais eventos podem causar mudanças no estado do sistema é uma etapa imprescindível na construção de uma ferramenta de simulação. Para facilitar o entendimento do projeto, os eventos definidos para ele serão apresentados em dois níveis. O primeiro nível, mais alto, é descrito a seguir, considerando o sistema da Figura 3.2. O segundo nível, mais baixo e detalhado, é descrito mais adiante, na seção 3.3.

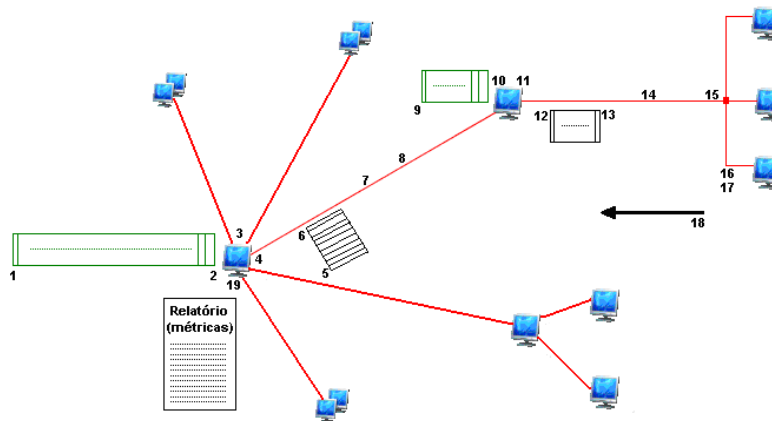


Figura 3.2: Os possíveis eventos de uma grade computacional.

Os números presentes na Figura 3.2 indicam as seguintes ações:

- 1) Chegada de uma tarefa na fila da grade (sistema “todo”);
- 2) Entrada da tarefa no  $FE_{grade}$  para início efetivo de todo o processo;
- 3) Sistema escalona/processa a tarefa;
- 4) Saída da tarefa do  $FE_{grade}$  ;
- 5) Chegada da tarefa na fila de um sistema de comunicação (supondo que o escalonamento determinou a ida da tarefa para o cluster no canto superior direito);
- 6) Entrada da tarefa no sistema de comunicação da grade;
- 7) Sistema de comunicação da grade envia a tarefa;
- 8) Saída da tarefa do sistema de comunicação da grade;
- 9) Chegada da tarefa na fila de um *cluster* (no  $FE_{cluster}$ );
- 10) Entrada da tarefa no  $FE_{cluster}$  para ser escalonada;
- 11) Saída da tarefa do  $FE_{cluster}$ ;
- 12) Chegada da tarefa na fila do sistema de comunicação;
- 13) Entrada da tarefa no sistema de comunicação do *cluster*;
- 14) Sistema de comunicação do *cluster* processa a tarefa;
- 15) Saída da tarefa do sistema de comunicação do *cluster*;
- 16) Entrada da tarefa em um sistema de processamento do *cluster* (sistemas de processamento não terão filas, ou ele está livre e recebe a tarefa, ou ele está ocupado e a tarefa fica na fila do próprio  $FE_{cluster}$ );
- 17) Sistema de processamento processa a tarefa;

- 18) DEVOLUÇÃO PARCIAL: Sistema de processamento devolve os resultados para o  $FE_{cluster}$ ;
- 19) DEVOLUÇÃO TOTAL:  $FE_{cluster}$  devolve resultados para o  $FE_{grade}$ ;

### 3.3 O pacote Simulação

Este pacote é o núcleo do motor de simulação. É composto pelas classes Simulacao.java, ListaEventosFuturos.java, NoLEF.java e Relogio.java.

#### 3.3.1 A classe Simulacao.java

A classe Simulacao.java pode ser considerada o núcleo do motor de simulação. Seus métodos realizam as três atividades principais do motor: a montagem da infraestrutura computacional da grade a ser simulada, a criação inicial das tarefas executadas pela infraestrutura previamente montada e, finalmente, a simulação propriamente dita.

Inicialmente, alguns objetos de outras classes são instanciados, são eles: um objeto da classe “RedesDeFilas.java”, outro da classe “ListaEventosFuturos.java”, um terceiro da classe “GeracaoNumAleatorios.java” e um último da classe “Relógio.java”. Em seguida, o método que monta a infraestrutura do sistema simulado é chamado e os elementos da rede de filas - centros de serviço, as ligações entre eles, seus escravos, servidores e filas - são adicionados a ela. Como essas adições são feitas no objeto redeDeFilas, elas são realizadas através de chamadas a determinados métodos da classe “RedesDeFilas.java” que serão abordados posteriormente [na seção 3.4.1](#).

Após a fase de montagem da infraestrutura computacional, faz-se a criação das tarefas executadas no sistema. Essa atividade pode ser feita de dois modos, sendo a diferença entre eles o fato de que em um deles a quantidade inicial de tarefas alocadas em cada nó escalonador da grade é determinada pelo usuário, já no outro essa característica é determinada automaticamente pelo sistema. Em comum, ambos os modos oferecem a opção do usuário determinar os parâmetros (tempo médio de

chegadas entre as tarefas e os valores mínimo, médio, máximo e de probabilidade para os tamanhos de processamento computacional e de comunicação das tarefas) necessários para a criação das tarefas em si.

Quando é o sistema que deve determinar a quantidade inicial de tarefas alocadas em cada nó escalonador, ele o faz dividindo a quantidade de tarefas igualmente entre cada escalonador, com o detalhe do último escalonador ficar também com o resto desta divisão.

A criação de cada tarefa propriamente dita consiste em, após a determinação da quantidade inicial de tarefas alocadas em cada nó escalonador, fazer, através do objeto da classe `GeracaoNumAleatorios.java`, a geração dos números aleatórios que representarão o tempo de chegada na fila e os tamanhos de processamento e de comunicação de cada uma dessas tarefas e, logo em seguida, adicionar cada tarefa ao centro de serviço previamente determinado para ela.

Posteriormente à criação das tarefas, há a chamada ao principal método do motor (`iniciaSimulacao`) e a simulação em si passa a acontecer. Conceitualmente, esse método pode ser dividido em duas partes que juntas formam a implementação do algoritmo *Event Scheduling/Time Advance*: a de controle e a da execução da simulação. O controle consiste em inserir e retirar, sempre de modo adequado, os eventos da LEF e atualizar o relógio da simulação. Enquanto isso, a execução consiste em efetivamente realizar o evento mais recente da lista de eventos futuros e agendar possíveis eventos futuros gerados pelo evento atual.

Com isso, o método adota a seguinte ordem: enquanto há tarefas que ainda não foram alocadas para seus respectivos centros de serviço, elas vão sendo alocadas e as que já foram alocadas habilitam o acontecimento de outros eventos, como por exemplo, o escalonamento de uma delas. Esses novos eventos podem acontecer antes mesmo de ainda restarem tarefas para serem alocadas, para isso basta o tempo de ocorrência deles ser menor que o tempo de chegada da tarefa na sua respectiva fila. Quando todas as tarefas já tiverem sido alocadas nos seus nós escalonadores iniciais, a simulação consiste apenas em repetir as operações: executar o evento atual e agendar possíveis novos eventos que esse evento atual gere. Essa repetição continua até todas as tarefas criadas inicialmente receberem a marcação “finalizada”.

A seguir, os eventos passíveis de acontecer no sistema e as interações entre eles serão descritos, mas para facilitar o entendimento da maneira com a simulação

**Excluído:** as tarefas já alocadas

acontece de fato, é possível analisar a Rede de Petri apresentada na Figura 3.3.

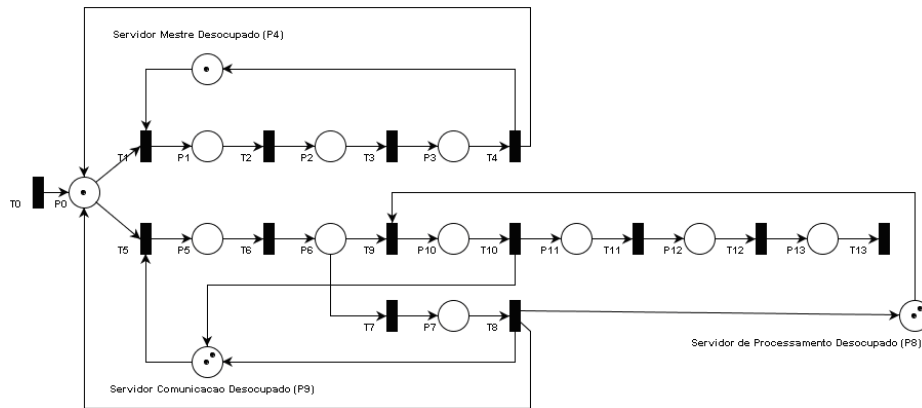


Figura 3.3: A modelagem com Rede de Petri dos eventos da simulação.

Tabela 3.1 – Os possíveis eventos da simulação.

Lugares		Transições	
P0	Chegada de uma tarefa a uma fila do sistema.	T0	Criação de tarefas.
P1	Entrada de uma tarefa em um nó escalonador/mestre.	T1	Migração da tarefa de uma fila para um nó escalonador.
P2	Escalonamento de uma tarefa.	T2	Habilitação do escalonamento de uma tarefa.
P3	Saída de uma tarefa de um nó escalonador.	T3	Habilitação da saída da tarefa de um nó escalonador.
P4	Servidor mestre desocupado.	T4	Alteração do estado do servidor mestre para livre e colocação da tarefa atual na fila de outro centro de serviço
P5	Entrada de uma tarefa em um nó de comunicação.	T5	Migração da tarefa de uma fila para um nó de comunicação.
P6	Envio de uma tarefa.	T6	Habilitação do envio de uma tarefa.
P7	Saída de uma tarefa em direção a um nó de comunicação intermediário.	T7	Migração da tarefa de um nó de comunicação para outro nó de comunicação (intermediário)
P8	Servidor de processamento desocupado.	T8	Habilitação da saída da tarefa de um nó de comunicação
P9	Servidor de comunicação desocupado.	T9	Migração da tarefa de um nó de comunicação para seu destino final.
P10	Saída de uma tarefa em direção ao destino final.	T10	Migração da tarefa de um nó de comunicação seu destino final.
P11	Entrada da tarefa em um servidor de processamento.	T11	Habilitação do processamento de uma tarefa.
P12	Processamento de uma tarefa.	T12	Habilitação para finalizar uma tarefa.
P13	Finalização de uma tarefa.	T13	Término da simulação

Após a definição dos eventos da Tabela 3.1, constatou-se que dois deles - processamento e envio de tarefas - consomem quantidades de tempo relevantes para os modelos simulados e, assim, alteram o relógio da simulação. Isso acontece, respectivamente, de acordo com as equações 3.1 e 3.2.

$$\text{tempo de processamento} = \frac{\text{tamanho computacional da tarefa}}{\text{poder computacional do servidor de processamento}} \quad (3.1)$$

$$\text{Tempo de comunicação} = \frac{\text{tamanho de comunicação da tarefa}}{\text{largura de banda do servidor de comunicação}} + \text{latência da rede} \quad (3.2)$$

. A partir daí, a implementação do método foi feita de modo a proporcionar o funcionamento relatado anteriormente. O exemplo da codificação de um dos eventos (evento 5 – que corresponde ao lugar P6 (envio da tarefa) da Rede de Petri) pode ser visto na Figura 3.4. Através desse exemplo, consegue-se perceber que o método foi construído para receber o evento atual da lista de eventos futuros, considerar seu tipo e realizar as ações necessárias. Além disso, há algumas ações comuns a todos os eventos que são: primeiramente atualizar o relógio da simulação, realizar a atividade particular do evento, remover o evento atual da lista de eventos futuros e gerar, sob determinadas condições, novos eventos futuros, adicionando-os à LEF. Se formos considerar a divisão conceitual do motor feita anteriormente, a linha 701 corresponde à parte de execução da simulação e o restante à parte de controle. Assim, fica evidente também que a parte de execução da simulação é realizada pelas outras classes do motor e não pela classe Simulação.java.

```

696 switch( eventoAtual.getTipoEvento() )
697 {
698     .
699     .
700     case 5: relógioSimulacao.atualizaRelógioSimulacao( eventoAtual.getTempoOcorrencia() );
701             incrementoRelógioEE = redeDeFilas.envioTarefa( tarefaAtual );
702             listaEF.poll();
703
704             csAtualEhEscalonador = redeDeFilas.csAtualEhEscalonador( tarefaAtual.getIdCSAtual() )
705             if( tarefaAtual.getTtl() > 1 && tarefaAtual.getFinalizada() == false )
706             { listaEF.adicionaEvento( 6, tarefaAtual, eventoAtual.getTempoOcorrencia() +
707               incrementoRelógioEE );
708             }
709             else if( tarefaAtual.getTtl() <= 1 && tarefaAtual.getFinalizada() == false )
710             { listaEF.adicionaEvento( 7, tarefaAtual, eventoAtual.getTempoOcorrencia() +
711               incrementoRelógioEE );
712             }
713             break;
714     .
715     .
716     .
717 }

```

Figura 3.4: A implementação da chamada do evento “envio de tarefa”.



### 3.3.2 As classes `ListaEventosFuturos.java`, `NoLEF.java` e `Relógio.java`

A classe “`ListaEventosFuturos.java`” suporta a LEF do motor de simulação e possui métodos para manipular os objetos da classe “`NoLEF.java`”. Os principais métodos adicionam e retiram eventos da LEF. Os outros métodos fornecem o suporte necessário para que os métodos principais exerçam suas funções.

Dois recursos interessantes da linguagem Java para a implementação de filas são a classe `PriorityQueue` e a interface `Queue` [Deitel et al., (2005)]. A interface `Queue` estende a interface `Collection` e fornece operações para inserir, remover e inspecionar elementos em uma fila. A classe `PriorityQueue` permite inserções de tal modo que o elemento de maior prioridade será o primeiro elemento a ser removido da estrutura. Esses recursos foram utilizados na implementação da classe `ListaEventosFuturos.java`, pois a inserção na LEF deve ser ordenada crescentemente pelo instante de ocorrência dos seus elementos, ou seja, os eventos de menor instante de ocorrência devem ser inseridos de maneira a permitir a retirada (escalonamento) deles antes dos outros.

O método `compareTo` é um método da classe `PriorityQueue` da linguagem Java que deve ser sobrescrito para poder indicar para sua classe qual será o critério de ordenação da fila de prioridade implementada na classe; no caso da classe `NoLEF.java`, ele foi sobrescrito para indicar que o instante de ocorrência da tarefa na fila (`tempoOcorrencia`) é o critério de ordenação, ou seja, as tarefas têm que ser ordenadas crescentemente pelo seus tempos de chegada na fila. A Figura 3.5 mostra o método `compareTo` da classe `NoLEF.java`.

```
public int compareTo( NoLEF aux )
{   return tempoOcorrencia.compareTo( aux.tempoOcorrencia );
}
```

Figura 3.5: O método `compareTo` da classe `NoLEF.java`.

Por sua vez, a classe `NoLEF.java` é a responsável por objetos que contêm as informações necessárias para a realização de cada um dos eventos da simulação. Essas informações - tais como instante de ocorrência do evento, identificador do centro de serviço onde o evento ocorrerá e dos recursos envolvidos (servidor, fila e tarefa) - são armazenadas nas variáveis de instância da classe para poderem ser

acessadas pelo motor de simulação.

O objeto da classe `Relogio.java` controla o tempo da simulação. A classe contém apenas uma variável de instância, que armazena o valor do tempo da simulação, e alguns métodos que realizam as seguintes funções: inicialização da variável, atualização da variável, incremento de um valor à variável e retorno de seu valor atual.

### 3.4 O pacote Rede de Filas

O pacote `RedesDeFilas.java` é o conjunto das classes que abstraem a teoria das filas e permitem a sua execução computacional. Os elementos desse conjunto permitem a representação tanto da infraestrutura computacional dos sistemas distribuídos quanto das tarefas a serem executadas neles. As classes do pacote são:

`RedesDeFilas.java`, `CentrosDeServico.java`, `Servidores.java`, `Filas.java` e `NoFila.java`.

**Excluído:** A Figura 4 ilustra a modelagem deste pacote.

#### 3.4.1 A classe `RedesDeFilas.java`

A ideia por trás desta classe é tratar os diferentes centros de serviço presentes no modelo a ser simulado como vértices de um grafo e armazenar suas ligações numa matriz de adjacência. Assim, a matriz será quadrada, ou seja, tanto a quantidade de linhas quanto a quantidade de colunas são o número de centros de serviço presentes no modelo. Os elementos da matriz serão “1” ou “0” representando o fato de haver ou não ligações entre os centros de serviço representados nas linhas e aqueles representados nas colunas.

Essa classe contém poucas variáveis de instância, pois todo trabalho de manipulação das informações para a simulação foi repassado para as outras classes deste pacote, cabendo à classe `RedesDeFilas.java` apenas o gerenciamento da estrutura e não o processamento dos dados. Isso quer dizer que, devido tanto à estrutura de codificação quanto às estruturas de dados utilizadas na implementação do projeto, os métodos da classe `RedesDeFilas.java`, na grande maioria das vezes, não executam efetivamente os eventos da simulação, o que eles fazem na verdade é

apenas localizar um determinado elemento (aquele no qual o evento ocorrerá) do seu conjunto de centros de serviço e repassar para este elemento a responsabilidade pela execução deste evento. Então, um fato a se notar é que foi dado a alguns desses métodos os mesmos nomes dos métodos de outras classes, mas isso foi feito propositalmente para que essa coincidência de funções se torne mais explícita, facilitando o entendimento do projeto.

Excluído: o

Apesar do grande número de métodos da classe, a compreensão das ações da grande maioria deles é feita facilmente apenas com a observação de suas implementações. Para tornar esse entendimento ainda mais simples é possível agrupá-los em algumas categorias de acordo com o papel exercido por eles. Esses grupos podem ser classificados como:

- Métodos de montagem da infraestrutura: `RedesDeFilas`, `adicionaCentroServico`, `adicionaServidorProcto`, `adicionaServidorCom`, `adicionaServidoresClr`, `adicionaFila`, `inteligasCSs`;
- Iniciadores dos eventos da simulação: `adicionaTarefaFila`, `entradaTarefaMestre`, `escalonaTarefa`, `saidaTarefaMestre`, `envioTarefa`, `entradaTarefaServComun`, `saidaTarefaServComunHopIntermediario`, `saidaTarefaServComunHopFinal`, `entradaTarefaServProc`, `saidaTarefaServidor`, `processamentoTarefa`;
- Auxiliares: `peekFilaMestre`, `peekFilaComun`, `determinaCaminho`, `verificaSeServEMestre`, `verificaMestreLivre`, `verificaFilaComunVazia`, `Dijkstra`, `verificaFilaMestreVazia`, `verificaSeServidorEstaLivre`, `confereRF`, `localizaCSanterior`, `buscaIdTarefaAlocadaMestre`, `instanciaMatrizVetor`, `buscaIdTarefaAlocadaServidor`, `csAtualEhEscalonador`;

Entre todos esses métodos, um deles merece atenção especial: o método `escalonaTarefa` do grupo dos métodos iniciadores dos eventos da simulação. Esse método pode ser dividido em duas partes, uma para o escalonamento local (interno) ou global (externo) das tarefas e outra para o roteamento da tarefa, ou seja, ele determina o caminho que a tarefa terá de percorrer para ir do seu centro de serviço atual até seu centro de serviço de destino. Atualmente o algoritmo de escalonamento implementado para ambos os casos de escalonamento (externos e internos) é o *Round-Robin* e o algoritmo de roteamento é baseado no algoritmo de Dijkstra

[Boaventura et al., (2009) para a determinação de caminhos mínimos entre dois vértices de um grafo.

### 3.4.2 A classe CentrosDeServico.java

Os métodos dessa classe são análogos aos métodos da classe RedesDeFilas.java e possuem as mesmas funções. As diferenças são seu construtor, os métodos *set* e *get* e o fato de que suas operações, agora são realizadas no patamar de um único centro de serviço, seus servidores e filas. Assim como na classe RedesDeFilas.java, alguns métodos da classe CentrosDeServico.java repassam a execução de suas atividades para as classes Servidores.java e Filas.java. Um exemplo é dado na Figura 3.6, que mostra o método `removeElementosFila` instanciando um objeto temporário da classe Filas.java e chamando o método `removeElementosFila` desse objeto para efetivamente remover o elemento.

```
public void removeElementosFila( int idFila )
{
    for( Filas temp : filas )
    {
        if( temp.getIdFila() == idFila )
        {
            temp.removeElementosFila();
            System.out.printf( "\t|--> Do Centro de Servico ID: %d\n", idCS );
        }
    }
}
```

Figura 3.6: O método `removeElementosFila` da classe `CentrosDeServico.java`

É importante destacar que para atender os requisitos do projeto, cada centro de serviço representa um entre quatro tipos disponíveis e que as ações e métodos efetuados por cada centro de serviço dependem do tipo dele e, conseqüentemente, de sua função específica na grade computacional. Os tipos são:

- Tipo 0: centros de serviços que contêm apenas uma máquina, seja um servidor mestre ou um servidor de processamento;
- Tipo 1: centros de serviço que contêm conglomerados homogêneos de máquinas (*clusters*);
- Tipo 2: centros de serviço que representam enlaces de rede ponto-a-ponto;

**Excluído:** que

**Excluído:** efetua

**Excluído:** e seu

- Tipo 3: centros de serviço que representam conexões de rede do tipo “Internet”.

Além do tipo, as variáveis de instância da classe `CentrosDeServico.java` armazenam informações como: identificador do centro de serviço, número máximo de servidores, número atual de servidores, número atual de servidores livres, número máximo de filas, número atual de filas, um vetor com os identificadores de seus escravos, posição do escravo que atualmente recebeu uma tarefa (para o algoritmo de escalonamento), entre outras.

### 3.4.3 A classe `Servidores.java`

Dentro da solução proposta neste projeto para a simulação de grades computacionais, a classe `Servidores.java` foi pensada para seus objetos representarem os elementos de processamento propriamente ditos, ou seja, os nós dos sistemas distribuídos.

A partir daí, as variáveis de instância dessa classe e as informações armazenadas por elas são: `idServidor` (identificador do servidor), `tipoServidor` (processamento ou comunicação), `mestreEscravo`, `servidorLivre` (armazena o estado livre/ocupado do servidor), `idTarefaAtual` (sinaliza a tarefa que atualmente ocupa o servidor), `tServicoProcto` (se o servidor é do tipo processamento, armazena seu poder computacional), `tServicoRede` (se o servidor é do tipo comunicação, armazena sua largura de banda), `latência` (se o servidor é do tipo comunicação, armazena sua latência), `txOcupacao` (contém a taxa de ocupação do servidor).

Os métodos da classe `Servidores.java` consistem nos construtores da classe (um para cada tipo de servidor), nos métodos *set* e *get* das variáveis de instância e no método `atribuiTarefaServidor` para atribuir tarefas aos diferentes servidores do sistema. Este último método pode ser visto na Figura 3.7.

```

public Double atribuiTarefaServidor( NoFila tarefa )
{
    Double retornoFuncao = 0.0;

    setIdTarefaAtual( tarefa.getIdTarefa() );
    ocupaServidor();

    return retornoFuncao = 0.0;
}

```

Figura 3.7: O método atribuiTarefaServidor da classe Servidores.java

### 3.4.4 As classes NoFila.java e Filas.java

As tarefas a serem processadas pelas grades e pelos *clusters* são simuladas através dos objetos das classes Filas.java e NoFila.java.

Como a abordagem definida para implementar o simulador consiste em fazer cada tarefa passar pelos estágios indicados anteriormente na Tabela 3.1, há uma quantidade considerável de informações a ser armazenada na classe, entre elas temos: os identificadores da tarefa, da fila, dos centros de serviços e servidores atuais e de destino, a distância da tarefa até um próximo servidor mestre que possa redirecioná-la, o tempo de vida dela - aqui foi usado conceito semelhante ao campo TTL do cabeçalho IP -, um vetor indicando o caminho de centros de serviço pelos quais ela deve passar, os tamanhos de processamento e de comunicação da tarefa, o tempo de chegada dela nas filas e uma variável para sinalizar se o processamento da tarefa foi finalizado, ou seja, se ela já passou por todos os estágios do processo. Sobre os métodos da classe, é importante citar que o *compareTo*, assim como na classe NoLEF, foi sobrescrito para poder indicar para sua classe qual é o critério de ordenação da fila de prioridade; no caso da classe NoFila.java ele foi sobrescrito para indicar que o tempo de chegada da tarefa na fila (*tempoChegadaFila*) é o critério de ordenação

A classe Filas.java abstrai os conceitos para poder-se instanciar os objetos representantes das filas de tarefas a serem executadas nos sistemas distribuídos. Ela possui apenas três variáveis de instância: uma para identificar a fila, outra para controlar o número atual de tarefas da fila e um objeto da classe *Priority Queue* que constitui a fila em si. Para manipular essas variáveis, sete métodos foram

implementados. Entre eles, estão o construtor da classe, os métodos *set* e *get* da variável de identificação da fila, um que retorna o número atual de tarefas, outro para adicionar uma tarefa na fila, outro para retornar o primeiro elemento da fila sem removê-lo e, finalmente, um que faz a remoção de elementos.

### 3.5 O pacote Números Aleatórios

Este pacote contém apenas a classe geradora de números aleatórios. É através dele que qualquer número aleatório usado durante a simulação é gerado. Ele consiste em apenas uma classe que implementa os métodos de geração necessários.

#### 3.5.1 A classe `GeracaoNumAleatorios.java`

Esta é uma classe auxiliar para os outros elementos do motor de simulação. Ela é responsável por, por exemplo, gerar os tempos de ocorrência dos eventos a partir dos parâmetros passados pelo usuário da ferramenta.

Ela possui apenas uma variável de instância que recebe o valor da semente para a geração de números aleatórios. Existem também três métodos que geram valores pseudo-aleatórios pertencentes a uma das seguintes distribuições: normal, exponencial e *two-stage uniform*. Os valores pertencentes à distribuição normal são gerados através da técnica da simulação direta, já os valores pertencentes à distribuição exponencial são gerados pela técnica da transformação inversa e os valores pertencentes à *two-stage uniform* são gerados por um algoritmo próprio representado pela Figura 3.8.

```

public double twoStageUniform(double low, double med, double hi, double prob)
{
    double a;
    double b;
    double tsu;
    double u;

    u = Math.random();

    if (u <= prob)
    {
        a = low;
        b = med;
    }
    else
    {
        a = med;
        b = hi;
    }

    tsu = (Math.random() * (b-a) ) + a;
    return tsu;
}

```

Figura 3.8: Geração de números pertencentes à distribuição *two-stage uniform*.

### 3.6 O pacote Estatística

Este é o pacote responsável por colher todas às métricas do simulador e seu funcionamento acontece a partir da classe `Simulacao.java`. Ele atua no motor coletando os dados da simulação para efetuar os cálculos das métricas (ociosidade, eficiência, satisfação, tempo médio de fila, etc) e é composto pelas classes `Estatistica.java`, `CaixaTextoEstatistica.java`, `MetricaCS.java`, `MedidasServidor.java`, `NoMedidasServidor.java`, `TarefasFila.java` e `NoTarefasFila.java`. Seus diagramas UML estão detalhados no apêndice A.8 e A.9.

#### 3.6.1 As classes `Estatistica.java` e `CaixaTextoEstatistica.java`

Dentro da solução proposta neste projeto, a classe `Estatistica.java` foi pensada para que seus objetivos representem as métricas que o usuário deseja como satisfação, tempo médio das tarefas na fila e ociosidade do sistema. Essa classe é o núcleo do pacote estatístico e nela são efetuados todos os cálculos de ociosidade, eficiência, satisfação e tempo de simulação através do método *set* de cada variável. Para cada métrica são usadas as definições, apresentadas no capítulo 2.



Para um melhor funcionamento da parte de estatísticas do sistema, nessa classe também são criadas duas listas: uma para armazenar as métricas relacionadas a todos os centros de serviço do sistema e outra para dados relacionados às tarefas. Essas listas se fazem necessárias já que cada métrica relacionada aos centros de serviço, ou as tarefas, é calculada de maneira independente e, somente as métricas globais do sistema, são calculadas pela classe `Estatistica.java`.

Para a criação das listas utilizou-se a coleção “List” da linguagem Java [Deitel et al., (2005)]. Sua criação e funcionamento acontece a partir da classe principal do sistema, chamada `Simulacao.java`, que executa o método `Simulação` e um objeto da classe `Estatistica.java` é instanciado. Já o método `fimSimulacao` (Figura 3.10), da classe `Estatistica.java`, é chamado ao final da simulação pela classe principal e os cálculos de ociosidade, tempo total e satisfação do sistema são efetuados.

A classe `Estatistica.java` utiliza a classe `CaixaTextoEstatistica.java` para criar uma janela e apresentar os resultados para o usuário. Para a impressão desses resultados na tela, a classe `CaixaTextoEstatistica.java` é utilizada criando uma janela para apresentar os resultados ao usuário.

```

public void fimSimulacao()
{
    Double aux = 0.0;
    String texto = "\t\tSaída da Simulação\n\n";
    setTempoTotal();
    texto += "Tempo Total Simulado = " + String.valueOf(getTempoTotal()) + "\n";
    setSatisfacao();
    texto += "Satisfacao = " + String.valueOf(getSatisfacao()) + "\n";
    for(MetricaCS temp:metCS){
        aux = temp.finalizaServProc();
        if( aux > 0.0 ){
            texto += "CS ID = " + String.valueOf(temp.getCS()) + " GASTOU " + String.valueOf(aux) + " SEGUNDOS PROCESSANDO\n";
        }
        aux = temp.finalizaServCom();
        if( aux > 0.0 ){
            texto += "CS ID = " + String.valueOf(temp.getCS()) + " GASTOU " + String.valueOf(aux) + " SEGUNDOS COMUNICANDO\n";
        }
    }
    setEficiencia();
    texto += "Eficiencia = " + String.valueOf(getEficiencia()) + "\n";
    if( getEficiencia() > 70.0 )
    {
        texto += "Eficiencia BOA\n ";
    }
    else if( getEficiencia() > 40.0 )
    {
        texto += "Eficiencia MEDIA\n ";
    }
    else
    {
        texto += "Eficiencia RUIM\n ";
    }
    texto += "\tTAREFAS\n ";
    for( TarefasFila no:filas )
    {
        if( no.getTipoMetrica() == 0 )
        {
            texto += " Comunicacao \n";
            texto += "   Tempo medio de fila " + String.valueOf(no.getTempoMedioFila()) + "segundos.\n";
            texto += "   Tempo medio de processamento " + String.valueOf(no.getTempoMedioServidor()) + "segundos.\n";
            texto += "   Tempo medio no sistema " + String.valueOf(no.getTempoMedioSistema()) + "segundos.\n";
        }
        else
        {
            texto += " Processamento \n";
            texto += "   Tempo medio de fila " + String.valueOf(no.getTempoMedioFila()) + "segundos.\n";
            texto += "   Tempo medio de processamento " + String.valueOf(no.getTempoMedioServidor()) + "segundos.\n";
            texto += "   Tempo medio no sistema " + String.valueOf(no.getTempoMedioSistema()) + "segundos.\n";
        }
    }
    CaixaTextoEstatistica janela = new CaixaTextoEstatistica("Métricas", texto);
    janela.setVisible(true);
}

```

Figura 3.9: O método fimSimulacao da classe Estatistica.java.

### 3.6.2 As classes MetricaCS, MedidasServidor.java e NoMedidasServidor.java

Essas são as classes responsáveis por recolherem as métricas dos servidores do sistema. A primeira classe (MetricaCS.java) é a classe que armazena informações gerais dos centros de serviço do sistema como o identificador do centro de serviço (idCS) e os servidores.

Todas as vezes que o sistema cria um centro de serviço, um nó da lista “servidores” (declarada na classe MetricaCS.java) é criado, conforme a Figura 3.11. Isso é feito através do método addMedidasServidores da classe MetricaCS.java.

```

public void addMedidasServidor(int id, int tipoServ )
{
    MedidasServidor serv = new MedidasServidor(id, tipoServ);
    servidores.add(serv);
}

```

Figura 3.10: Método da classe `MetricaCS.java` que adiciona dados dos servidores ao centro de serviço.

Nesse nó são armazenadas informações relacionadas ao servidor como seu identificador (ID), o tipo do servidor (comunicação ou processamento), o tempo total que o servidor ficou livre (TTSLivre) e o tempo total que o servidor ficou ocupado (TTSOcupado). Para efetuar os cálculos necessários para obter os tempos citados anteriormente, a classe cria uma nova lista chamada “servidor” utilizando, como nós, a classe `NoMedidasServidor.java`.

### 3.6.3 As classes `TarefasFila.java` e `NoTarefasFila.java`

Essas classes são análogas às classes `MediadasServidor.java` e `NoMedidasServidor.java`. Elas, entretanto, calculam as métricas das tarefas do sistema utilizando dados como o tempo de chegada da tarefa em uma fila e o tempo de entrada, da mesma, em um servidor.

Para efetuar esses cálculos, a classe `TarefasFila.java` cria dois nós da lista “filas” (declarada na classe `Estatistica.java`), um nó com as métricas da tarefa relacionada a comunicação e outro com dados do processamento.

Em cada um desses nós são armazenadas informações como o identificador da tarefa, quantas vezes a tarefa passou por um servidor de comunicação, ou processamento, o tempo médio de fila da tarefa, seu tempo médio em um servidor e o tempo médio dela em um sistema.

Quando uma tarefa é alocada no motor, uma nova fila chamada “fila” é estabelecida com os atributos e métodos da classe `NoTarefasFila.java` e, através do método `addNoTarefasFila`, seus nós são criados. Nesses nós são colocadas informações da tarefa como o identificador, a carga, e o tempo de chegada na fila, no servidor ou no sistema e o tempo total gasto em uma fila, servidor ou sistema conforme a Figura 3.11.

```
//Entrada de uma tarefa em um no de comunicacao
case 4: System.out.printf("Antes\n");

//MARCO
estatistica.addNoTarefasFila( 0, tarefaAtual.getIdTarefa(), tarefaAtual.getTamanhoComTarefa(), tarefaAtual.getTempoChegadaFilaCom() );
```

Figura 3.11: Trecho de código retirado do motor de simulação mostrando a chamada do método `addNoTarefasFila`.

### 3.7 Considerações Finais

Este capítulo apresentou as principais etapas e estratégias para o desenvolvimento do projeto, inclusive utilizando elementos visuais e ilustrações para facilitar o entendimento. Esses passos têm como objetivo apresentar as características mais importantes na implementação das especificações do motor de simulação do iSPD.

Os testes efetuados para a validação do correto funcionamento do projeto serão apresentados no próximo capítulo. Estes testes serão seguidos por seus resultados e pela análise das informações obtidas.

## CAPÍTULO 4

### Testes

Neste capítulo são apresentados os testes realizados no motor de simulação. Inicialmente são feitos os testes dos geradores de números aleatórios, pois o correto funcionamento deles garante a não propagação de erros na simulação. Em seguida, os testes com um centro de serviço do tipo  $M/M/1$  são executados como forma de representar a rede de filas. Os testes dos outros tipos de centro de serviço são feitos de forma implícita no teste final que mostra a coleta das métricas de desempenho.

#### 4.1 Testes dos geradores de números aleatórios

Os testes dos geradores de números aleatórios consistiram em gerar 65356 números em cada uma das distribuições implementadas, separar esses números em classes de valores e plotar histogramas com as frequências em que cada classe foi gerada. As Figuras 4.1, 4.2 e 4.3 mostram, respectivamente, os histogramas para as distribuições *two-stage uniform*, normal e exponencial. Os valores utilizados como parâmetros para cada uma das distribuições estão listados na Tabela 4.1.

Tabela 4.1: Os parâmetros usados para gerar as distribuições de probabilidade.

<i>Two-stage Uniform</i>				Normal		Exponencial
Mínimo	Média	Máximo	Probabilidade	Média	Desvio Padrão	Média
0,0	50,0	100,0	0,20	50,0	8,0	20,0

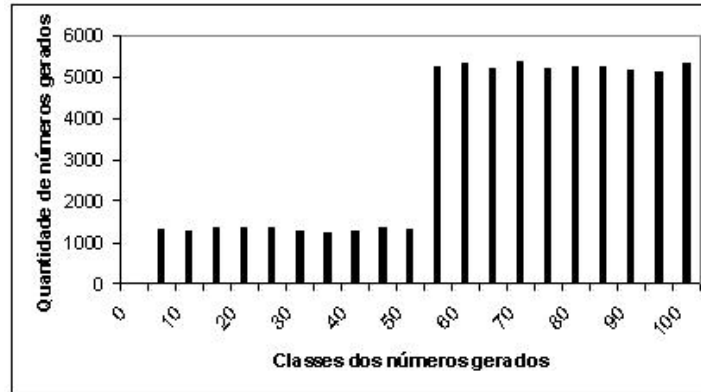
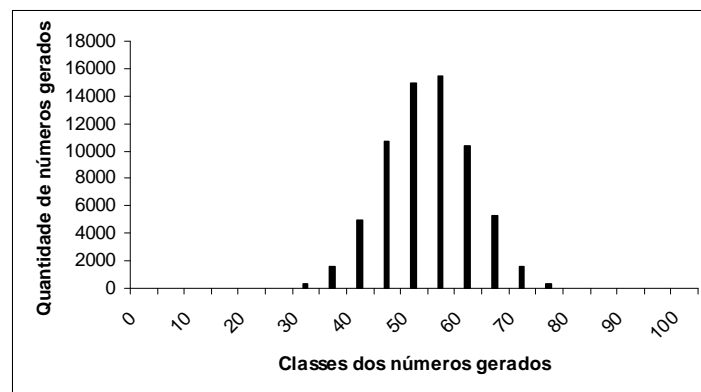
Figura 4.1: Histograma da distribuição *two-stage uniform*.

Figura 4.2: Histograma da distribuição normal.

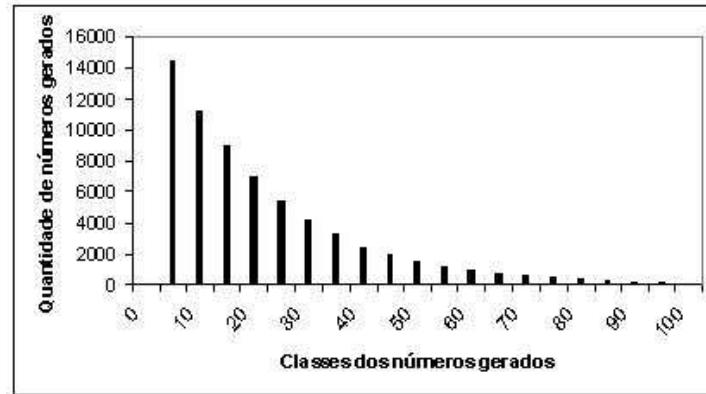


Figura 4.3: Histograma da distribuição exponencial.

Através dos histogramas apresentados, é possível concluir que, mesmo com as dificuldades inerentes à geração de números apresentadas em [Zafalon et al., (2006)], os geradores de números aleatórios seguem o comportamento esperado.

## 4.2 Testes do sistema M/M/1

Os testes com um sistema M/M/1 foram realizados como forma de representação dos modelos de filas implementados. A Tabela 4.2 mostra as características do sistema simulado e os resultados teóricos (esperados) e os práticos (obtidos) para ele. Pelas características de um processo de simulação, o teste consistiu em simular dez vezes o sistema, calcular as médias das variáveis de interesse entre os resultados dessas simulações e finalmente comparar essas médias com os resultados teóricos. Para se obter um valor de  $\lambda$  (taxa de chegada das tarefas na fila) igual a 0,2, fez-se o tempo médio entre as chegadas das tarefas ser de 5 segundos, o que resulta numa média de 0,2 tarefas a cada segundo. Já para se obter o valor de  $\mu$  (taxa de atendimento) igual a 0,5, fez-se o tamanho médio das tarefas ser de 5 MFlops e o poder computacional do servidor ser de 10 MFlops/s o que resulta em uma taxa de atendimento média de 0,5 s.

Excluído: à

Tabela 4.2: As características e os resultados do sistema M/M/1.

Características do sistema:				
$\lambda=$	0,2			
$\mu=$	0,5			
Resultados teóricos esperados para as variáveis de interesse:				
TF	TS	NF	NS	
1,333333	3,333333	0,266667	0,666667	
Resultados práticos obtidos para as variáveis de interesse:				
Medição	TF	TS	NF	NS
1	1,637582	2,391946	0,327516	0,47839
2	1,775538	2,488804	0,355108	0,49776
3	1,934365	2,668114	0,386873	0,53362
4	1,578288	2,302504	0,315658	0,46050
5	1,890351	2,609780	0,378070	0,52196
6	0,913478	1,664639	0,182696	0,33293
7	0,656252	1,380063	0,131250	0,27601
8	1,450484	2,198786	0,290097	0,43976
9	1,444755	2,159054	0,288951	0,43181
10	2,179829	2,911281	0,435966	0,58226
Médias:	1,546092	2,277497	0,309218	0,45550

Considerando-se mais uma vez as características de um processo de simulação e o tamanho da amostra, a proximidade entre os valores esperados e os obtidos, torna os resultados obtidos pelo simulador bastante satisfatórios.

### 4.3 Teste Final

Para o teste final, foi elaborado um modelo com todos os ícones que o simulador disponibiliza para o usuário. Foi criado um *cluster* com 5 máquinas e com poder computacional de 1000000 MFlop/s, 7 máquinas com poder computacional de 111 MFlop/s, um link de Internet com largura de banda igual a 100 Mbps, e conexões de rede internas de 10000 Mbps. A carga computacional das tarefas é de no mínimo 0 MFlop e máximo 10000 MFlop, enquanto que a carga de comunicação tem valor



mínimo de 0 MFlop e máximo de 10000 MFlop também. Esse modelo pode ser visto na Figura 4.4.

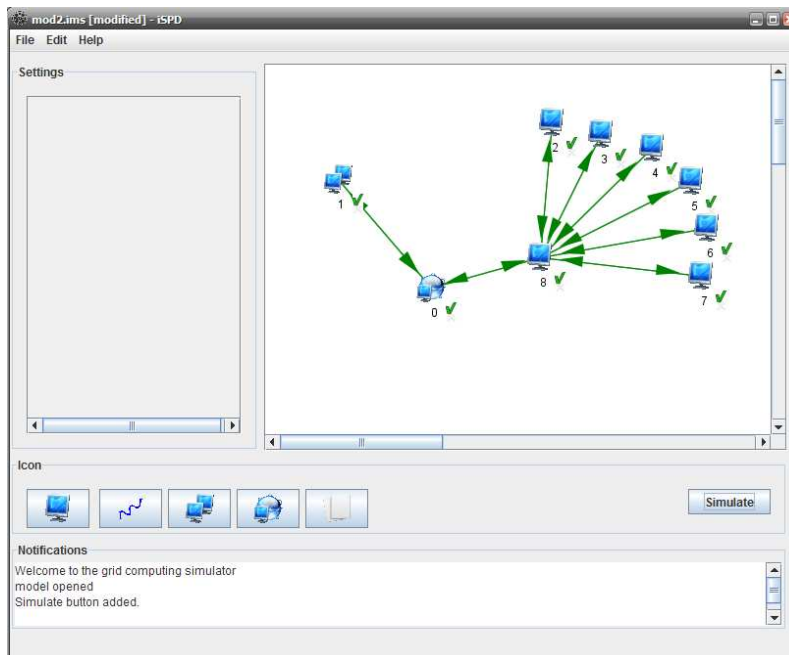


Figura 4.4: Plataforma criada para o teste final do iSPD.

Foram realizados dois testes, um com 2 tarefas e outro com 300 tarefas. Os respectivos resultados desses testes podem ser vistos nas Figuras 4.5 e 4.6.

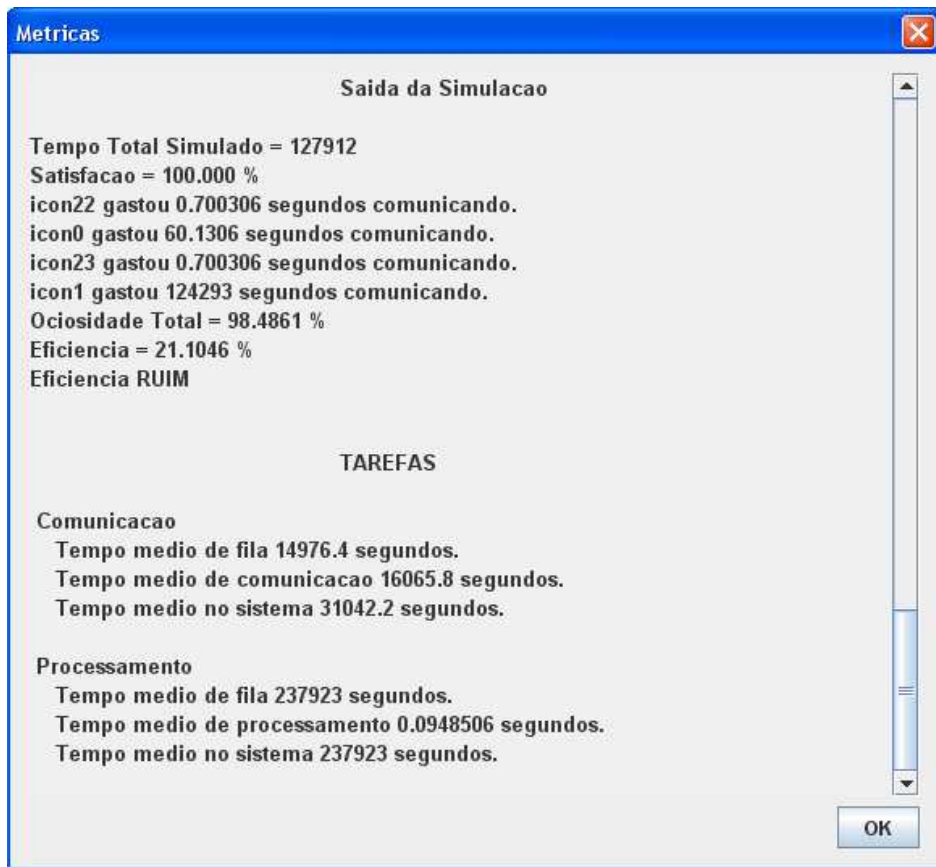


Figura 4.5: Resultado do teste com 2 tarefas.

Para o teste seguinte, a carga computacional das tarefas é de no mínimo 0 MFlop e máximo 10MFlop, enquanto que a carga de comunicação tem valor mínimo de 0 MFlop e máximo de 10 MFlop também.

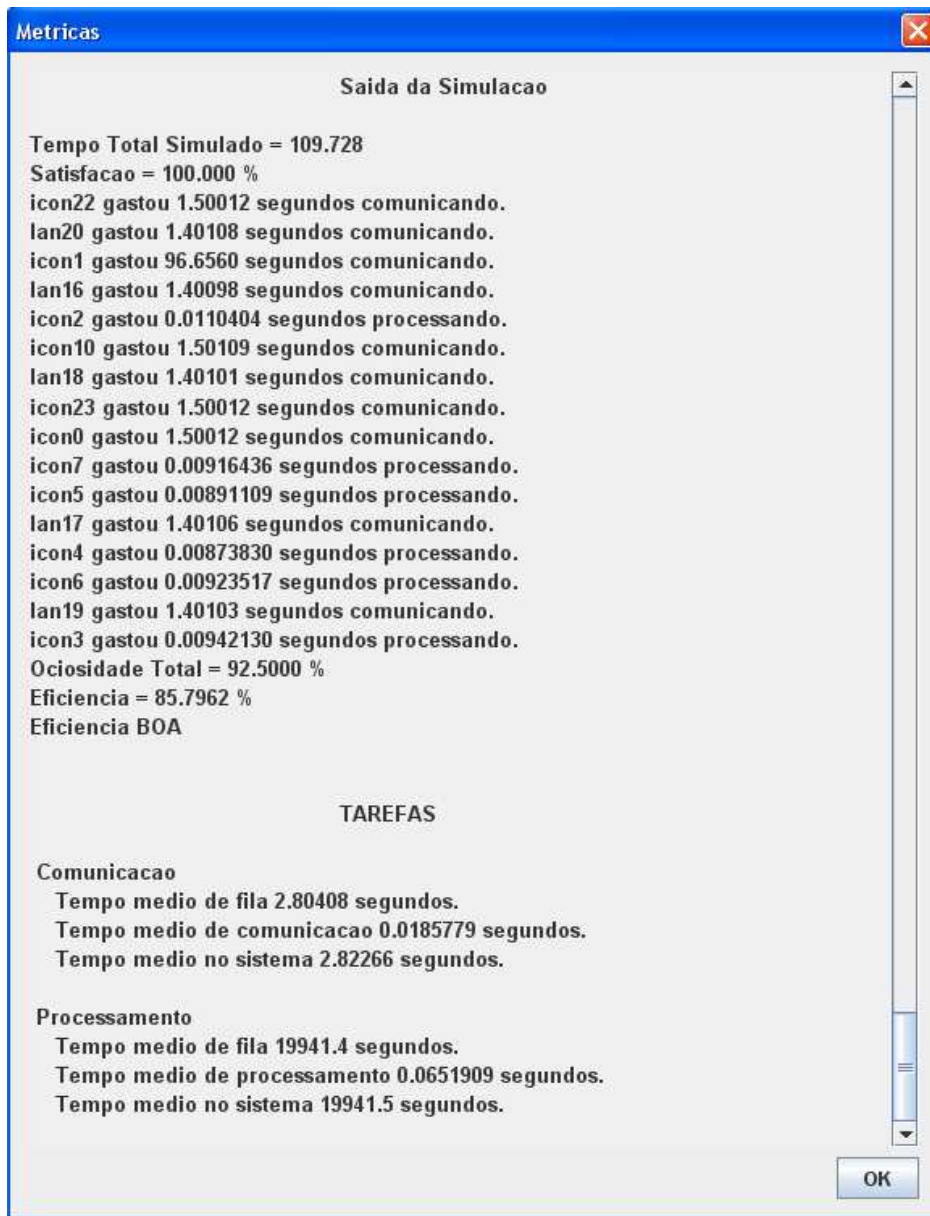


Figura 4.6: Resultado do teste com 300 tarefas.

Após os testes realizados pudemos notar o esperado. A métrica de ociosidade diminui, já que o número de tarefas aumentou. O tempo médio de fila, processamento e sistema diminuiu, pois as tarefas do conjunto de testes eram menores.

#### **4.4 Considerações Finais**

Nesta seção foram exibidos alguns testes realizados com os geradores de números aleatórios, o que se justifica por garantir a não ocorrência de erros nesta etapa da simulação e, mais ainda, a não propagação de erros para as fases posteriores.

Outro teste parcial realizado foi o do sistema M/M/1. Esse sistema foi escolhido por ser bastante usado no motor de simulação e ser um bom representante dos modelos de filas. Além disso, os outros modelos foram automaticamente testados durante o teste final da ferramenta que, por sua vez, foi essencial para as conclusões expostas no próximo capítulo.

Por sua vez, o teste final confirmou o cumprimento de todas as funcionalidades previstas para o simulador, desde os processos internos do simulador até a obtenção de métricas para avaliação de desempenho dos sistemas simulados.

## CAPÍTULO 5

### Conclusões

Este capítulo irá apresentar as principais contribuições obtidas com o desenvolvimento deste trabalho, as dificuldades encontradas, conclusões e propostas para trabalhos futuros.

Além disso, este projeto foi publicado e apresentado, em diferentes fases de elaboração, em dois eventos do estado de São Paulo a fim de expor os resultados obtidos com o desenvolvimento deste trabalho, são eles:

- XXI Congresso de Iniciação Científica da Unesp (CIC Unesp) de São José do Rio Preto - SP [Aoqui *et al.*, (2009)] [Guerra *et al.*, (2009)] [Oliveira *et al.*, (2009)];
- I Escola Regional de Alto Desempenho de São Paulo (ERAD-SP 2010) [Aoqui *et al.*, (2010)] [Guerra *et al.*, (2010)] [Oliveira *et al.*, (2010)].

#### 5.1 Contribuições

A elaboração deste trabalho permitiu o desenvolvimento de um motor de simulação concomitantemente robusto e modular, sendo que essas características são imprescindíveis para seu aprimoramento contínuo, ou seja, para que novas

Excluído: é

Excluído: !

funcionalidades sejam adicionadas a ele. É interessante destacar também seu bom desempenho e o relativo baixo consumo de recursos computacionais que ele demanda para realizar as simulações, muito disso devido à linguagem Java e seus recursos, como por exemplo, a limpeza periódica de memória. Além disso, quando aliado à sua interface gráfica, ele possibilita a construção de modelos de simulação de grades computacionais de forma fácil e intuitiva mesmo por usuários sem muita familiaridade com esses sistemas.

## 5.2 Dificuldades encontradas

As principais dificuldades encontradas durante o desenvolvimento deste trabalho foram o entendimento dos processos de simulação, elaborar o método adequado de implementação do algoritmo *Event Scheduling/Time Advance* para ele simular corretamente o funcionamento e os eventos de uma grade computacional e o aprendizado da linguagem Java para implementar todo o sistema.

## 5.3 Conclusões

O estudo dos principais simuladores existentes contribuiu significativamente com a identificação de requisitos funcionais e métricas de desempenho necessárias para um ambiente flexível e simples de usar. Dessa forma, a partir do levantamento de requisitos, do estudo sobre simulação de sistemas e geração de números aleatórios, teoria das filas e linguagem UML, foi possível especificar, propor e implementar o motor de simulação.

No decorrer do trabalho a importância tanto do correto entendimento das funcionalidades do sistema a ser simulado quanto da adequação dessas funcionalidades às técnicas de simulação ficaram bastante evidentes. Por sua vez, os resultados obtidos através dos testes realizados mostraram que esse nível de entendimento foi obtido, culminando na implementação eficiente de um robusto motor de simulação e da geração das métricas de saída. Neste ponto é interessante

destacar o quanto a escolha pela linguagem Java para implementação do projeto se mostrou acertada, já que ela possui recursos essenciais para a codificação do projeto.

Por fim, ressalta-se o fato de que todas as atividades citadas no cronograma da proposta de projeto final foram cumpridas fielmente.

## 5.4 Propostas para Trabalhos Futuros

Para trabalhos futuros, espera-se que o motor de simulação seja capaz de executar outros algoritmos tanto de escalonamento quanto de roteamento, possibilitando, ainda que o usuário seja capaz de implementar seus próprios algoritmos, e assim a ferramenta estará completamente apta para avaliar uma gama ainda maior de sistemas, incluindo casos extremamente peculiares. Além disso, é possível melhorar a fidelidade da simulação das ligações de rede do tipo “Internet” com a implementação de novas topologias de rede.

É possível ainda trabalhar em adequações para que a ferramenta atenda simulações de ambientes virtualizados e de entrega de *hardware* sob demanda. Com isso, o ambiente de simulação poderá avaliar sistemas de *cloud computing*.

Trabalhos futuros também devem criar interfaces gráficas para a apresentação das métricas de desempenho definidas. Isso permitirá ao usuário ter uma melhor visualização sobre o desempenho dos vários algoritmos de escalonamento para saber qual dos algoritmos se comporta melhor na grade simulada.

Excluído: dendo-se

Excluído: possibilitar

## REFERÊNCIAS BIBLIOGRÁFICAS

- [Aoqui e Guerra, (2010)] AOQUI, V., GUERRA, A. I., **Plataforma de Simulação de Grades Computacionais: Interface Icônica e Intepretador de Modelos Externos**. Monografia (Graduação em Ciência da Computação) - Instituto de Biociências, Letras e Ciências Exatas – IBILCE – Departamento de Ciências de Computação e Estatística – DCCE – UNESP, 2010;
- [Aoqui *et al.*, (2009)] AOQUI, V., BRAIT, T. P., GUERRA, A. I., OLIVEIRA, P. H. M. A. e LOBATO, R. S. **Interpretador de Modelos Multiformato Para Plataforma de Simulação de Grades Computacionais**. In: XXI Congresso de Iniciação Científica da Unesp. Anais do XXI Congresso de Iniciação Científica da UNESP. v. CD-ROM. p. 1-4. São José do Rio Preto: Instituto de Biociências, Letras e Ciências Exatas – IBILCE-UNESP, 2009;
- [Aoqui *et al.*, (2010)] AOQUI, V., GUERRA, A. I., GARCIA, M. A. B. A., OLIVEIRA, P. H. M. A., LOBATO, R. S. e MANACERO JR, A. **Interpretador de Modelos Externos Para Simulador de Grades Computacionais**. In: I Escola Regional de Alto Desempenho de São Paulo. v. CD-ROM. p. 1-2. São Paulo: Universidade Presbiteriana Mackenzie, 2010;
- [Baliere, (2005)] BALIEIRO, M. O. S., Protocolo **Conservativo CMB para Simulação Distribuída**, Dissertação de Mestrado, DCT-UFMS, 2005;
- [Banks et al., (2001)] BANKS, J. CARSON, J. S., NICOL, D. M., NELSON B. L., **Discrete-Event System Simulation**, 3ª edição, Prentice-Hall, International Series In Industrial and Systems Engineering, 2001;
- [Bell et al, (2003)] BELL, W., CAMERON, D.G; CAPOZZA, L., MILLAR, A.P.; Stockinger, K.; Zini, F. **Optorsim: a grid simulator for studying dynamic data replication strategies**. International Journal of High Performance Computing Applications, p. 403416, 2003. Disponível em: <<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.8027>>;
- [Boaventura et al., (2009)] BOAVENTURA NETTO, P. O., JURKIEWICZ, S. **Grafos: Introdução e Prática**, Editora Blucher, São Paulo, 2009;
- [Bricks Systems, (1999)] **Bricks Systems**. Disponível em <<http://ninf.apgrid.org/bricks/contents/hpdc99.html#overview>>. Acesso em: 08 Nov. 2009;
- [Buyya et al., (2002)] BUYYA, R., MURSHED, M. **Gridsim: A toolkit for the**



- modeling and simulation of distributed resource management and scheduling for grid computing.** Concurrency and Computation: Practice and Experience (CCPE), v.14, p.1175-1220, Novembro-Dezembro 2002. Disponível em: <<http://www.gridbus.org/papers/gridsim.pdf>>;
- [Chiacchio, (2005)] CHIACCHIO, R. L. C. **Biblioteca orientada a objetos para simulação de redes de filas**, 71p. Monografia (Graduação em Ciência da Computação) - Instituto de Biociências, Letras e Ciências Exatas – IBILCE – Departamento de Ciências de Computação e Estatística – DCCE – UNESP, 2005;
- [Deitel et al., (2005)] DEITEL, H. M. & DEITEL, P. J. **Java - Como Programar**. 6ª Edição. Editora Pearson, 2005;
- [Dumitrescu et al., (2005)] DUMITRESCU, C. L., FOSTER, I. **Gangsim: a simulator for grid scheduling studies**. In: CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid05). Washington, DC, Estados Unidos: IEEE Computer Society, 2005. v. 2, p.1151-1158. ISBN 0-7803-9074-1. Disponível em: <<http://people.cs.uchicago.edu/cldumitr/docs/GangSim.pdf>>;
- [EU DataGrid Project, (2001)] EU DataGrid Project, **The DataGrid architecture**. Technical Report DataGrid-12-D12.4-333671-3-0, C E R N, Geneva, Switzerland, 2001;
- [Falavinha Jr, (2009)] FALAVINHA Jr., J. N., **Escalonamento de tarefas em Sistemas Distribuídos baseado no Conceito de Prioridade Distribuída**, Faculdade de Engenharia de Ilha Solteira -FEIS- UNESP, 2009;
- [Guedes, (2009)] GUEDES, G. T. A., **UML 2: Uma abordagem prática**, Editora Novatec, São Paulo, 2009;
- [Guerra *et al.*, (2009)] GUERRA, A. I., AOQUI, V., GARCIA, M. A. B. A., OLIVEIRA, P. H. M. A. e LOBATO, R. S. **Projeto de Interface Icônica Para Simulador de Grades Computacionais**. In: XXI Congresso de Iniciação Científica da Unesp. Anais do XXI Congresso de Iniciação Científica da UNESP. v. CD-ROM. p. 1-4. São José do Rio Preto: Instituto de Biociências, Letras e Ciências Exatas – IBILCE-UNESP, 2009;
- [Guerra *et al.*, (2010)] GUERRA, A. I., GARCIA, M. A. B. A., OLIVEIRA, P. H. M. A., AOQUI, V., LOBATO, R. S. e MANACERO JR, A. **Plataforma de Simulação de Grades Computacionais: Interface Icônica**. In: I Escola Regional de Alto Desempenho de São Paulo. v. CD-ROM. p. 1-2. São Paulo: Universidade Presbiteriana Mackenzie, 2010;
- [GRAS, (1999)] **Module GRAS do SimGrid**. Disponível em: <[http://simgrid.gforge.inria.fr/doc/group\\_\\_GRAS\\_\\_API.html](http://simgrid.gforge.inria.fr/doc/group__GRAS__API.html)>. Acesso em 23 Jul. 2009;

- [GridSim, (2007)] The GridSim Toolkit. Disponível em <<http://www.cct.lsu.edu/~dsk/eScience2007Posters/sulistio.html>>. Acesso em 10 Nov. 2009
- [Hillier, (1988)] HILLIER, F. S., LIEBERMAN, G. J., **Introdução a pesquisa operacional**, 1ª edição, tradução de Helena L. Lemos, editora da USP, São Paulo, 1988;
- [Jain et al., (1991)] Jain R. **The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling**. John Wiley & Sons, 2nd edition edition, 1991;
- [Kleinrock, (1975)] KLEINROCK L. **Queueing Systems**, 1ª Edição. Wiley-Interscience, 1975;
- [Lublin, (2003)] LUBLIN U., FEITELSON, D. G. **The workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs**. Journal of Parallel and Distributed Computing, Volume 63, fascículo 11, páginas 1105-1122, Novembro de 2003;
- [Oliveira, (2008)] OLIVEIRA, L. J. **Comparação de ferramentas de simulação de grades computacionais**. Relatório Técnico - Instituto de Biociências, Letras e Ciências Exatas – IBILCE – Departamento de Ciências de Computação e Estatística – DCCE – UNESP, 2008. Disponível em: <http://www.dcce.ibilce.unesp.br/spd/pubs/gridsimulationtools.pdf>;
- [Oliveira *et al.*, (2009)] OLIVEIRA, P. H. M. A., AOQUI, V., GUERRA, A. I., GARCIA, M. A. B. A. e MANACERO JR, A. **Especificação de Um Motor de Simulação de Grades Computacionais**. In: XXI Congresso de Iniciação Científica da Unesp. Anais do XXI Congresso de Iniciação Científica da UNESP. v. CD-ROM. p. 1-4. São José do Rio Preto: Instituto de Biociências, Letras e Ciências Exatas – IBILCE-UNESP, 2009;
- [Oliveira *et al.*, (2010)] OLIVEIRA, P. H. M. A., GUERRA, A. I., GARCIA, M. A. B. A., AOQUI, V., MANACERO JR, A. e LOBATO, R. S. **O Motor de Uma Plataforma de Simulação de Grades Computacionais**. In: I Escola Regional de Alto Desempenho de São Paulo. v. CD-ROM. p. 1-2. São Paulo: Universidade Presbiteriana Mackenzie, 2010;
- [Machado, (2008)] MACHADO, D. J. **algSim – Linguagem algorítmica para simulação de redes de filas**, 82p. Monografia (Graduação em Ciência da Computação) - Instituto de Biociências, Letras e Ciências Exatas – IBILCE – Departamento de Ciências de Computação e Estatística – DCCE – UNESP, 2008;
- [MSG, (1999)] **Module MSG do SimGrid**. Disponível em: <[http://simgrid.gforge.inria.fr/doc/group\\_\\_MSG\\_\\_API.html](http://simgrid.gforge.inria.fr/doc/group__MSG__API.html)>. Acesso em 23 Jul. 2009;

- [Tanenbaum et al.,(2007)] TANENBAUM, A.S., VAN STEEN, M., **Sistemas distribuídos: princípios e paradigmas**, 2ª edição, tradução de Arlete Simille Marques, Prentice-Hall, São Paulo, 2007;
- [Santana et al, (1994)] SANTANA, R. H. C., SANTANA, M. J., ORLANDI, R. C. G. S., SPOLON, R., Júnior, N. C., **Técnicas para Avaliação de Desempenho de Sistemas Computacionais**, Notas didáticas do ICMC, ICMC – USP, 1994;
- [Shannon, (1975)] SHANNON, R.E. **Systems Simulation: The Art and Science**. Prentice-Hall. Englewood Cliffs, N.J. 1975;
- [SimGrid Project, (1999)] **SimGrid Project**. Disponível em <<http://simgrid.gforge.inria.fr/doc/index.html>>. Acesso em: 08 Nov. 2009;
- [Soares, (1990)] SOARES, L. F. G., **Modelagem e Simulação Discreta de Sistemas**, VII Escola de Computação, 1990;
- [SMPI, (1999)] **Module SMPI do SimGrid**. Disponível em: <[http://simgrid.gforge.inria.fr/doc/group\\_\\_SMPI\\_API.html](http://simgrid.gforge.inria.fr/doc/group__SMPI_API.html)>. Acesso em 23 Jul. 2009;
- [SimDag, (1999)] **Module SimDag do SimGrid**. Disponível em: <[http://simgrid.gforge.inria.fr/doc/group\\_\\_SD\\_API.html](http://simgrid.gforge.inria.fr/doc/group__SD_API.html)>. Acesso em 23 Jul. 2009;
- [SURF, (1999)] **Module SURF do SimGrid**. Disponível em: <[http://simgrid.gforge.inria.fr/doc/group\\_\\_SURF\\_API.html](http://simgrid.gforge.inria.fr/doc/group__SURF_API.html)>. Acesso em 23 Jul. 2009;
- [Zafalon et al, (2006)] Zafalon, G. F. D., Manacero Jr, A. **Construção de geradores independentes de números aleatórios para diferentes distribuições probabilísticas**. In: Anais do Workshop em Computação e Aplicações - WCOMPA, Campo Grande, CD-ROM, p. 16-21, 2006;

## Apêndice A – Diagramas de Classe UML

Este apêndice é reservado para a apresentação dos diagramas de classe UML dos componentes do motor de simulação, uma vez que os mesmos tomariam muito espaço no capítulo em que se descreve o projeto.

A Figura A.1 ilustra o diagrama de classes do pacote RedesDeFilas.java, cujas classes suportam a montagem da infraestrutura computacional da grade a ser simulada.

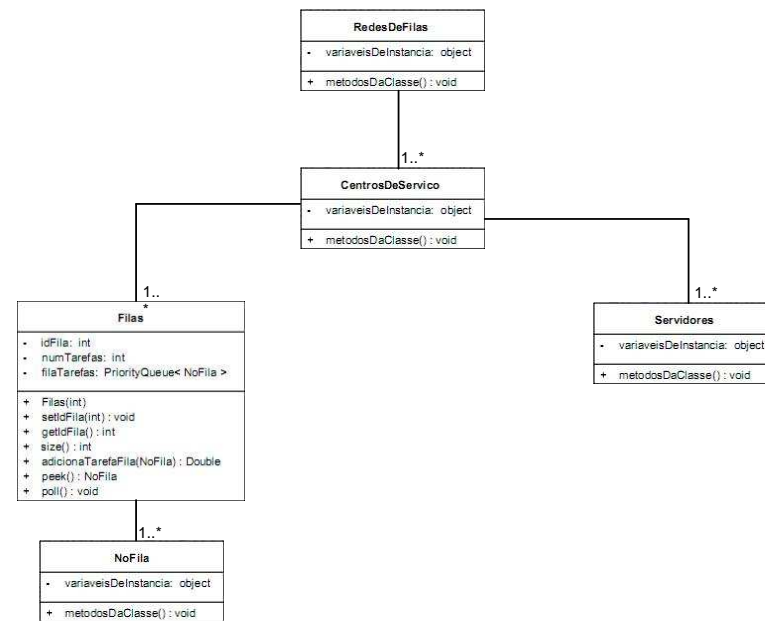


Figura A.1: Diagrama de classes UML do pacote RedesDeFilas.java.

Já os pacotes modelados pelo diagrama de classes da Figura A.2 podem ser considerados o núcleo do motor de simulação e realizam tanto o controle quanto a execução da simulação em si.

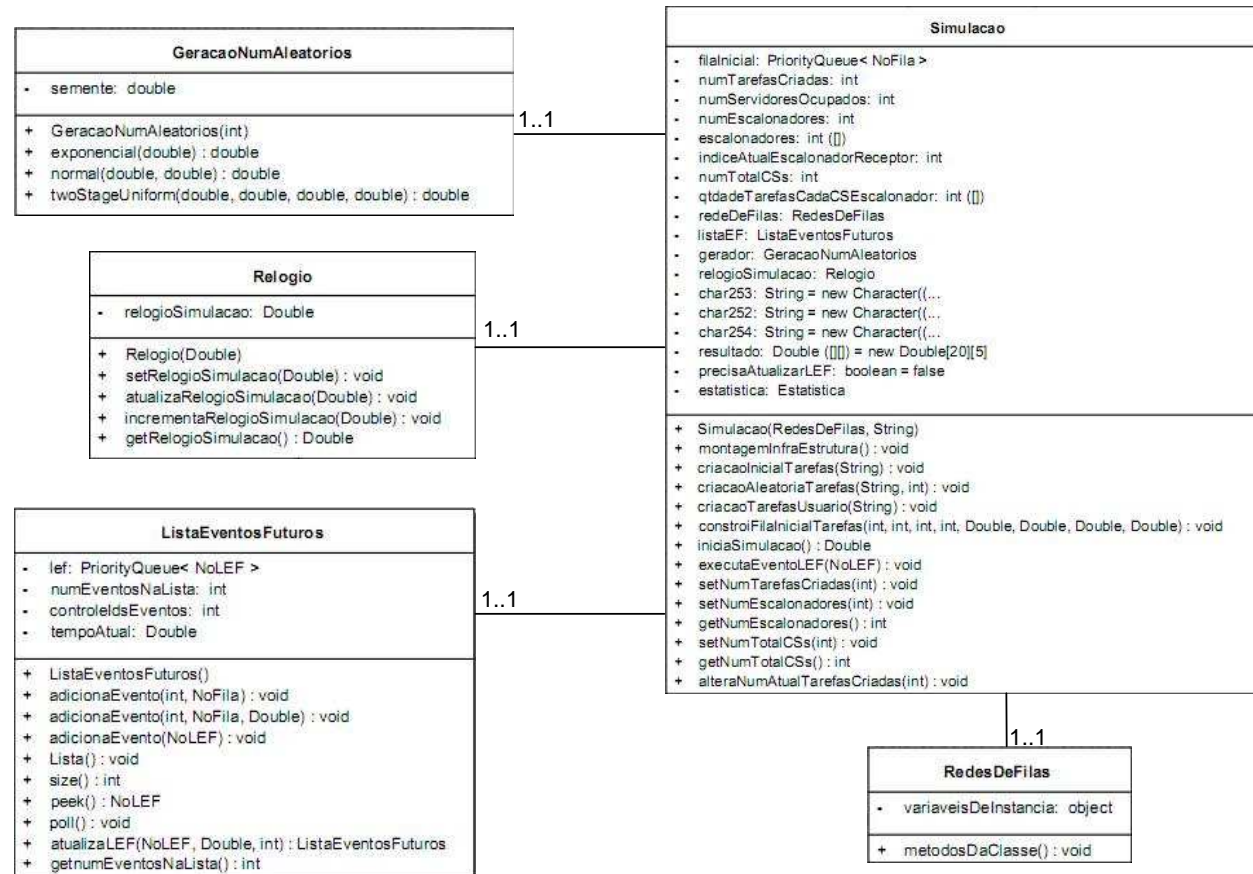


Figura A.2: Diagrama de classes UML dos pacotes Simulacao.java e GeracaoNumAleatorios.java.

A Figura A.3 mostra a relação existente entre as classes ListaEventosFuturos.java e NoLEF.java. Essas classes são parte integrante do pacote Simulacao.java.

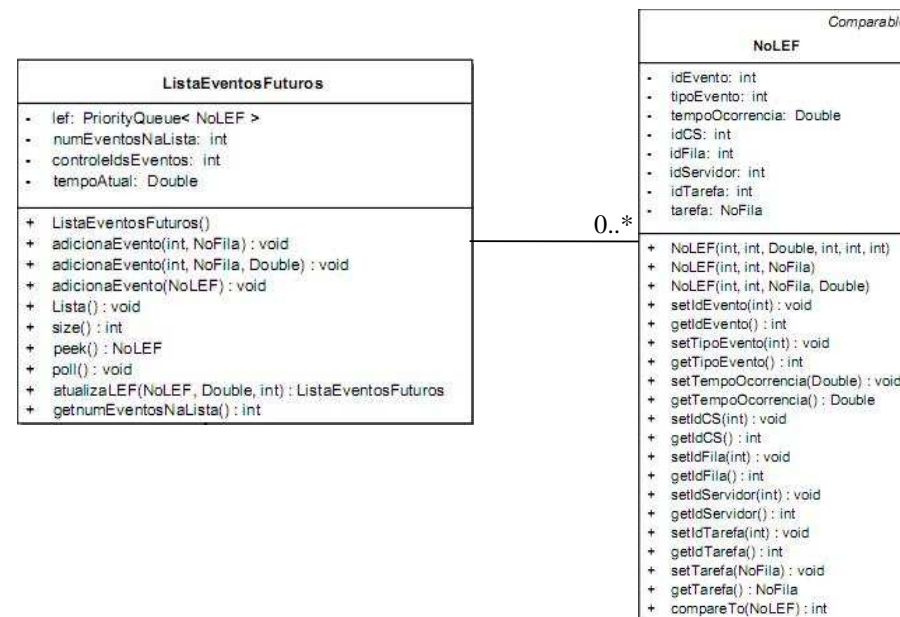


Figura A.3: Diagrama de classes das classes ListaEventosFuturos.java e NoLEF.java.

Excluído: r

As Figuras A.4, A.5, A.6 e A.7 são, respectivamente, a representação detalhada das classes `RedesDeFilas.java`, `CentrosDeServico.java`, `Servidores.java` e `NoFila.java`. Foi necessário fazer esse detalhamento, pois as classes são muito extensas e foram reduzidas nas figuras anteriores.

Excluído: i



Figura A.4: Diagrama de classes UML detalhado da classe `RedesDeFilas.java`.



Figura A.5: Diagrama de classes UML detalhado da classe CentrosDeServico.java.



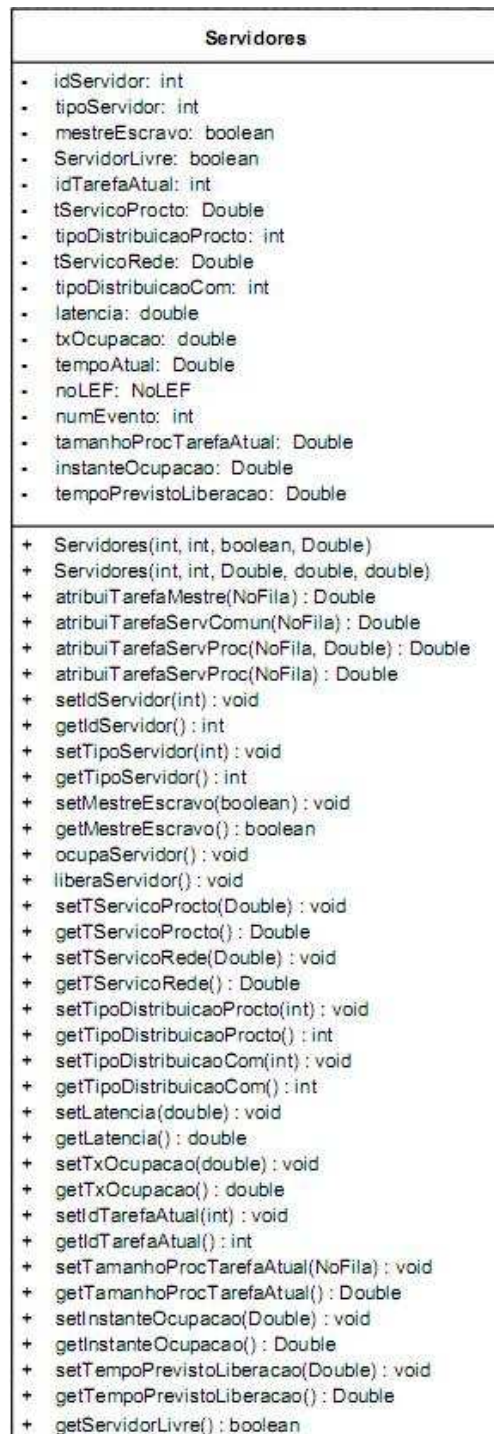


Figura A.6: Diagrama de classes UML detalhado da classe Servidores.java.



Figura A.7: Diagrama de classes UML detalhado da classe NoFila.java.

A Figura A.8 ilustra o diagrama de classes dos pacotes Estatistica.java e CaixaTextoEstatistica.java, cujas classes suportam a montagem da infraestrutura computacional da grade a ser simulada para o armazenamento das métricas necessárias.

Devido à extensão das classes TarefasFila.java, NoTarefasFila.java, MetricaCS.java, MedidasServidor.java e NoMedidasServidor.java, mais adiante há a representação e interação de cada uma delas com o diagrama Estatistica.java.

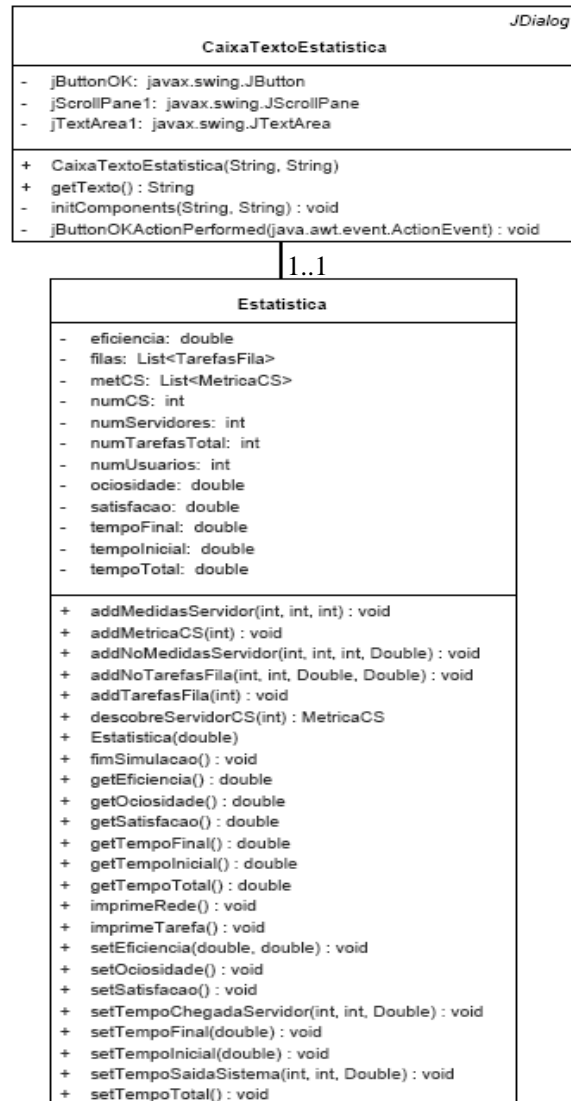


Figura A.8: Diagrama de classes UML detalhado das classes CaixaTextoEstatistica.java e Estatistica.java.

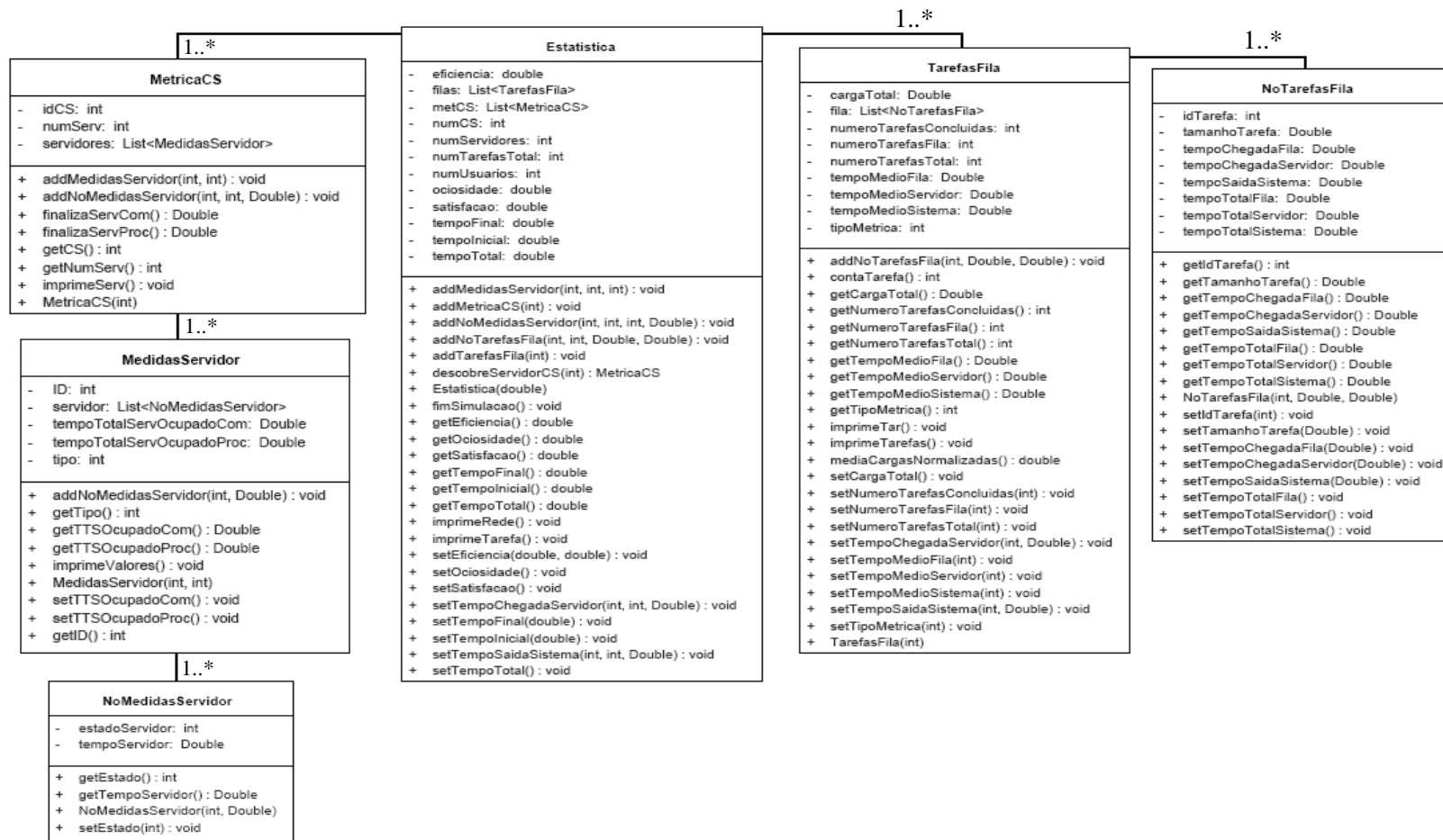


Figura A.9: Diagrama de classes UML detalhado das classes Estatistica.java, MetricaCS.java, MedidasServidor.java, NoMedidasServidor.java, TarefasFila.java e NoTarefasFila.java.