AURÉLIO AKIRA MELLO MATSUI

**UM AMBIENTE DE DESENVOLVIMENTO E EXECUÇÃO DE
APLICAÇÕES GRID ESCRITAS TOTALMENTE EM JAVA**

Dissertação apresentada à Escola
Politécnica da Universidade de São Paulo
para obtenção do grau de Mestre em
Engenharia Elétrica.

São Paulo
2006

AURÉLIO AKIRA MELLO MATSUI

**UM AMBIENTE DE DESENVOLVIMENTO E EXECUÇÃO DE
APLICAÇÕES GRID ESCRITAS TOTALMENTE EM JAVA**

Dissertação apresentada à Escola
Politécnica da Universidade de São Paulo
para obtenção do grau de Mestre em
Engenharia Elétrica.

Área de concentração: Sistemas Digitais

Orientadora: Profa. Dra. Líria Matsumoto
Sato

São Paulo
2006

Este exemplar foi revisado e alterado em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, 04 de Agosto de 2006.

Assinatura do autor _____

Assinatura do orientador _____

FICHA CATALOGRÁFICA

Matsui, Aurélio Akira Mello

Um ambiente de desenvolvimento e execução de aplicações grid escritas totalmente em Java / Aurélio Akira Mello Matsui – ed. rev. – São Paulo, 2006. 167p.

Dissertação (Mestrado) – Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1. Sistemas distribuídos 2. Arquitetura de sistemas distribuídos I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t

ALGUNS DIREITOS RESERVADOS

Este texto está licenciado sob uma licença Creative Commons. Esta licença permite o uso não comercial deste trabalho, desde que os créditos sejam fornecidos. Para mais informações sobre esta licença, visite <http://creativecommons.org/licenses/by-nc/1.0/>.

SOME RIGHTS RESERVED

This text is licensed under a Creative Commons license. This license permits non-commercial use of this work, so long as attribution is given. For more information about the license, visit <http://creativecommons.org/licenses/by-nc/1.0/>.

À Nê, pela sua recuperação.

AGRADECIMENTOS

À Profa. Dra. Líria Matsumoto Sato, pela orientação, idéias e incentivo durante todo este trabalho.

Ao Prof. Dr. Edson Toshimi Midorikawa e ao Prof. Dr. Mario Donato Marino pela infra-estrutura do laboratório.

A todos os colegas de laboratório, pela companhia durante o trabalho.

À Janaína, pela infinita paciência e companheirismo durante o mestrado, e pelos seus conhecimentos lingüísticos e seu empenho durante a escrita do texto.

RESUMO

Este trabalho de mestrado propõe uma plataforma para o desenvolvimento e a execução de aplicações distribuídas em grids computacionais. Tal plataforma visa simplificar o desenvolvimento dessas aplicações, ao mesmo tempo em que as torna capazes de usar recursos compartilhados em um grid. No esquema proposto, o ambiente de desenvolvimento do grid tem estreito relacionamento com o próprio grid, na medida em que o grid distribui a execução dos programas gerados pelo ambiente de desenvolvimento de forma automática. Além de definir a forma de desenvolver as aplicações, o presente trabalho de mestrado também propõe uma infra-estrutura para grids composta por um sistema de troca de mensagens, um escalonador baseado na teoria fuzzy e um mecanismo simples de persistência distribuída.

ABSTRACT

This thesis proposes a platform for computational grids distributed applications development and execution. The purpose of this platform is to simplify the development of such applications while delivering access to shared resources in a grid environment. In the proposed schema, the grid development environment has a close link with the grid itself as the grid distributes the execution of programs created by the development environment in an automated fashion. Besides presenting a way to develop applications, this thesis also proposes a grid infrastructure composed by a message system, a job scheduler based on fuzzy theory and a simple mechanism of distributed persistence.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

1	Introdução	1
1.1	Objetivo.....	2
1.2	Motivação.....	4
1.3	Trabalhos relacionados.....	6
1.4	Metodologia	7
1.5	Organização do Texto	8
2	Conceitos.....	9
2.1	Organizações Virtuais	9
2.2	Grid	12
2.3	Serviços Grid.....	13
2.3.1	Notificações	14
2.3.2	Reserva de recursos.....	14
2.3.3	Controle de ciclo de vida de serviços grid	15
2.3.4	Contratos entre os componentes de um grid	15
2.3.5	OGSA e OGSF.....	16
2.4	Orientação a aspectos	17
2.4.1	Exemplo de aspecto	20
2.4.2	Outro exemplo mais complexo	25
2.5	Java EE.....	30
2.6	Teoria Fuzzy	33
2.6.1	Variáveis Lingüísticas.....	34
2.6.2	Regras Fuzzy	34
2.6.3	Partições Fuzzy	35
2.6.4	Conjuntos Fuzzy	37
2.6.5	Operações com conjuntos fuzzy.....	39
3	Ambiente de desenvolvimento e execução de grids	43
3.1	Ambiente de desenvolvimento	45
3.1.1	Modelo de programação.....	45
3.1.2	Ferramentas de Desenvolvimento	60
3.2	Ambiente de execução	63
3.2.1	Clientes / nós do grid	63
3.2.2	Servidor de aplicações	64
3.2.3	Sistema de Troca de Mensagens	64
3.2.4	Banco de dados distribuído	66
3.2.5	Tolerância a falhas	73
3.2.6	Monitoramento de recursos.....	74
3.2.7	Predição de carga	76
3.2.8	Exemplo	82
3.3	Balanceamento de carga usando a teoria Fuzzy.....	85

3.3.1	Controle fuzzy	85
3.3.2	Modelo de carga usado	87
3.3.3	Algoritmo Fuzzy para escalonamento.....	87
3.3.4	Variáveis Linguísticas de Entrada.....	90
4	Implementação e resultados	97
4.1	Implementação	97
4.1.1	Orientação a Aspectos.....	97
4.1.2	<i>Classloader</i> para grid	102
4.1.3	Uso de POJO	104
4.1.4	GridControlerMDB	105
4.1.5	Sistema de troca de mensagens.....	107
4.2	Resultados obtidos e análises	108
4.2.1	Testes unitários	109
4.2.2	Teste de estresse.....	110
4.2.3	Testes de performance.....	110
5	Conclusões	116
5.1	Conclusões do trabalho	116
5.2	Contribuições do trabalho	117
5.3	Trabalhos futuros	118
5.3.1	Escalonador	118
5.3.2	Sistema de troca de mensagens.....	118
5.3.3	Banco de dados	119
5.3.4	Atualização das ferramentas acompanhando as mudanças que eventualmente ocorrerão na linguagem AspectJ.....	121
5.3.5	Segurança	121
5.3.6	Compatibilidade com a OGSA	122
5.3.7	Mecanismos de notificação e reserva de recursos.....	122
5.3.8	Otimização do uso da memória.....	122
5.3.9	Cache do <i>classloader</i>	123
5.3.10	Codificação	124
5.3.11	Objetos remotos	124
	Referências Bibliográficas	125
	Apêndice A - Mecanismo para a criação de protocolos definidos pelo usuário	129
	Apêndice B – Manual de programação sucinto	134

LISTA DE FIGURAS

Figura 1 – Estrutura de uma organização virtual	10
Figura 2 – Estrutura física da mesma organização virtual	11
Figura 3 - Fases na compilação de um programa escrito usando orientação a aspectos	18
Figura 4 – Compilação de um programa que usa AspectJ e Java 5.	19
Figura 5 – Exemplo de JavaBean.....	21
Figura 6 – Exemplo de programa que usa o JavaBean da Figura 5.	21
Figura 7 – Alteração que deveria ser feita para uso de log no início de cada método <i>setter</i>	22
Figura 8 – Aspecto que insere o comportamento de escrita de log antes de toda chamada a um método <i>setter</i>	23
Figura 9 – Saída do programa da Figura 6.....	23
Figura 10 – Código-fonte da Figura 6, após a aplicação automática do aspecto da Figura 8.	24
Figura 11 – Estrutura de um <i>pointcut</i>	25
Figura 12 – Interface que é usada para identificar quais JavaBeans devem receber o comportamento do aspecto.....	26
Figura 13 – Aspecto re-reescrito para tratar automaticamente os <i>listeners</i>	27
Figura 14 – O JavaBean reescrito	28
Figura 15 – O programa reescrito	29
Figura 16 – Saída do programa da Figura 15.....	30
Figura 17 – Estrutura de um servidor de aplicações Java EE	32
Figura 18 – Exemplo de partição fuzzy	36
Figura 19 – Partição fuzzy complementar	36
Figura 20 – Uma possível representação gráfica de um conjunto (a) clássico e um (b) fuzzy.....	39
Figura 21 – Aplicação de se...então sobre partições fuzzy	41
Figura 22 – O mesmo algoritmo sendo executado (a) normalmente e (b) com o uso de outro computador do grid.....	46
Figura 23 – Duas versões do mesmo programa que executa o método “demorado()” (a) localmente e (b) remotamente no grid.....	47
Figura 24 – Uma aplicação com dois threads sendo executados não-simultaneamente	49
Figura 25 – A mesma aplicação, com os dois thread sendo executados simultaneamente.....	50
Figura 26 – Na tarefa T2 termina a sua execução antes que T1 precise do seu produto	50
Figura 27 – Uma tarefa pode chamar outra sucessivamente.....	51
Figura 28 – Situação mais complexa	51
Figura 29 – A Figura 28 redesenhada para mostrar as pequenas tarefas que compõem cada thread	52
Figura 30 – Diagrama de fluxo de dados equivalente à figura Figura 28.....	53
Figura 31 – Diagrama da Figura 30 redesenhado para mostrar os threads	53
Figura 32 – Distribuição das chamadas aos métodos entre três computadores	54
Figura 33 – Exemplo de uso da anotação @SuppressWarnings	56

Figura 34 – Classe que, se compilada sem o G4JEE, pode causar problemas às aplicações	59
Figura 35 – Exemplo de classe que envia métodos para execução remota.....	61
Figura 36 – Estrutura do banco de dados distribuído.....	68
Figura 37 – Esquema de consultas ao banco de dados distribuído	69
Figura 38 – Criação de uma tabela no banco de dados distribuído.....	71
Figura 39 – Exemplo de POJO que descreve uma tabela	72
Figura 40 – Comando SQL criado automaticamente a partir da classe da Figura 39	73
Figura 41 – Esquema para cálculo de aderência a modelos temporais	77
Figura 42 – A mesma figura redesenhada para facilitar o entendimento do raciocínio	78
Figura 43 – Obtenção do perfil de uso de um computador	80
Figura 44 – Erros nas medidas	82
Figura 45 – Dependências entre tarefas para o exemplo.....	82
Figura 46 – Perfis de carga dos servidores considerados.....	83
Figura 47 – Comparação entre os desempenhos sequencial e paralelo	84
Figura 48 – Diagrama do controle fuzzy.....	86
Figura 49 – Curvas de partição fuzzy das variáveis de entrada E_1 (a) e E_2 (b).....	92
Figura 50 – Partição fuzzy das variáveis de saída.....	95
Figura 51 – Java Bean incompleto	98
Figura 52 – JavaBean da Figura 51 completado	99
Figura 53 – O método em destaque não pode ser inserido numa classe automaticamente pelo uso de aspectos.....	100
Figura 54 – Método tal qual usado na implementação deste projeto.....	101
Figura 55 – Declarações que criam avisos para o programador	102
Figura 56 – Arquitetura do tópico JMS e o papel do GridControlerMDB	106
Figura 57 – Resultados de um teste com 4 frames em um grid composto de computadores Intel P4 3GHz 1GB RAM.....	112
Figura 58 – Desempenho de um cálculo recursivo local executado em Athlon XP 2400+ 1.99 GHz 768MB RAM.....	113
Figura 59 – Desempenho de um cálculo em um grid heterogêneo calculando 24 frames.....	114
Figura 60 – O grid pode conter vários tipos de arranjo de banco de dados, inclusive concorrentes com outras aplicações fora do grid.	120
Figura 61 – Interface UserMessageType	130
Figura 62 – Exemplo de protocolo definido pelo usuário.....	131
Figura 63 – Envio de uma mensagem definida pelo usuário	131
Figura 64 – Parametrização do tipo UserMessage.....	132
Figura 65 – Interface ProcessNodeListener	133
Figura 66 – Exemplo de programa que recebe mensagens definidas pelo usuário..	133
Figura 67 – IDE Eclipse sem projetos.....	135
Figura 68 – Wizard de importação de projeto.....	136
Figura 69 – Seleção do arquivo template_g4j2ee.zip	136
Figura 70 – O projeto modelo é criado dentro do IDE	137
Figura 71 – Criação de uma classe executável.....	138
Figura 72 – A classe PrimeiroPrograma logo após a sua criação	139
Figura 73 – Edição da classe para incluir elementos remotos	140
Figura 74 – Edição do arquivo jndi.properties.....	141

Figura 75 – Execução do programa	142
Figura 76 – Resultado da execução do método remotamente.....	143
Figura 77 – Alteração do diretório padrão para salvamento de dados.....	144

LISTA DE TABELAS

Tabela 1 – Regras para o nível de dedicação σ_d	93
Tabela 2 – Regras para a confiabilidade σ_c	93
Tabela 3 – Regras para o tempo de resposta $\sigma_{t^{-1}}$	93

LISTA DE ABREVIATURAS E SIGLAS

AJDT: (AspectJ Development Tools) – Conjunto de ferramentas que permitem ao IDE Eclipse auxiliar o programador a desenvolver software baseado em AspectJ.

ANT: Linguagem de scripts desenvolvida pela organização Apache que permite ao programador criar facilmente procedimentos repetitivos como o empacotamento de aplicações. Os scripts ANT têm sido adotados largamente pela comunidade Java para automatizar as montagens de software *open source*.

AOP: (Aspect Oriented Programming) – Programação Orientada a Aspectos – Técnica de programação na qual usa-se um pré-compilador que insere código-fonte em posições específicas do código-fonte original reconhecidas por meio da identificação de ocorrências notáveis. A identificação dessas ocorrências é feita usando-se descrições fornecidas pelo próprio programador.

AspectJ: Linguagem orientada a aspectos e ferramenta de desenvolvimento de software orientados a aspectos para a plataforma Java. Tal ferramenta é composta de um pré-compilador, um navegador para a observação de aspectos em códigos-fonte Java e bibliotecas para a distribuição de aplicativos escritos usando a linguagem AspectJ dentre outros recursos.

BPEL: (Business Process Enterprise Language) – Linguagem baseada na WSDL que define processos compostos de invocações a *web services* em cadeia. Essa linguagem

cria *web services* que fazem a orquestração de outros *web services* para compor fluxos de processos corporativos.

CORBA: (Common Object Request Broker Architecture) – Arquitetura de invocação remota de objetos que usa o esquema cliente servidor.

DBMS: (Data Base Management System) – Sistema de gerenciamento de banco de dados, em português, também chamado simplesmente de “banco de dados”, refere-se ao software que mantém as informações de um banco de dados. Este acrônimo é usado para diferenciar os dados em si (Database), do software que mantém estes dados (DBMS).

Derby DB: Banco de dados da organização Apache criado usando Java puro e compatível com os padrões SQL, JDBC e Java EE.

Distribuição: Em inglês, usa-se o termo “deployment” no contexto de servidores de aplicação para indicar a ação de instalação de uma aplicação corporativa dentro de um servidor de aplicação. Neste texto, o termo “instalação” será usado nas ocasiões em que o termo correspondente em inglês for “deployment”.

EAR: (Enterprise ARchive) – Formato de arquivo J2EE contendo uma aplicação corporativa a ser instalada dentro de um servidor de aplicações Java. Estes arquivos tipicamente contém aplicações web e componentes EJB.

Eclipse: tanto a IDE *open source* quanto o consórcio de empresas que cria esta IDE.

EJB: (Enterprise Java Beans) – Nome genérico dado aos componentes corporativos da J2EE.

EJB 3.0: Versão 3 da especificação EJB cuja principal contribuição foi a simplificação dos procedimentos de distribuições das aplicações através do uso de POJOs no lugar de classes e arquivos descritores simultaneamente.

HTTP: (Hyper Text Transfer Protocol) – Protocolo para transferência de informações de hipertexto, atualmente usado tanto para hipertexto, quanto para transferência de arquivos e comunicação de *web services*.

G4JEE: (Grid for Java Enterprise Edition) – Plataforma de desenvolvimento e execução de aplicações distribuídas em um grid desenvolvida neste trabalho de mestrado.

GAF4J: (Grid Application Framework for Java) – Ferramenta para o desenvolvimento de aplicações clientes em Java, seguindo o modelo cliente-servidor para acessar serviços grid.

GAR: (Globus ARchive) – Arquivo contendo um serviço grid empacotado. Dentro do arquivo estão contidas as classes do serviço grid, junto a descritores de instalação.

GARA: (Globus Architecture for Reservation and Allocation) – Arquitetura para a alocação dinâmica e reserva de recursos de um grid.

Grid: O termo “grid” tem sido traduzido para “grade” em algumas publicações na língua portuguesa. Em inglês “grid” é utilizado comumente para nomear a malha de distribuição de energia elétrica. O uso do termo “grid” foi escolhido propositalmente para criar uma analogia com a topologia de interconexões usada na distribuição de energia, carregando maior significado que “grade”, em português, razão pela qual o termo “grid” será utilizado neste texto ao invés do termo “grade”.

GridFTP: (Grid File Transfer Protocol) – Protocolo de transferência de arquivos para grid, semelhante ao FTP.

GT: (Globus Toolkit) – Implementação de referência da OGSA.

GT3: (Globus Toolkit 3) – Versão 3 do Globus Toolkit.

GT4: (Globus Toolkit 4) – Versão 4 do Globus Toolkit.

GS: (Grid Service) – Serviço Grid.

GUI: (Graphical User Interface) – Interface gráfica com o usuário. Neste texto, basicamente, aplicativos gráficos.

GWSDL: (Grid Web Services Definition Language) – Extensão da WSDL para serviços grid. Esta linguagem foi desenvolvida como uma forma de contornar a ausência de importações na linguagem WSDL.

IDE: (Integrated Development Environment) – Ferramenta de desenvolvimento de aplicações integrada que fornece ao programador um ambiente de desenvolvimento rico e que provê a execução simples e automática de tarefas repetitivas, como a compilação do código-fonte.

J2EE: (Java 2 Enterprise Edition) – Nome antigo da plataforma Java EE.

J2ME: (Java 2 Micro Edition) – Nome antigo da plataforma Java ME.

J2SE: (Java 2 Standard Edition) – Nome antigo da plataforma Java SE.

JAAS: (Java Authentication and Authorization Service) – API que implementa a PAM para a plataforma Java. É parte integrante da J2SE.

JAR: (Java ARchive) – Arquivo contendo classes Java de propósito geral e meta informações sobre essas classes.

Java DB: (Java Database) – Banco de dados relacional compatível com o padrão JDBC que usa o código-fonte do banco de dados Derby DB e adiciona a ele novas funcionalidades.

Java EE: (Java Enterprise Edition) – Conjunto de APIs e especificações cuja finalidade é permitir o desenvolvimento de aplicações corporativas. Conjunto de ferramentas e frameworks da linguagem Java para aplicações corporativas.

Java ME: (Java Micro Edition) – Plataforma de desenvolvimento de aplicações para uso em hardware móvel ou com poucos recursos.

Java SE: (Java Standard Edition) – Plataforma de desenvolvimento e execução de aplicações Java de propósito geral. Também entendida como uma plataforma de criação de aplicações para desktop.

JDBC: (Java Database Connectivity) – Padrão de drivers para permitir a conexão de aplicativos Java a bancos de dados relacionais de uma forma padronizada.

JBoss: Servidor de aplicações Java EE *open source* modular.

JMS: (Java Message Service) – API da J2EE que permite o uso de sistemas de troca de mensagens, como o MQSeries, com aplicações corporativas escritas na linguagem Java.

JMX: (Java Management Extension) – Finalizar esta definição – Extensão da linguagem Java que permite o gerenciamento de aplicações remotamente, no estilo usado pela SNMP.

JVM: (Java Virtual Machine) – Máquina Virtual Java – Base do JRE, a JVM é responsável por prover o ambiente de execução para software escrito em Java.

JRE: (Java Runtime Environment) – Ambiente de Execução Java – Ambiente para mínimo necessário para a execução de software Java em um desktop.

PAM: (Pluggable Authentication Module) – Padrão de autenticação modular adotado largamente em ambientes de desenvolvimento de aplicações corporativas, bem como em distribuições Linux ou FreeBSD, e que usa autorização com base em usuários.

MDB: (Message-Driven Bean) – Tipo de EJB que lida com mensagens em um sistema de troca de mensagens compatível com JMS.

.NET: Plataforma corporativa criada pela Microsoft de propósito semelhante ao da Java EE.

OGSA: (Open Grid Services Architecture) – Arquitetura para serviços grid.

OGSI: (Open Grid Services Infrastructure) – Especificação da comunicação entre serviços grid.

OOP: (Object-Oriented Programming) – Técnica de programação que usa a orientação a objetos para descrever os elementos de um sistema.

P2P: (Peer-to-peer) – Tipo de computação distribuída no qual um computador se conecta a outro diretamente (sem o intermédio de um servidor) para solicitar um serviço simples (como a transferência de um arquivo). Os serviços são disponibilizados de forma voluntária para benefício mútuo obrigatório entre as partes, já que não existe vínculo entre os usuários ou os computadores desse tipo de rede.

PDI: (Performance Data Interface) – Tanto a especificação para a distribuição de dados de performance de forma histórica entre serviços grid, quanto as ferramentas que auxiliam nessa distribuição.

POJO: (Plain-old Java Object) – Modelo de programação para a criação de componentes no qual o componente é todo definido em uma única classe pelo uso de anotações. Esse modelo é uma correção do modelo anterior da J2EE para componentes, no qual além da classe contendo a implementação do componente, o programador precisava escrever interfaces remotas e descritores em XML. O modelo POJO mostrou-se mais sintético, claro e tão capaz quanto o modelo antigo composto de vários arquivos.

RFT: (Reliable File Transfer) – Serviço de transferência de arquivos baseado em serviços grid.

RMI: (Remote Method Invocation) – Protocolo de invocação remota de métodos nativo da linguagem Java no qual um servidor RMI torna disponível um conjunto de métodos declarados por meio de um *stub*.

SNMP: (Simple Network Management Protocol) – Protocolo de gerenciamento usado para administrar elementos numa rede, como roteadores ou firewalls, remotamente e de forma padronizada.

SQL: (Structured Query Language) – Linguagem usada para a criação, administração e uso de bancos de dados relacionais.

Virtualização/Virtualizar: Termos usados com o significado de tornar algo complexo, dinâmico e composto de várias partes efetivamente mais simples do ponto de vista de um usuário. Em inglês, os termos usados são “virtualization” e “to virtualize”, respectivamente.

VO: (Virtual Organization) – Organização virtual

WAR: (Web ARchive) – Formato de arquivo da J2EE que contém uma aplicação web pronta para ser instalada num contêiner web que segue a especificação J2EE.

WSDL: (Web Services Definition Language) – Tanto a linguagem XML usada para a troca de mensagens em *web services*, quando a extensão do arquivo descritor de um *web service*.

XDOCLETS: Ferramenta para a criação de componentes em Java. Esta ferramenta usa comentários para fornecer parâmetros para scripts que criam os componentes. Esta abordagem é bastante semelhante à usada pelo EJB 3.0.

1 Introdução

A computação distribuída tem tido um papel fundamental na nova ordem da computação mundial, na qual o processamento dos dados pode ocorrer virtualmente em qualquer lugar do planeta. A nova característica da computação tem sido objeto de interesse tanto para a pesquisa acadêmica (FOSTER, 2005), quanto para mercado de consumo de serviços de tecnologia da informação, demonstrando quão necessária é a demanda por novos recursos na computação distribuída. Apesar de já ter sido idealizada há pelo menos 6 anos, só agora a Computação Grid está se tornando possível, sobretudo devido ao aumento de computadores conectados e à melhoria da confiabilidade das redes.

Na idéia original, o Grid seria uma infra-estrutura de serviços trocados por todos os computadores da Internet. Nessa infra-estrutura, o poder computacional de um vasto número de computadores é compartilhado, provendo diversas formas de distribuição do armazenamento de dados, de processamento de informações e mesmo de recursos de novos tipos de equipamentos na rede, como sensores de laboratório.

A computação grid representa a etapa seguinte em termos de computação distribuída por apresentar um maior grau de interação entre computadores em rede que o padrão cliente-servidor, atualmente mais utilizado. A computação grid entende cada computador, simultaneamente, como cliente e provedor de recursos. Nesse

ponto de vista, o modelo de serviços da computação grid é uma extensão do modelo de serviços cliente-servidor.

Apesar de não usar o modelo cliente-servidor, a computação grid também se difere da computação P2P (peer-to-peer). Frequentemente esses dois modelos são confundidos entre si por apresentarem uma mudança em relação ao paradigma centralizador dos arranjos cliente-servidor.

Na computação P2P cada participante da rede P2P é independente dos demais assim como no caso das redes grid. A diferença principal entre essas duas formas de distribuição da computação está no grau de interação entre os computadores participantes.

No arranjo em grid, os participantes têm um objetivo comum e estabelecem um vínculo entre si representado pela organização virtual, sendo que a própria interação entre os usuários é desejada mutuamente.

Já no arranjo P2P cada computador usa os serviços dos demais como uma troca de favores que é interessante para ambos. Porém, a cooperação resume-se a essa troca. Assim, a comunicação em uma rede P2P ocorre de forma eventual.

Porém, por ser uma área de pesquisa ainda em desenvolvimento, já que, como visto, só recentemente foi posta em prática, a computação grid ainda possui muitas questões em aberto, como o modelo de programação e a plataforma de execução de aplicações.

1.1 Objetivo

A partir das características da programação orientada a objetos, representadas neste trabalho pela linguagem Java (JAVA, 2006), bem como das ferramentas e técnicas de desenvolvimento mais recentes, os objetivos deste trabalho de pesquisa

são produzir uma plataforma de desenvolvimento de aplicações grid e implementar uma plataforma de execução dessas aplicações. A ferramenta criada neste trabalho, chamada G4JEE (Grid for Java Enterprise Edition), terá como meta criar grids de processamento de dados.

O primeiro objetivo pretende que o ambiente de desenvolvimento permita que um programador crie uma aplicação grid, simplesmente, por meio da escrita de uma aplicação Java multithread, tão próxima de uma aplicação convencional quanto possível. Essa abordagem pretende prover uma forma elegante de criar aplicações distribuídas, sem que seja necessário que o programador tenha que controlar explicitamente a comunicação entre os processos ou mesmo saber onde realmente cada processo está sendo executado.

Para isso, será criado um mecanismo de escrita de classes no qual o código-fonte é alterado de forma automática por meio do uso da técnica de orientação a aspectos durante o seu desenvolvimento. Essa alteração automática fará substituições em códigos-fonte de aplicações, transformando-as em aplicações que são executadas pelo do grid.

Este trabalho estuda a possibilidade de desenvolvimento da computação grid de forma diferente daquela que se baseia nos serviços grid. No caso dos serviços grid os servidores devem conter todo o algoritmo que pode ser executado através de chamadas aos seus serviços. Cada componente de um grid baseado na plataforma do presente trabalho será capaz de executar tarefas definidas em qualquer nó do grid. A iniciativa de execução dessas tarefas será da máquina na qual esta tarefa foi escrita.

Ou seja, a responsabilidade por manter o programa que será executado pelo grid recai sobre o nó que solicita a execução desse programa. Assim, a instalação do grid é muito simples e não é necessária a instalação de serviços no grid, já que tanto os pedidos de execução de rotinas no grid quanto as próprias definições das rotinas são fornecidas pelo nó solicitante da execução.

Por outro lado, os grids criados usando a plataforma desse projeto exigem que o software cliente contenha todo o algoritmo que deve ser executado pelo grid. Assim, um software cliente desse grid deve ser necessariamente mais complexo que um software cliente de um serviço grid.

Já o segundo objetivo desse trabalho é criar um ambiente de execução capaz de processar programas escritos usando o ambiente de desenvolvimento, distribuindo cada uma das suas tarefas para computadores diferentes e participantes do mesmo grid. Para isso, a plataforma de execução conterà um escalonador, desenvolvido com base na teoria fuzzy.

Ao criar cada uma das duas partes deste projeto, optou-se por criar uma nova plataforma de grid para demonstrar a viabilidade de se construir um grid simples usando como base software *open source* (Eclipse, JBoss, Java EE¹), usado atualmente com sucesso em projetos comerciais de grande porte.

1.2 Motivação

Atualmente existe uma grande gama de soluções para a criação de grids computacionais. Entretanto, a despeito dos esforços feitos nos últimos anos para a

¹ Embora a implementação da JVM fornecida pela Sun Microsystems seja gratuita, mas de código fechado, a especificação da plataforma Java EE é aberta. Além disso, existe atualmente algumas JVMs alternativas *open source*. A própria Sun Microsystems anunciou que irá abrir o código fonte da sua JVM para a comunidade de desenvolvedores Java.

criação de padrões para os grids, como a OGSA, por exemplo, ainda não existe uma solução aceita consensualmente.

Assim, os esforços das pesquisas em grid têm se voltado para a infra-estrutura do grid, seus aspectos de interoperabilidade e seu desempenho. Pouca atenção tem sido dada, no entanto, para o ambiente de desenvolvimento dessas aplicações grid.

O desenvolvimento para as plataformas corporativas, como o Java EE e o .NET estão em um estágio mais evoluído, no qual um grande esforço é gasto com melhorias no ambiente de desenvolvimento (IDEs, bibliotecas e técnicas de desenvolvimento). Nestas plataformas, já existe uma vasta gama de ferramentas que facilitam o desenvolvimento de sites, *web services* e componentes corporativos.

Atualmente, existem alguns kits de desenvolvimento para grid, como apresentado em Laszewski; Foster; Gawor; Lane (2001), destinado ao desenvolvimento em Java. Apesar disso, tanto escrever aplicações para grids quanto instalar grids ainda é bastante custoso para o desenvolvedor e a abordagem desta pesquisa procura tornar as duas tarefas mais acessíveis ao programador não especializado em computação distribuída, possibilitando que um grid simples, mas capaz de virtualizar² capacidade de processamento e armazenamento, seja criado com pouco esforço do desenvolvedor.

A plataforma desenvolvida neste projeto é particularmente útil nos casos em que o código-fonte dos grids muda frequentemente. Este é o caso, por exemplo, de grids desenvolvidos para executar a análise de dados ou simulações. Nesse exemplo de aplicação, os dados experimentais podem ser tratados de uma infinidade de formas para testar teorias a respeito do fenômeno estudado.

² Ver referência sobre o termo “virtualizar” na lista de abreviaturas e siglas

Além disso, os resultados deste projeto, principalmente o mecanismo de alteração automática de código e o escalonador fuzzy, podem ser aproveitados em outras plataformas de grid. Aplicações paralelas cujo processamento é demorado e cujo algoritmo é alterado muito freqüentemente pelos programadores, poderão ser desenvolvidas facilmente usando um grid construído nos moldes da plataforma proposta neste projeto.

1.3 Trabalhos relacionados

Alguns trabalhos possuem uma proposta semelhante à deste projeto. Antes de apresentar a proposta deste trabalho no Capítulo seguinte, é interessante apresentar outras abordagens e seus contrastes em relação à proposta desse projeto.

A proposta da ferramenta GAF4J (GAF4J, 2006) é disponibilizar uma API para o desenvolvimento de aplicações baseadas em serviços grid. As classes dessa API são responsáveis por chamar serviços grid já instalados nos servidores do grid.

Ao usar essa API, o programador deve criar threads específicos para o grid e saber chamar seus serviços. Essa abordagem resolve o problema de disponibilizar os recursos do grid para software escrito em Java, mas obriga o programador a criar software dependente e vinculado ao grid.

Além disso, a abordagem requer que os serviços grid sejam instalados de antemão e que o programador aprenda a usar a API específica da GAF4J, inclusive adotando classes de processamento paralelo específicas ao invés de usar as classes padrão da linguagem Java.

1.4 Metodologia

O método utilizado para este trabalho de mestrado teve como ponto de partida um estudo panorâmico sobre a pesquisa em computação grid. Para isso, recorreu-se inicialmente a alguns livros nos quais se buscou aprender os conceitos fundamentais e estabelecidos da computação grid.

A seguir, foi feito um levantamento bibliográfico dos artigos mais representativos do estado da arte da pesquisa nessa área. Os artigos serviram como base para que fosse feito um levantamento das oportunidades da pesquisa em grid.

Após a leitura de parte dessa bibliografia, ficou claro que a plataforma Globus era a plataforma mais aceita para a construção de grids computacionais. Assim, decidiu-se inicialmente para este projeto que a pesquisa utilizaria como base o Globus Toolkit.

O passo seguinte consistiu em um aprendizado sobre como utilizar o Globus Toolkit. Algum tempo depois do início do desenvolvimento do software deste projeto, concluiu-se que a plataforma Globus seria útil apenas para resolver a questão de comunicação entre os nós do grid.

Mas as plataformas corporativas orientadas a objetos (como a plataforma Java EE) já implementam mecanismos eficientes de comunicação entre clientes e tornam o desenvolvimento de aplicações distribuídas mais simples do que uma plataforma ainda em desenvolvimento como o Globus.

Uma vantagem da plataforma Globus é a sua aderência à OGSA. Um software implementado sobre a plataforma Globus é capaz de interagir com outros sistemas que também sejam compatíveis com a OGSA.

1.5 Organização do Texto

O restante deste texto está dividido numa sequência lógica que visa inicialmente apresentar a teoria na qual se baseia este trabalho, a estratégia escolhida para esta pesquisa e, por fim, os resultados obtidos neste trabalho.

No Capítulo 2, são explicados os conceitos básicos para o entendimento da abordagem usada para a solução do problema. O Capítulo 0 apresenta a estratégia utilizada nesta pesquisa. Nesse capítulo, são apresentados argumentos que suportam as decisões arquiteturais da solução encontrada para tornar o objetivo desta pesquisa concreto.

Detalhes a respeito dos problemas resolvidos durante a implementação da plataforma desta pesquisa são expostos no Capítulo 4. Procurou-se apresentar uma coleção dos problemas mais complexos, para que seja possível entender melhor o funcionamento interno das principais partes componentes do grid.

Finalizando o conteúdo principal deste texto, o Capítulo 5 resume os resultados que foram obtidos nesta pesquisa e as conclusões obtidas a partir desses resultados.

Os apêndices ao final do texto apresentam algumas partes importantes do código-fonte do sistema e alguns manuais sucintos de uso da plataforma. Os códigos-fonte são referenciados em algumas partes do conteúdo principal deste texto e os manuais mostram de forma clara o processo de desenvolver usando as ferramentas criadas nesta pesquisa.

2 Conceitos

Neste capítulo, são explicados os principais conceitos utilizados na criação da plataforma para grids. Desta forma, serão fornecidos subsídios para o entendimento das decisões tomadas durante o desenvolvimento, tema que será objeto de discussão dos capítulos seguintes.

2.1 Organizações Virtuais

Uma organização virtual é uma entidade composta por um conjunto de pessoas e por uma infra-estrutura computacional, emprestadas de diversas organizações reais. O objetivo de uma organização virtual é tornar possível o trabalho colaborativo de grupos de pessoas distantes entre si. Em uma organização virtual, a colaboração entre esses grupos pode se dar por meio da troca de informações ou pela doação mútua de parte da sua capacidade de computação de dados. Uma organização real, neste contexto, pode ser uma empresa ou um instituto de pesquisa, por exemplo.

A infra-estrutura computacional de uma organização virtual é composta por máquinas situadas em organizações reais. Cada organização real encerra um chamado domínio administrativo: o espaço que delimita um conjunto de máquinas que são administráveis da mesma forma e pela mesma equipe. Máquinas em domínios administrativos distintos são presumivelmente administradas por equipes distintas e podem estar subordinadas a políticas diferentes de manutenção.

Por exemplo, na Figura 1, quatro organizações reais colaboram com parte dos seus computadores para a organização virtual que está representada pela linha tracejada. As quatro organizações reais podem ser, por exemplo, quatro institutos de pesquisa sobre um mesmo assunto e cada um dos quatro institutos deve ter interesse em consultar informações geradas pelas pesquisas dos outros três.

Figura 1 – Estrutura de uma organização virtual

A organização Org1 disponibiliza uma estação de trabalho e um servidor para o grid de forma não exclusiva. A estação de trabalho pode ser usada por um funcionário da organização durante as horas de trabalho e o servidor deve prover serviços para as quatro estações de trabalho da organização. Assim, por exemplo, se a estação de trabalho da organização Org2 pedir um serviço localizado no servidor da organização Org1, é possível que este serviço não seja processado prontamente se

este servidor estiver ocupado com uma tarefa solicitada por alguma estação de trabalho da organização Org1.

Na Figura 1, é exibida esquematicamente a estrutura lógica da organização virtual após a virtualização da infra-estrutura de rede. Um exemplo de estrutura física que pode ter sido virtualizada é melhor representado na Figura 2.

Figura 2 – Estrutura física da mesma organização virtual

Pela Figura 2, pode-se observar que a forma de comunicação entre as organizações pode variar ou mesmo ser redundante. Do ponto de vista de um usuário de um serviço situado em um domínio administrativo distinto, não há como requerer, a priori, que o serviço esteja sempre disponível. Assim, computadores componentes

de uma organização virtual podem ser classificados em graus de confiabilidade ou disponibilidade.

O software usado pelo usuário deve ter a capacidade de acessar qualquer recurso da organização virtual. Para que isso seja possível, o software de todos os participantes do grid deve seguir um mesmo padrão.

Cada usuário de uma VO deve ter uma única identidade com a qual acessa os serviços da VO. A existência dessas contas únicas evita que o usuário precise memorizar várias senhas ou que o software que utiliza precise conhecer várias formas de uso dessas senhas.

Cada identidade de usuário possui acesso a um conjunto específico de serviços. Por exemplo, pode haver serviços de administração da VO que são usados apenas pelos usuários administradores, enquanto que serviços de usuário final podem ter um direito de acesso menos restritivo.

2.2 Grid

Um grid (FOSTER et al., 2001-A), (FOSTER et al., 2001-B) pode ser entendido como sendo a infra-estrutura computacional que possibilita a existência de uma organização virtual conforme descrito na Seção 2.1. Mais especificamente, um grid é uma infra-estrutura de serviços que visa virtualizar os recursos dos componentes do grid, ocultando os detalhes de como usar cada um desses recursos.

Redes Grid são estruturas compostas de serviços de rede, mecanismos unificados de segurança, estações de trabalho e servidores que se comunicam através de padrões para longas distâncias (entre dois pontos quaisquer da Terra). Nas redes grid, toda infra-estrutura complexa, da qual o grid depende, é virtualizada para que

os usuários possam usar seus serviços através do domínio do padrão usado por todo grid.

Um grid é composto por um conjunto dinâmico de computadores. Novos computadores tornam-se disponíveis ou indisponíveis para o grid a qualquer momento. Um componente pode ser uma estação de trabalho que é desligada ao final do expediente ou que pode não estar em perfeito funcionamento, por exemplo.

Ao construir um grid, é previsto que um dos seus computadores não consiga controlar ou prever a disponibilidade dos recursos situados em um domínio administrativo diferente do seu. Dentro de um mesmo computador, aplicações locais e serviços grid sendo executados podem competir pelos recursos do computador. Num caso como esse, geralmente as aplicações locais têm prioridade sobre os serviços grid. Outro computador no mesmo grid, em um domínio administrativo diferente, não é capaz de prever quando isso ocorre.

Em linhas gerais, um teste simples apresentado em Foster et al. (2002) permite verificar um arranjo de rede sobre a aderência ao conceito de “grid”.

2.3 Serviços Grid

Toda organização real que queira contribuir para alguma organização virtual através do empréstimo de parte de sua capacidade computacional deve disponibilizar os seus recursos seguindo o padrão adotado pela VO. Num grid, essa padronização é obtida fazendo-se com que cada computador que compõe o grid forneça o mesmo tipo de serviços para os demais. Por exemplo, todos os componentes de um grid podem fornecer webservices ou todos podem fornecer serviços RPC. Esses serviços trocados por todos os computadores em um grid são os chamados serviços grid.

Os serviços grid diferem-se de um serviço do tipo cliente-servidor simples por permitirem mecanismos de notificação, reserva de recursos e controle de ciclo de vida remotamente.

2.3.1 Notificações

As notificações permitem que uma aplicação cliente de um grid peça para que um serviço do grid envie uma mensagem de aviso tão logo uma determinada condição seja verificada. Numa arquitetura cliente-servidor, na qual a iniciativa da comunicação deve partir sempre do cliente, o mesmo efeito seria obtido somente se o cliente fizesse uma série de chamadas ao servidor, pedindo para que o servidor verificasse a mesma condição repetidas vezes. Essa abordagem requer um esforço desnecessário do servidor e obriga que a comunicação seja síncrona com o cliente, tornando impossível identificar a condição assim que ela ocorrer.

Por outro lado, o mecanismo de notificações requer que o cliente de um serviço grid também seja capaz de receber dados. Isso é obtido, em algumas plataformas grid, como por exemplo o Globus Toolkit, ao tornar o cliente, obrigatoriamente, também um servidor.

2.3.2 Reserva de recursos

A reserva de recursos, como descrita em Foster et al. (1999), permite que os recursos virtualizados pelo grid sejam reservados previamente por um cliente deste recurso. Assim, o cliente pode ter acesso exclusivo ao recurso e é possível que o recurso seja usado de forma mais otimizada, visto que dessa forma elimina-se concorrências pelo mesmo recurso entre diversos clientes.

Por exemplo, um cliente pode requerer previamente a transferência de um arquivo. Uma parte da banda disponível para o tráfego desse arquivo pode ser

alocada especialmente para esta transferência. A transferência desse arquivo seguramente ocorrerá mais rapidamente do que num caso em que outros clientes requerem outras transferências de arquivo ao mesmo tempo.

2.3.3 Controle de ciclo de vida de serviços grid

Finalmente, o controle de ciclo de vida permite que os serviços prestados por um grid sejam dinâmicos e criados conforme a necessidade do grid. O controle do ciclo de vida feito dessa forma faz com que o grid não precise instanciar recursos para antecipar eventuais necessidades futuras. Ao invés disso, o grid passa a ter instâncias apenas de serviços que serão realmente usadas pelos clientes.

De forma similar, as instâncias dos serviços podem ser eliminadas assim que não forem mais necessárias.

2.3.4 Contratos entre os componentes de um grid

Esses três mecanismos (de notificação, de reserva de recursos e de controle de ciclo de vida) reforçam a necessidade de um forte contrato entre os participantes de um grid. Ou seja, os participantes devem concordar com um mesmo padrão de comunicação e devem confiar até certo ponto nos demais participantes.

Por exemplo, se o acesso aos serviços grid ocorresse sem restrições a determinados clientes, correr-se-ia o risco de que os recursos do grid fossem reservados por um usuário que causasse o travamento de um recurso do grid de forma intencional.

2.3.5 OGSA e OGSi

A OGSA (Open Grid Services Architecture), especificação atualmente mais aceita para os serviços grid, descreve a virtualização dos recursos por meio de *web services*.

Ao invés de propor uma estrutura completamente nova para definir os serviços grid, apenas a interpretação de como os *web services* devem ser usados foi expandida. Assim, para a OGSA, um serviço grid é um tipo especial de *web service*, da mesma forma que um *web service* pode ser entendido um tipo especial de documento dinâmico disponível através de uma URL.

Os serviços grid requerem que o solicitante do serviço também seja um servidor de serviços grid para que ocorra comunicação bidirecional. Isso possibilita que notificações sejam enviadas para os clientes de serviços grid, como visto na Seção 2.3.

A arquitetura descrita pela OGSA necessita de um protocolo entre os clientes e os servidores. Este protocolo se traduz, logicamente, em uma especificação de um conjunto de métodos que um *web service* deve disponibilizar para que ele seja considerado um serviço grid. Esse conjunto de métodos, bem como a sua interpretação, é descrito pela especificação OGSi (Open Grid Services Infrastructure), cujo documento é Tuecke et al. (2003).

Neste trabalho, algumas idéias principais da OGSA serão usadas, mas não se aproveitará a OGSi, visto que a plataforma deste trabalho não usa a comunicação baseada nos *web services*, conforme será visto mais tarde na Seção 3.2.3 (Sistema de Troca de Mensagens).

2.4 Orientação a aspectos

Orientação a aspectos ou AOP (Aspect-Oriented Programming) é uma técnica de programação que estende a técnica orientada a objetos (HAVINGA; NAGY; BERGMANS). Na AOP, um código compilável é gerado por um pré-compilador configurável a partir de um código original redigido pelo programador. Este código original é alterado para tratar de forma uniforme os problemas recorrentes que o programador precisa resolver.

Assim, o programador desobriga-se de rever manualmente o código-fonte à procura desses problemas recorrentes e, também manualmente, de resolver todos esses problemas da mesma forma, garantindo clareza, modularidade e praticidade de codificação.

Em orientação a aspectos, os problemas recorrentes são chamados “cross-cutting concerns”, pois são entendidos como preocupações encontradas de forma horizontal por todo o software. A técnica de programação orientada a aspectos consiste em escrever detectores desses problemas recorrentes e em informar ao pré-compilador qual deve ser a alteração no código-fonte original quando ocorrer a detecção desses problemas recorrentes.

Por exemplo, usando orientação a aspectos, o programador pode fazer com que, antes da execução de qualquer método que receba como parâmetro uma conexão com o banco de dados, seja escrito uma entrada de log num arquivo. Nesse exemplo, a escrita de logs é um *cross-cutting concern*.

Figura 3 - Fases na compilação de um programa escrito usando orientação a aspectos

Na Figura 3, pode-se observar as etapas para a criação de um programa compilado a partir de um código-fonte original e de uma definição de aspectos. O pré-compilador de aspectos usa a definição de aspectos para determinar de qual forma o código-fonte original deve ser alterado. A alteração desse código-fonte original gera um código-fonte no padrão da linguagem base. Esse último código-fonte pode, então, ser lido pelo compilador padrão da linguagem base, gerando um programa compilado.

A definição de aspectos deve ser escrita usando uma linguagem orientada a aspectos. Tendo Java como a linguagem base, a linguagem de programação orientada a aspectos é a AspectJ (ASPECTJ, 2006), desenvolvida pelo consórcio Eclipse (ECLIPSE, 2006).

A própria versão 5.0 da linguagem Java (JAVA 1.5, 2006) inseriu o conceito de pré-compilador na linguagem para suportar uma maior flexibilidade do código e outras características como tipos parametrizados. Assim, um programa compilado

usando tanto AspectJ quanto a versão 5.0 da linguagem Java passará por dois compiladores, conforme mostrado na Figura 4.

Figura 4 – Compilação de um programa que usa AspectJ e Java 5.

Essas duas fases causam um aumento no tempo de compilação, mas esse tempo não mostrou ser apreciável durante os testes efetuados. Ou seja, mesmo com o peso de dois pré-compiladores, o ambiente de desenvolvimento responde rápido o suficiente para permitir seu uso em ambientes de desenvolvimento de aplicações nos quais os requisitos mudam constantemente.

Com a ferramenta AspectJ, é possível criar aspectos de forma análoga à criação de classes, interfaces ou anotações. Um aspecto é composto por uma declaração que define as ocasiões em que esse aspecto é ativado (o chamado *pointcut*), e o código que é executado quando tal condição é detectada (o chamado *advice*).

As ocorrências, dentro de uma linha de execução Java, de um *pointcut* é chamada “join point”. Isto é, um *join point* é uma ocasião em que um *advice* é

aplicado num programa. *Advices* podem ser aplicados antes de uma chamada a um método, depois dessa chamada ou no lugar dessa chamada.

Os aspectos podem ser aplicados em programas pré-existentes, adicionando um tratamento mais refinado a questões recorrentes, sem que haja a necessidade de reescrever o código-fonte. Essa ferramenta também pode ser utilizada para facilitar a programação com testes, criando pontos de verificação automaticamente.

Além de permitir a execução de *advices*, a AspectJ permite também que os aspectos alterem a estrutura das classes e interfaces, adicionando variáveis de instância, de classe ou novos métodos. Esse mecanismo é conhecido em AOP como “inter-type declaration”.

2.4.1 Exemplo de aspecto

O funcionamento da orientação a aspectos, pode ser observado por meio de um exemplo simples. Nesse exemplo, deseja-se criar uma forma padronizada de se incluir mecanismos de *listeners* (alvos de notificações) em classes do tipo JavaBean.

JavaBean é o nome dado aos componentes Java que seguem a recomendação JavaBean. Segundo esta recomendação, leituras e alterações em um campo chamado X de uma classe devem ser feitas por meio de métodos respectivamente chamados getX e setX. Esses métodos são chamados de forma geral e respectivamente de métodos *setter* e *getter*. A recomendação também especifica como deve ser escrita uma classe JavaBean para que outros objetos cadastrem-se como *listeners* de eventos que podem ocorrer com o JavaBean.

Tome-se, inicialmente o JavaBean da Figura 5. Esse JavaBean é um componente simples que contém dois campos: número e texto.

```
1 package tests;
2
3 public class ProdutoBean
4 {
5     private int codigo = 0;
6     private String descricao = "";
7
8     public int getCodigo()
9     {
10         return codigo;
11     }
12     public void setCodigo(int codigo)
13     {
14         this.codigo = codigo;
15     }
16     public String getDescricao()
17     {
18         return descricao;
19     }
20     public void setDescricao(String descricao)
21     {
22         this.descricao = descricao;
23     }
24 }
```

Figura 5 – Exemplo de JavaBean

Tome-se, por exemplo, o pequeno programa da Figura 6 que usa o JavaBean da Figura 5.

```
1 package tests;
2
3 public class Program
4 {
5     public static void main(String args[])
6     {
7         ProdutoBean produto = new ProdutoBean();
8         produto.setCodigo(10);
9         produto.setDescricao("Sabonete");
10    }
11 }
```

Figura 6 – Exemplo de programa que usa o JavaBean da Figura 5.

A técnica tradicional de desenvolvimento obriga os programadores que desejem a inclusão de uma mensagem num log, antes de cada alteração de uma propriedade deste JavaBean, a escrever uma chamada para tal inclusão no começo de cada método *setter*, conforme mostrado em destaque na Figura 7.

```

1 package tests;
2
3 public class ProdutoBean
4 {
5     private int codigo = 0;
6     private String descricao = "";
7
8     public int getCodigo()
9     {
10         return codigo;
11     }
12     public void setCodigo(int codigo)
13     {
14         System.out.println("Algum valor de " + this + " sera alterado");
15         this.codigo = codigo;
16     }
17     public String getDescricao()
18     {
19
20         return descricao;
21     }
22     public void setDescricao(String descricao)
23     {
24         System.out.println("Algum valor de " + this + " sera alterado");
25         this.descricao = descricao;
26     }
27 }

```

Figura 7 – Alteração que deveria ser feita para uso de log no início de cada método *setter*

Esse problema torna-se ainda mais complexo quando se tem uma quantidade grande de JavaBeans e de propriedades desses JavaBeans. Obrigar o programador a seguir um padrão desse tipo torna o código poluído e aumenta enormemente a possibilidade de que nele existam imperfeições.

O mesmo efeito seria obtido mais facilmente, de forma mais consistente e elegante, através da aplicação do aspecto mostrado na Figura 8. Ou seja, o aspecto da Figura 8 faz com que o pré-compilador da linguagem AspectJ transforme o código-fonte da Figura 5 num código-fonte equivalente ao da Figura 7.

```
1 package tests;
2
3 public aspect MeuAspecto
4 {
5     pointcut change(Object ref) : call(void *.set*(..)) && target(ref);
6
7     before(Object ref) : change(ref)
8     {
9         System.out.println("Algum valor de " + ref + " sera alterado");
10    }
11 }
```

Figura 8 – Aspecto que insere o comportamento de escrita de log antes de toda chamada a um método *setter*

A Figura 9 mostra a saída quando os códigos da Figura 5, da Figura 6 e da Figura 8 são compilados juntos e a classe Program é executada. Observando-se os códigos-fonte de Program e de ProdutoBean sob o ponto de vista da programação orientada a objetos convencional, nota-se que não existe no código-fonte original chamada à impressão de qualquer mensagem na tela do usuário. O resultado obtido ocorre por influência automática do aspecto MeuAspecto.

```
1 Algum valor de tests.ProdutoBean@11b86e7 sera alterado
2 Algum valor de tests.ProdutoBean@11b86e7 sera alterado
```

Figura 9 – Saída do programa da Figura 6

Ao usar a ferramenta AJDT (AJDT, 2006) como um plugin dentro do IDE Eclipse, os aspectos são aplicados automaticamente a todo código-fonte presente no mesmo projeto dos aspectos. Dessa forma, a simples presença do aspecto da Figura 8 altera a classe da Figura 6, fazendo com que ela fique tal qual o código-fonte da Figura 10. Essa transformação ocorre de forma transparente para o programador. Nessa última figura, as linhas inseridas pelo aspecto estão destacadas em negrito.

```

1 package tests;
2
3 public class Program
4 {
5     public static void main(String args[])
6     {
7         ProdutoBean produto = new ProdutoBean();
8         System.out.println("Algum valor de " + produto + " sera alterado.");
9         produto.setCodigo(10);
10        System.out.println("Algum valor de " + produto + " sera alterado.");
11        produto.setDescricao("Sabonete");
12    }
13 }

```

Figura 10 – Código-fonte da Figura 6, após a aplicação automática do aspecto da Figura 8.

A declaração do *pointcut* “change” na linha 5 da Figura 8 informa que um *join point* é detectado num código-fonte toda vez que for feita uma chamada para métodos cujo nome começa com “set”, que não retornam um valor, que são definidos em qualquer classe e que recebem qualquer conjunto de parâmetros.

A sintaxe básica para as chamadas de métodos está apresentada na Figura 11. O tipo de *pointcut* define o tipo de evento que será identificado pelo *pointcut*. Além de uma chamada para um método (*pointcut* do tipo “call”), os *pointcuts* podem também detectar, por exemplo, o disparo de exceções (tipo “handler”) ou as chamadas feitas dentro de uma classe (tipo “within”), dentre outros tipos de *pointcut*. O nome da classe, o nome do método e a lista de argumentos podem usar *wild cards* (asteriscos ou sinais de interrogação) na definição de um *pointcut*.

Figura 11 – Estrutura de um *pointcut*

O *pointcut* da Figura 11 detecta qualquer chamada aos métodos cujos nomes começam com “get”, que estão definidos em classes cujos nomes terminam com “Factory”, que retornam um valor inteiro e cuja lista de argumentos começa com uma `String` ou é composta apenas de uma `String`.

Uma vez que AspectJ detecta um *join point*, a ferramenta altera o código-fonte de acordo com o tipo de *advice* que está relacionado com o *join point*. Nesse exemplo, como o *advice* é do tipo “before”, a ferramenta AspectJ insere as linhas do código-fonte antes da linha na qual foi identificado o *join point*.

2.4.2 Outro exemplo mais complexo

No exemplo que foi estudado, o aspecto “before” é aplicado em toda e qualquer classe que contenha um método cujo nome é iniciado por “set”. Este comportamento pode ser indesejado, por alterar classes usando um critério pouco rigoroso para identificar os *join points*. Em um caso mais próximo de um uso prático, o programador deve querer ser capaz de indicar onde aplicar o aspecto.

Para obter esse efeito, pode-se limitar a atuação do aspecto somente a classes que implementem uma determinada interface. O programador passa, então, a

escolher se o seu `JavaBean` deve ser ou não alvo do aspecto, ao controlar se a classe do seu `JavaBean` implementa ou não aquela interface detectada pelo aspecto.

```
1 package tests;  
2  
3 public interface AOPJavaBean<W>  
4 {  
5 }
```

Figura 12 – Interface que é usada para identificar quais `JavaBeans` devem receber o comportamento do aspecto

Sendo a interface detectada a descrita pela Figura 12, a declaração do *pointcut* poderia ser trocada para a mostrada na linha 20 da Figura 13. A cláusula “target” desta linha permite a captura do objeto que é alvo do aspecto e a declaração do parâmetro “ref” no *pointcut* limita a atuação do *pointcut* para tipos que implementam a interface `AOPJavaBean`.

```

1 package tests;
2
3 import java.util.Iterator;
4 import java.util.Vector;
5
6 public aspect MeuAspecto
7 {
8     private Vector<ChangeListener<W>> AOPJavaBean.listeners = new ➔
➔ Vector<ChangeListener<W>>();
9
10    public void AOPJavaBean.addChangeListener(ChangeListener<W> listener)
11    {
12        this.listeners.add(listener);
13    }
14
15    public void AOPJavaBean.removeChangeListener(ChangeListener<W> listener)
16    {
17        this.listeners.remove(listener);
18    }
19
20    pointcut change(AOPJavaBean ref) : call(void *.set*(...)) && target(ref);
21
22    before(AOPJavaBean ref) : change(ref)
23    {
24        System.out.println("Algum valor de " + ref + " sera alterado");
25    }
26
27    after(AOPJavaBean ref) : change(ref)
28    {
29        Iterator<ChangeListener> i = ref.listeners.iterator();
30        while(i.hasNext())
31        {
32            i.next().changePerformed(ref);
33        }
34    }
35 }

```

Figura 13 – Aspecto re-escrito para tratar automaticamente os *listeners*

O JavaBean deve ser reescrito conforme o código-fonte da Figura 14. A única diferença desse código-fonte para o da Figura 5 é a presença, na linha 3, da cláusula “implements AOPJavaBean<ProdutoBean>” que marca o JavaBean para receber alterações do aspecto MeuAspecto.

```

1 package tests;
2
3 public class ProdutoBean implements AOPJavaBean<ProdutoBean>
4 {
5     private int codigo = 0;
6
7     private String descricao = "";
8
9     public int getCodigo()
10    {
11        return codigo;
12    }
13
14    public void setCodigo(int codigo)
15    {
16        this.codigo = codigo;
17    }
18
19    public String getDescricao()
20    {
21        return descricao;
22    }
23
24    public void setDescricao(String descricao)
25    {
26        this.descricao = descricao;
27    }
28 }

```

Figura 14 – O JavaBean reescrito

Conforme já descrito, os JavaBeans também podem definir *listeners*. Para padronizar o uso dos *listeners*, a especificação dos JavaBeans recomenda que se utilize métodos para incluir e remover os *listeners*. Os nomes desses métodos devem seguir o padrão “addXListener” e “removeXListener”, respectivamente, onde X é o tipo de evento que é observado pelos *listeners*.

Para conseguir esse efeito em todos os JavaBeans detectados, o aspecto da Figura 13 altera a interface AOPJavaBean usando “*inter-type declaration*”. Todas as alterações feitas por um aspecto numa interface são propagadas pela AspectJ para as classes que implementam essa interface.

O aspecto da Figura 13 altera as classes através da criação de um campo privado chamado “listeners”, do tipo Vector, que mantém referências a todos os *listeners*, e da inclusão dos métodos addChangeListener e removeChangeListener.

Para notificar os *listeners* a respeito da chamada de um método *setter*, o aspecto contém um *advice* do tipo “after” que chama, um a um, os métodos *changePerformed* de cada *listener*.

É interessante notar que, uma vez escrito o aspecto, uma classe tão simples quanto a da Figura 5 passa a contar com um log e com um mecanismo de *listeners* simplesmente por meio da implementação de uma interface.

A Figura 15 mostra uma outra versão do programa que usa o *JavaBean*. Nessa figura, primeiro são criados dois *listeners* cujos tipos são duas *inner classes* anônimas. Esses dois *listeners* são adicionados ao objeto “produto” e a seguir são chamados os dois métodos *setters*, que causam a impressão das oito linhas apresentadas na Figura 16.

```

1 package tests;
2
3 public class Program
4 {
5     public static void main(String args[])
6     {
7         ChangeListener<ProdutoBean> listener0;
8         listener0 = new ChangeListener<ProdutoBean>()
9         {
10             public void changePerformed(ProdutoBean object)
11             {
12                 System.out.println("codigo: " + object.getCodigo());
13                 System.out.println("descricao: " + object.getDescricao());
14             }
15         };
16
17         ChangeListener<ProdutoBean> listener1;
18         listener1 = new ChangeListener<ProdutoBean>()
19         {
20             public void changePerformed(ProdutoBean object)
21             {
22                 System.out.println("111");
23             }
24         };
25
26         ProdutoBean produto = new ProdutoBean();
27         produto.addChangeListener(listener0);
28         produto.addChangeListener(listener1);
29
30         produto.setCodigo(10);
31         produto.setDescricao("Sabonete");
32     }
33 }

```

Figura 15 – O programa reescrito

```
1  Algum valor de tests.ProdutoBean@9cab16 sera alterado.  
2  codigo: 10  
3  descricao:  
4  111  
5  Algum valor de tests.ProdutoBean@9cab16 sera alterado.  
6  codigo: 10  
7  descricao: Sabonete  
8  111
```

Figura 16 – Saída do programa da Figura 15.

Após a criação do aspecto, outros JavaBeans podem ser criados com o mesmo comportamento simplesmente fazendo com que os JavaBeans implementem a interface `AOPJavaBean`.

Além de facilitar a criação de futuros JavaBeans, essa abordagem torna o comportamento de todos os JavaBeans de um sistema muito mais modular e uniforme. Para alterar o mecanismo de *listeners* de todos os JavaBeans, bastaria alterar o algoritmo dentro do aspecto.

Existem outras formas de detecção de *pointcuts* e de verificações de consistência na linguagem AspectJ. Nesta seção, foram apresentados os conceitos básicos da linguagem que dão subsídios para o entendimento do uso da AspectJ neste trabalho. Para informações mais detalhadas a respeito da linguagem AspectJ, recomenda-se a leitura de ASPECTJ5 (2006).

2.5 Java EE

Java EE (Java Enterprise Edition) é a versão da linguagem Java para o desenvolvimento de aplicações corporativas. Esta versão da linguagem é composta de um conjunto de especificações e bibliotecas que servem para a criação de aplicações servidoras que tipicamente servem uma quantidade muito grande de usuários remotos simultaneamente.

Atualmente, a maior parte das aplicações escritas em Java tem sido desenvolvida sobre a plataforma Java EE pelo fato de a linguagem Java ser bastante adequada na criação de componentes reutilizáveis, o que é particularmente interessante para a criação de aplicações servidoras de grande porte. Nessas aplicações, a quantidade de código-fonte é muito grande, sendo a gerência desse código-fonte crucial para a manutenção dos sistemas. Os sistemas corporativos também se caracterizam pela constante mudança dos seus pré-requisitos, que acompanham as mudanças de necessidades dos usuários desses sistemas.

As aplicações grid também requerem manutenções constantes, pois seus requisitos estão em constante mudança, dados a complexidade dessas aplicações e o caráter dinâmico das necessidades dos usuários. Por isso, optou-se por uma plataforma corporativa como base para os grids criados a partir da ferramenta desse trabalho.

Além disso, desenvolver o grid com componentes da Java EE torna o grid acessível para aplicações baseadas na Java EE. Assim, já que a plataforma base para o grid é a plataforma Java EE, é muito simples desenvolver uma aplicação web ou um *web service* baseados em Java EE que usem os grids deste trabalho. Da mesma forma, seria simples alterar uma aplicação Java EE já existente para usar o grid.

Figura 17 – Estrutura de um servidor de aplicações Java EE

A Figura 17 mostra a estrutura de um servidor de aplicações dessa plataforma. Na figura está representada a capacidade que os servidores de aplicações têm de servir um vasto número de aplicações diferentes e simultâneas. Cada aplicação pode ter seu próprio protocolo. O servidor de aplicações encarrega-se de lidar com esses protocolos e converter as mensagens desses protocolos sobre a forma de chamadas a métodos das aplicações corporativas instaladas dentro dele.

O componente principal da Java EE é o servidor de aplicações (*application server*) que é uma aplicação servidora genérica. O servidor de aplicações recebe aplicações Java EE e trata de prover-lhes serviços básicos como conexões a bancos de dados, serviços de log, comunicação de rede com os clientes, gerenciamento de concorrências entre serviços, instanciação de componentes, administração de tópicos e filas de mensagens, transações, segurança, dentre outros serviços.

As aplicações Java EE, escritas para serem executadas dentro do servidor de aplicações, não precisam, pois, resolver os problemas recorrentes a toda aplicação

corporativa. Ao invés disso, as aplicações Java EE limitam-se a controlar os fluxos de dados e implementar os algoritmos de lógica de negócios.

O contêiner do servidor de aplicações provê um isolamento que coloca de um lado os protocolos e os recursos como banco de dados, por exemplo, e de outro as aplicações Java EE. As aplicações Java EE são executadas dentro do contêiner. Esse contêiner disponibiliza os elementos que servem de base para a execução da aplicação sobre a forma de objetos Java. Ao mesmo tempo, o contêiner limita a operação dessa aplicação servidora, impossibilitando que essa aplicação cause danos ao servidor ou interfira na execução de outras aplicações que estejam sendo executadas no mesmo servidor.

Neste trabalho, o servidor de aplicações é usado para cuidar da criação dos componentes servidores do grid e das tarefas de comunicação dos tópicos JMS (Java Message Service).

Além de usar o servidor de aplicações Java EE, algumas estratégias da última versão da Java EE serviram de modelo para a estratégia de criação de aplicações para este grid, conforme será visto em detalhes na seção 4.1.3 (Uso de POJO).

2.6 Teoria Fuzzy

Uma vez entendido o ambiente no qual aplicações grid são executadas, é necessário entender os elementos da teoria fuzzy que serão utilizados pelo escalonador para lidar com a grande quantidade de variáveis que interferem na performance de um grid.

A teoria fuzzy, descrita por Zadeh no final da década de 60 e cujo conceito foi consolidado em Zadeh (1998) e Zadeh (1996), pretende estender a lógica clássica. Enquanto a lógica clássica atribui às sentenças somente os valores de verdade ou

falsidade, a teoria fuzzy adiciona a possibilidade de associar graus de verdade contínuos a sentenças. Toda sentença, então, segundo a teoria fuzzy, é ao mesmo tempo verdadeira e falsa em algum grau. Nas situações limites, em que os graus de verdade são 0% e 100%, as associações recaem sobre a lógica clássica nos valores de falsidade e verdade, respectivamente.

2.6.1 Variáveis Lingüísticas

Um sistema escrito com base na teoria fuzzy deve lidar com variáveis lingüísticas internamente, embora as entradas e saídas do seu contorno sejam geralmente variáveis numéricas.

Uma variável lingüística é uma etiqueta para valores de um predicado como “quão alto é o objeto”. Os valores que poderiam assumir essa variável seriam, por exemplo, “muito alto”, “não muito alto” ou “baixo”. A imprecisão de uma afirmação como esta é também objeto de estudo da teoria fuzzy.

Para que o seu processamento seja possível, os valores possíveis que uma variável lingüística pode ter devem ser previamente estabelecidos. No exemplo que acabou de ser discutido, são três os valores possíveis da variável lingüística “quão alto é o objeto”.

2.6.2 Regras Fuzzy

As variáveis lingüísticas tornam-se particularmente interessantes quando é possível combiná-las. A operação sobre variáveis em teoria fuzzy é feita pelas regras fuzzy. Essas regras são representadas sobre a forma de tabelas que relacionam as variáveis lingüísticas de entrada com uma conclusão também lingüística.

Por exemplo, poder-se-ia desenvolver um algoritmo que procurasse decidir quanta memória deve ser alocada numa aplicação. A quantidade de memória a ser

alocada pode ser uma consequência das variáveis lingüísticas “quão importante é a aplicação para o usuário” e “quão complexa é a aplicação”.

Uma regra fuzzy, neste exemplo, poderia ser expressa por:

“se a aplicação é muito importante e muito complexa, então muita memória deve ser alocada” (1)

Nessa sentença, os conectores lógicos fuzzy estão em negrito e os valores das variáveis lingüísticas estão sublinhados. A operação principal é do tipo “se ..., então ...”. Neste trabalho, as operações serão sempre deste tipo.

2.6.3 Partições Fuzzy

Predicados como “quão importante é a aplicação” ou “quão alto é o objeto” não têm valor computacional se uma interpretação não for fornecida para que ele possa ser quantificado. Para que seja possível executar computação sobre valores lingüísticos, é necessário um método de conversão de valores lingüísticos para valores numéricos e vice-versa.

Esse método consiste em fixar um conjunto finito de valores lingüísticos possíveis para cada variável lingüística e atribuir a cada valor lingüístico uma curva. Essa curva relaciona cada valor da variável numérica a um grau de verdade.

Na Figura 18, pode-se ver um exemplo de partição fuzzy. Esta partição poderia referir-se ao valor “agradável” para a variável lingüística “quão agradável é a temperatura ambiente”. O eixo das ordenadas representa o grau de verdade e o das abscissas a temperatura ambiente.

Segundo o gráfico, a temperatura entre 20°C e 30°C é certamente agradável. Ou seja, para temperaturas dentro desta faixa, é totalmente verdadeira a frase “a

temperatura ambiente é agradável”. Temperaturas abaixo de 15°C ou acima de 35°C são certamente não-adequadas. As demais temperaturas (entre 15°C e 20°C, e entre 30°C e 35°C), não são simplesmente adequadas ou não-adequadas, mas apresentam uma graduação entre esses dois extremos.

Figura 18 – Exemplo de partição fuzzy

A curva que representaria o valor “inadequada” para a mesma variável lingüística “quão adequada é a temperatura ambiente” poderia ser:

Figura 19 – Partição fuzzy complementar

É interessante observar que a soma dos graus de verdade das duas figuras é 100% para qualquer valor da variável numérica. Combinando duas figuras, observamos que as temperaturas 17,5°C e 32,5°C não são particularmente adequadas

ou desagradáveis. Ou, então, que não é possível decidir se tais temperaturas são ou não agradáveis.

A interpretação sobre o que é uma temperatura “agradável”, assim como a interpretação do que é um servidor “de alto desempenho” são tarefas humanas. A inserção de uma linha de processadores de baixíssimo custo e alta velocidade no mercado poderia alterar a interpretação dos conceitos de “alto” ou “baixo” desempenho dos processadores. De forma similar, uma pessoa acostumada a viver numa cidade perto da Linha do Equador certamente terá uma opinião diferente a respeito do que é uma temperatura agradável em relação a uma pessoa que vive perto do Pólo Norte.

O desenho dessas partições é também influenciado pelo contexto. Por exemplo, atualmente um processador de 400MHz para um computador de mão é considerado rápido. O mesmo processador, se inserido no contexto dos servidores de Internet, seria considerado lento.

Assim, é essencial que um operador humano faça a calibração desses conceitos para que um sistema que usa teoria fuzzy funcione conforme esperado.

O mesmo serve para os valores de saída do sistema fuzzy. Após o processamento das variáveis de entrada, os sistemas fuzzy apresentam um valor numérico de saída que deve ser transformado num valor lingüístico por meio da aplicação de uma curva de partição fuzzy operando na direção oposta.

2.6.4 Conjuntos Fuzzy

Enquanto nos conjuntos clássicos, cada elemento do conjunto universo pertence ou não a cada conjunto definido, no caso de conjuntos fuzzy (ZADEH, 1968), cada elemento do universo apresenta um grau de pertinência a cada conjunto.

Ou seja, todo elemento do universo pertence a todo conjunto fuzzy definido sobre esse conjunto universo em maior ou menor grau. Nas duas situações extremas, nas quais os graus são somente 0% ou 100%, a relação entre elemento e conjunto é a relação clássica.

Por exemplo, se o conjunto fuzzy criado fosse interpretado como o conjunto dos países de grandes dimensões, o grau de pertinência do elemento China a este grupo seria bem próximo de 100%, enquanto que o grau de pertinência do elemento Luxemburgo seria bem próximo a 0%.

As relações entre os elementos do conjunto universo e um conjunto fuzzy podem ser dadas por uma função de pertinência ou por uma enumeração. As funções de pertinência para cada elemento x são do tipo:

$$F(x) : U \rightarrow \{0,1\} \quad (2)$$

A Figura 20 apresenta uma possível representação de um conjunto clássico e um conjunto fuzzy. Os dois conjuntos têm como conjunto universo os pontos de duas dimensões do plano xy . O grau de pertencimento de cada ponto do plano xy é representado por um eixo ortogonal ao plano e cujo intervalo de valores vai desde 0% até 100%.

Figura 20 – Uma possível representação gráfica de um conjunto (a) clássico e um (b) fuzzy

No conjunto clássico, os pontos cujas coordenadas seguem a regra “ $1 < x < 3$ e $1 < y < 5$ ” fazem parte do conjunto, sendo que todos os outros pontos não fazem parte do conjunto. Cada ponto tem, então, um grau de pertencimento igual a 0% ou a 100%.

O mesmo não ocorre no conjunto fuzzy. Existem pontos com qualquer grau de pertinência ao conjunto (b). Por exemplo, o ponto (3, 1) tem grau de pertinência 0%, o ponto (3, 2) tem grau 50% e o ponto (3, 3) tem grau 100%.

2.6.5 Operações com conjuntos fuzzy

As partições fuzzy fazem a interface entre o domínio dos valores lingüísticos e os valores que representam uma grandeza real. Cada valor que uma variável lingüística pode ter é representado por uma partição fuzzy.

Um conjunto fuzzy pode ter a sua expressão de pertinência dada por uma partição fuzzy. Neste ponto de vista, um conjunto fuzzy pode ser entendido como o valor de uma variável lingüística. A variável lingüística em si pode ser entendida como o conjunto universo do conjunto fuzzy.

Desta forma, operações lógicas como E, OU, ou SE...ENTÃO podem ser aplicadas a dois ou mais conjuntos fuzzy. O resultado dessas operações será um outro conjunto fuzzy, que também pode ser expresso por meio de uma partição fuzzy.

Neste trabalho, somente serão usadas relações do tipo SE...ENTÃO. Por este motivo, somente será apresentado nesta seção o procedimento de cálculo para este tipo de relação.

Graus de pertinência também podem ser entendidos como probabilidades. Assim, dizer que “o grau de pertinência do material x dentro do conjunto dos materiais densos é 0,8” equivale a dizer que “a probabilidade que o material x seja classificado no conjunto dos materiais densos é de 80%”.

Raciocinando em termos de probabilidades, o grau de pertinência de um dado elemento a dois conjuntos simultaneamente equivale a determinar a probabilidade de o elemento pertencer aos dois conjuntos simultaneamente. Essa probabilidade deve, então, ser igual à produtória das probabilidades de pertencimento a cada conjunto.

A operação E, então, tem a forma de um produto. De forma similar, a operação OU tem a forma de uma soma ou somatória. Essas duas operações também são aproximadas por alguns autores, respectivamente, para operações “mínimo” e “máximo”. Neste trabalho, usou-se esse último modo de interpretação.

Independente do método utilizado, o resultado de uma operação feita sobre uma partição fuzzy tem como valor de saída uma outra partição fuzzy. Por exemplo, se tivermos as variáveis lingüísticas “temperatura” e “posição do potenciômetro”, que serão chamadas de a e b , e se tivermos respectivamente dois valores (ou partições fuzzy) a_1 e b_1 dados pelas funções:

$$\begin{aligned} a_1(x) : \{0, L_a\} &\rightarrow \{0,1\} \\ a_1 : "agradável" \end{aligned} \quad (3)$$

$$\begin{aligned} b_1(x) : \{0, L_b\} &\rightarrow \{0,1\} \\ b_1 : "não_muito_alto" \end{aligned} \quad (4)$$

A operação que representa a condição “se a temperatura é agradável, então a posição do potenciômetro deve ser não muito alto” pode ser representada por “se a é a_1 , então b é b_1 ”. A operação atua sobre um valor qualquer a_v de a e gera um novo valor b_v de b .

Figura 21 – Aplicação de se...então sobre partições fuzzy

A Figura 21 mostra de forma esquemática como esse cálculo é feito. A partir de uma relação SE...ENTÃO sobre dois valores conhecidos a_1 e b_1 , cria-se o bloco do meio. Ao receber um valor a_v , o bloco trata de processá-lo para gerar o valor b_v . É interessante notar que o valor a_v pode ser qualquer partição fuzzy criada arbitrariamente por um usuário de um sistema que usa teoria fuzzy. No exemplo da Figura 21, a variável poderia ter sido chamada “razoavelmente desagradável”, por apresentar valores extremos e pouca probabilidade de valores intermediários.

A operação usada para o cálculo dentro do bloco do meio será:

$$b_v(x) = \lim_{n \rightarrow \infty} \max(m(x, L_a \frac{1}{n}), m(x, L_a \frac{2}{n}), \dots, m(x, L_a \frac{n}{n})) \quad (5)$$

onde

$$m(x, y) = \min(a_1(y), b_1(x), a_v(y)) \quad (6)$$

e L_a é o valor máximo de x na variável a , ou, de forma equivalente, é o domínio no qual a variável lingüística a é definida. Esta expressão é uma forma contínua da matriz de conormas. Para efeito de cálculo numérico, quanto maior for o valor de n usado, tanto menores serão as possíveis distorções, pois n particiona L_a .

Também é interessante notar que cada uma das variáveis lingüísticas deve ter, tipicamente, um domínio diferente, ou seja, em geral L_a e L_b são diferentes. No exemplo, L_a poderia ser a temperatura de $50^\circ C$ enquanto que L_b poderia ser o ângulo de 360° .

O valor b_v gerado pode ser usado, por exemplo, para um servomecanismo alterar a posição de um potenciômetro de um circuito que regula um termostato.

3 Ambiente de desenvolvimento e execução de grids

Os serviços grid, conforme especificados pela OGSA, são baseados nos *web services*, que por sua vez são baseados em troca de mensagens no formato XML. O protocolo que suporta essa troca de mensagens geralmente é o HTTP. Os *web services*, na forma como foram criados, visam permitir a interoperabilidade de sistemas extremamente heterogêneos, provendo uma infra-estrutura de troca de mensagens em formatos padronizados.

Um efeito negativo da heterogeneidade e dos padrões de fácil adoção dos *web services* é o baixo desempenho destes serviços diante de soluções *ad hoc*. Neste trabalho, propõe-se criar uma plataforma de desenvolvimento de grids para executar aplicações de forma similar ao papel que os servidores de aplicação executam para as plataformas corporativas. Mas neste caso, a G4JEE atua de forma inversa, na qual a aplicação cliente envia o algoritmo que vai ser executado. Grids criados usando-se a G4JEE teriam, portanto, que receber aplicações e fornecer-lhes um ambiente de execução.

Parte deste trabalho visa criar componentes de persistência distribuída transparente para o desenvolvedor de aplicações grid. Essa transparência visa concentrar a atenção do programador nos algoritmos que são desenvolvidos para o grid e não aos aspectos de implementação do próprio grid.

Entende-se que o programador-alvo deste trabalho deseja utilizar os recursos virtualizados de um grid para executar um processo computacional específico e que a principal preocupação desse programador é com o processo em si e não com o funcionamento do próprio grid.

Assim, decidiu-se criar uma estrutura que torne possível o desenvolvimento de aplicações grid sem que o programador tenha que lidar diretamente com uma infraestrutura de grid complexa. A simplicidade do desenvolvimento implica:

- em não ser necessário que o programador desenvolva subsistemas de trocas de mensagens, ou que tenha que desenvolver descritores para que o seu código-fonte seja entendido pelo grid;
- no aproveitamento de investimentos feitos em componentes corporativos já desenvolvidos usando a plataforma Java EE;
- na alteração automática de código-fonte, que transforma os programas para que estes usem o grid de modo a executar métodos remotamente;
- em não ser necessário o investimento em mecanismos de persistência que obriguem o desenvolvedor a lidar com mais de um paradigma ao mesmo tempo (por exemplo, através do uso simultâneo de orientação a objetos e bancos de dados relacionais).

Espera-se que os programadores que usam a G4JEE sigam o modelo de programação para o qual a G4JEE foi concebida. Seguindo esse modelo, então, o programador consegue tirar melhor proveito da ferramenta criada neste trabalho.

Desta forma, a primeira seção deste capítulo explica como é esse modelo. O entendimento desse modelo permite justificar a arquitetura desenvolvida para a G4JEE. A seção seguinte apresenta as ferramentas providas pela G4JEE e a terceira

seção do capítulo detalha o ambiente que executa aplicações criadas com essas ferramentas.

3.1 Ambiente de desenvolvimento

O ambiente de desenvolvimento visa criar uma separação entre os problemas que o programador precisa resolver (a escrita dos seus algoritmos) e toda a infraestrutura complexa de troca de mensagens e escalonamento que suporta a execução dos programas criados por ele.

Tendo em vista que o programador é o usuário principal da ferramenta G4JEE, é interessante entender para qual tipo de desenvolvimento a ferramenta foi escrita. Tal entendimento também é interessante para que adaptações possam ser feitas aos programas sem causar problemas para que a G4JEE distribua o processamento de aplicativos no grid.

3.1.1 Modelo de programação

O modelo de programação utilizado segue o modelo de threads da linguagem Java. Semáforos, threads e outros elementos de computação paralela são de responsabilidade do próprio programador.

Para conseguir que o seu código seja executado mais rapidamente pelo grid, a estratégia básica que o programador deve seguir consiste simplesmente em transferir a execução de um método demorado para outro nó do grid capaz de executá-lo mais rapidamente.

Procedendo desta forma, um algoritmo puramente sequencial como o da Figura 22 (a) tornar-se-ia mais rápido como apresentado na Figura 22 (b). Na Figura 22 (a), o método que está sendo chamado na máquina SA está destacado com a área

pontilhada. O mesmo método, quando é executado na máquina SB é executado mais rapidamente, pois esta máquina deve ter mais recursos que a máquina SA. Durante a execução do método na máquina SB, a máquina SA simplesmente fica esperando o método ser finalizado (representado com a área com linhas diagonais).

Apesar de essa abordagem deixar SA ociosa durante o tempo da execução do método em questão, a abordagem é muito simples de ser implementada. Outro ponto vantajoso dessa abordagem é que outros processos de SA independentes do programa em estudo podem ser executados de forma oportunista durante o tempo em que SA está esperando SB terminar a execução do método.

Figura 22 – O mesmo algoritmo sendo executado (a) normalmente e (b) com o uso de outro computador do grid

Enquanto a Figura 23 (a) mostra o código-fonte original que é executado localmente, a estratégia poderia ser implementada simplesmente pelo código-fonte da Figura 23 (b). Por enquanto basta entender que o código-fonte da figura faz com que o método “demorado()” seja executado pelo grid e não mais localmente. A forma como esse efeito é obtido será estudada mais tarde neste texto.

```

1 package org.aamm.tests;
2
3 public class ProgramaSimples
4 {
5     public static void main(String[] args)
6     {
7         ProgramaSimples ps = new ProgramaSimples();
8         // ...
9         ps.demorado();
10        // ...
11        // ...
12    }
13
14    private void demorado()
15    {
16        // ...
17    }
18 }

```

(a)

```

1 package org.aamm.tests;
2
3 import org.aamm.g4j2ee.rpj.GR;
4 import org.aamm.g4j2ee.rpj.GRemote;
5
6 @GR
7 public class ProgramaSimples implements GRemote
8 {
9     public static void main(String[] args)
10    {
11        ProgramaSimples ps = new ProgramaSimples();
12        // ...
13        try
14        {
15            ps.demorado();
16        }
17        catch(Exception e)
18        {
19        }
20        // ...
21        // ...
22    }
23
24    private void demorado() throws Exception
25    {
26        // ...
27    }
28 }

```

(b)

Figura 23 – Duas versões do mesmo programa que executa o método “demorado()” (a) localmente e (b) remotamente no grid

As regiões pontilhadas da Figura 22 seriam, nesse caso, o tempo demorado para o processamento do método “demorado()”. O ProgramaSimples seria o programa

que lança chamadas para o grid. O elemento central dos grids G4JEE são as aplicações que criam e iniciam os processos que são executados dentro do grid.

O programa da Figura 23 tem o problema de ser seqüencial, e de não explorar, portanto, paralelismo para executar duas tarefas simultaneamente. Por exemplo, se o computador que recebe a tarefa de executar o método “demorado()” tiver a mesma capacidade de processamento que a máquina na qual ProgramaSimples está sendo executado, o tempo de processamento nas duas abordagens será o mesmo.

Se, além disso, o ProgramaSimples não estiver concorrendo com nenhum outro processo local, então o uso do grid não terá trazido nenhuma vantagem em termos de desempenho. Pelo contrário, as chamadas para o grid, por mais rápidas que sejam, causam sempre um pequeno *overhead* ao programa.

É importante notar que a parte do programa que foi executada remotamente está num contexto diferente daquele da aplicação que iniciou o processamento. Por exemplo, o sistema de entrada e saída são diferentes, assim como a interface com o usuário. Por isso, o modelo de programação prevê que o programador restrinja o acesso ao I/O e à interface com o usuário à porção da aplicação que é executada localmente. Caso contrário, o programador fará com que a aplicação use dados inconsistentes de I/O ou que a interface com o usuário tenha seus elementos gráficos aparecendo remotamente em qualquer nó do grid, o que em geral é indesejado.

Para que ocorra paralelismo, o programador deve criar um thread local que fique aguardando a finalização do processamento de um método remoto. Geralmente, após a execução desse método remoto, ocorre a execução de algum procedimento local para fazer a sincronização entre os diferentes threads.

Mesmo nos casos em que um método remoto contém erros que travem o seu processamento, a possibilidade desse travamento não é desconhecida para os programadores Java. A implementação atual dos threads em Java não permite que um thread seja interrompido durante o seu processamento. Implementações mais antigas permitiam que o programador pudesse interromper um thread, mas essa funcionalidade dos threads foi depreciada, pois se descobriu que era com frequência a causadora de deadlocks.

Assim, os programadores Java já estão acostumados a desenvolver a sincronização prevendo que a execução dos threads eventualmente nunca termine. Essa abordagem contrária à interrupção de um thread durante a sua execução é bastante adequada ao modelo de programação da G4JEE, visto que um thread local que dispara um método remoto não tem controle sobre como o método remoto está sendo executado.

Figura 24 – Uma aplicação com dois threads sendo executados não-simultaneamente

Na Figura 25, a tarefa T1 chama a tarefa T2 e continua sendo executada. Durante certo tempo, os dois threads são executados simultaneamente, aumentando o desempenho da aplicação. A tarefa T1 termina a sua execução no ponto marcado pelo triângulo (um ponto de sincronização) e passa a aguardar T2 (região hachurada). Quando T2 termina sua execução e retorna o resultado para T1, essa última tarefa pode continuar o processamento contando com os dados fornecidos por T2.

Figura 25 – A mesma aplicação, com os dois thread sendo executados simultaneamente

O contrário, ou seja, T2 terminar antes de T1, também pode acontecer: a resposta de T2 pode chegar antes do ponto de sincronização. Nesse caso (mostrado na Figura 26), o tempo de execução é mais bem aproveitado, visto que não existe tempo de espera em nenhum dos threads.

Figura 26 – Na tarefa T2 termina a sua execução antes que T1 precise do seu produto

T2 pode ser o processamento de algum cálculo, a medida de um dado em um sensor ou uma busca por alguma informação. T2 pode também chamar outra tarefa T3, como visto na Figura 27. Para T1, T2 funciona como na Figura 25.

Figura 27 – Uma tarefa pode chamar outra sucessivamente

Essa discussão tem profundo reflexo na forma como as tarefas são distribuídas. O servidor que recebe T2 tem total autonomia para decidir como as tarefas iniciadas por T2 devem ser processadas. Assim, esse segundo servidor SB pode enviar T3 para um terceiro servidor SC e assim sucessivamente. Se ocorrer algo que impeça SC de processar a tarefa T3, SB deve repassar T3 para outro servidor. Isso pode ser feito sem que SA tenha conhecimento.

Figura 28 – Situação mais complexa

Por fim, uma situação um pouco mais complexa é apresentada na Figura 28. Nesta figura, pode se ver que do ponto de vista de T1, existem duas tarefas geradas:

T2 e T4. Ao terminar sua execução, T4 envia seu resultado para T1, que por sua vez espera até que T2 tenha terminado a sua execução para continuar seu processamento.

O diagrama da Figura 29, que identifica cada uma das sub-tarefas pode ser redesenhado como na Figura 30, na qual um grafo mostra mais claramente a dependência entre as sub-tarefas.

Figura 29 – A Figura 28 redesenhada para mostrar as pequenas tarefas que compõem cada thread

A execução na Figura 29 também pode ser representada pelo diagrama da Figura 31.

Figura 30 – Diagrama de fluxo de dados equivalente à figura Figura 28

Figura 31 – Diagrama da Figura 30 redesenhado para mostrar os threads

A Figura 32 mostra o esquema de distribuição dessas tarefas, localizando os nós nos quais cada uma é executada.

Figura 32 – Distribuição das chamadas aos métodos entre três computadores

Ao fazer a leitura do valor de uma variável qualquer que não esteja definida estritamente no escopo do método (ou seja, que não seja declarada dentro dele e que não seja um de seus argumentos), o valor da variável é transferido pela rede. Isso ocorre para que o método consiga usar o valor correto dessa variável no processamento do método.

Internamente, a ferramenta G4JEE faz a transferência dos valores das variáveis por meio de uma troca de mensagens. O nó que precisa do valor de uma variável envia uma mensagem para o nó criador solicitando o valor dessa variável. O nó criador, então, envia outra mensagem ao requerente passando o valor da variável.

Analogamente, toda alteração feita em uma variável que não está definida estritamente no escopo do método obriga o grid a transferir o novo valor para o nó que contém a variável originalmente.

Essas duas situações (de leitura e de escrita de valores) devem ser, tanto quanto possível, evitadas para diminuir o custo associado à comunicação de dados. A única

forma de evitar essa transferência de dados é evitar que essas situações ocorram. Dessa forma, o programador que desenvolve usando a G4JEE é incentivado a ler dados somente a partir dos seus argumentos.

Sendo baseado nos conceitos da Java EE, a G4JEE usa o conceito de contêiner para prover serviços para as aplicações que são executadas dentro do grid. Um desses serviços é o escalonador de tarefas, que distribui as tarefas de forma transparente.

Além disso, outros serviços podem ser hospedados dentro do contêiner. Neste projeto, visa-se apenas demonstrar a viabilidade da criação desses serviços. Assim, optou-se por criar-se um serviço de persistência de dados simples.

3.1.1.1 Alertas durante a codificação

Alguns alertas são incluídos durante a escrita do código-fonte. Esses alertas são exibidos para que o programador esteja ciente de que o seu código-fonte está usando uma estratégia que poderia ser melhorada.

Por exemplo, um dos alertas adverte o programador de que um método executado remotamente está tentando alterar o valor de uma variável definida localmente. O alerta informa que tal ação causa a transferência do novo valor da variável pela rede. Recomenda-se, nesse caso específico, que a transferência de dados pela rede seja feita somente no começo (na figura dos argumentos) ou no fim (como o valor de retorno) dos métodos remotos.

Se, assim mesmo, o programador, por algum motivo, deseja ignorar as sugestões para melhorar a performance da aplicação, é possível suprimir os alertas através da anotação dos métodos conforme a linha 12 do código-fonte da Figura 33 .

```

... // ...
6
7 @GR
8 public class Test7 implements GRemote
9 {
10     public static String s = "1234";
11
12     @SuppressWarnings
13     public String getS() throws Exception
14     {
15         return s + " is the string";
16     }
17 }

```

Figura 33 – Exemplo de uso da anotação @SuppressWarnings

3.1.1.2 Mecanismo de feedback

A anotação @GLocal, como já apresentado, faz com que o método seja sempre executado no criador. Essa execução local pode ser interpretada de mais de uma forma. Essa interpretação depende da estratégia usada pelo programador ao construir seu programa distribuído. O entendimento dessa diferença permite que o programador crie uma comunicação entre o método remoto que está sendo executado e o criador.

Para um código que esteja sendo executado localmente, uma chamada a um método marcado como @GLocal deve ser interpretada como uma forma de evitar a distribuição de um método para outro computador no grid.

Já a chamada a um método local, feita por uma rotina que está sendo executada remotamente, faz com que o grid envie uma mensagem ao criador do processo para que o criador execute este método.

Suponha-se, por exemplo, que o método que está sendo executado remotamente demore em geral algumas horas até terminar. Suponha-se, também, que a máquina que criou o processo remoto deseja, de quando em quando, exibir em uma GUI a porcentagem de completude do processo. Nesse caso, o programador pode escrever o

método remoto de forma que ele chame um método local que atualize essa informação na tela do usuário.

Em resumo, esse mecanismo pode ser utilizado para os seguintes propósitos:

- acessar algum recurso centralizado no nó criador, como um arquivo, por exemplo. O acesso a esse recurso pode ser encapsulado em um método;
- notificar o criador a respeito de alguma condição, permitindo que o criador tome decisões com base no andamento da execução do método. O método assim criado pode servir como um agente remoto que coleta informações e as envia para o criador;
- entregar ao criador um resultado parcial da execução do método. Num arranjo produtor-consumidor, esta entrega permite que o criador no papel de consumidor faça o consumo dos resultados mesmo antes do final da produção;
- receber novos dados para execução. Fazer com que o método peça novos dados assim que eles sejam necessários permite que o método inicie seu processamento mesmo sem conter todo o conjunto de dados de que necessita (por exemplo, pelo fato de eles poderem não estar ainda prontos para uso). Esse procedimento, também, permite que o próprio método escolha quais dados ele necessita, pedindo-os para o criador;
- dar oportunidade ao criador de interferir na execução do método. O valor de retorno do método local chamado pode conter uma mensagem do criador informando qual deve ser o próximo passo do método.

3.1.1.3 Variáveis de classe e de instância

Enquanto variáveis locais (definidas dentro do contexto de um método) em métodos remotos são de inteiro controle e visibilidade do próprio método, variáveis de classe (estáticas) e de instância³ (não estáticas) são compartilhadas com outros métodos (variáveis privadas) ou também com outras classes (variáveis públicas, protegidas ou de visibilidade default).

Esse compartilhamento é controlado pelo grid para evitar que cada um dos nós que tem uma referência a essas variáveis tenha um valor diferente de qualquer uma delas. O controle é feito por meio de mensagens JMS que atualizam esses valores entre os nós. Essa troca de mensagens, obviamente, causa uma carga extra ao processamento do grid. O programador deve estar ciente deste efeito para evitar ou minimizar o uso dessas variáveis em nós remotos.

O acesso local a essas variáveis não causa a propagação do valor dessas variáveis para os nós remotos, pois o nó criador mantém a melhor cópia de cada variável. Assim, o acesso local a essas variáveis não cria carga extra no grid.

A implementação alternativa seria criar cópias dessas variáveis em cada nó que executa um dos métodos da aplicação. Essa abordagem simplificaria a leitura das variáveis, mas criaria a necessidade de transferência de mais mensagens para atualizar os valores das variáveis o tempo todo. Escolheu-se, então, incentivar o programador a usar o mecanismo de feedback ao invés de usar variáveis de classe ou de instância para permitir que o programador controle o uso da rede, otimizando a aplicação.

³ Variáveis de instância e de classe são também chamadas de campos de instância e campos de classe. Em inglês o termo “field” é usado no lugar de “campo”.

Assim como em qualquer aplicação multithread, as aplicações multithread distribuídas num grid devem ser criadas tomando-se cuidados para que criados mecanismos de sincronismo entre as rotinas que concorrem por recursos.

Apesar desse controle, o grid não possui gerência sobre os estados internos de classes que tenham sido criadas usando-se um processo de compilação distinto do presente no grid. Assim, se uma classe que contém estados internos, como a da Figura 34, for usada pelo grid, o valor do seu estado não será compartilhado por cada um dos nós do grid. O programador deve estar ciente desse fato para evitar o compartilhamento de informação através desses estados.

```
1 package com.fornecedor;  
2  
3 public class ContadorUtil  
4 {  
5     private static int valor = 0;  
6  
7     public void incrementar()  
8     {  
9         valor++;  
10    }  
11  
12    public int valor()  
13    {  
14        return valor;  
15    }  
16 }
```

Figura 34 – Classe que, se compilada sem o G4JEE, pode causar problemas às aplicações

Ao invés disso, o programador é incentivado a usar variáveis que ele mesmo define e que, sendo definidas em código que será alterado pelos aspectos, serão controladas pelo grid.

Seria possível ter desenvolvido a G4JEE de forma a transferir a responsabilidade pela execução de tais métodos para o nó criador. Mas, nesse caso, os métodos remotos causariam o tráfego de uma quantidade muito grande de

mensagens pela rede e isso eliminaria a vantagem de se executar o processamento distribuído.

3.1.2 Ferramentas de Desenvolvimento

A primeira parte deste projeto, conforme já citado, consiste em um conjunto de ferramentas para o desenvolvimento de aplicações grid. Essas ferramentas visam permitir que uma aplicação qualquer seja convertida numa aplicação que possa ser executada num grid de forma simples.

Para obter esse efeito, diversos aspectos e classes são vinculados, de forma automática, ao código que está sendo escrito pelo programador. A AOP é responsável por fazer a detecção das oportunidades de aproveitamento da infraestrutura do grid. A ferramenta também conta com uma biblioteca de classes que contém toda a inteligência do grid, além de uma API que o programador pode usar para explicitamente invocar os recursos do grid. Essa API fornece recursos para que um programador mais avançado consiga tirar maior proveito do grid.

3.1.2.1 Abordagem para a alteração automática de código

Para a alteração de código foram usados diversos aspectos (conceito já apresentado na seção 2.4). Esses aspectos procuram identificar *pointcuts* nos quais ocorrem chamadas que podem ser distribuídas no grid.

O programador, ao criar a aplicação grid, tem a possibilidade de marcar seu código-fonte. Essas marcações são reconhecidas pelos aspectos, que se encarregam de alterar o código-fonte, acionando a comunicação dentro do grid.

```

1 package org.aamm.g4j2eesampleclients.text;
2
3 import org.aamm.g4j2ee.rpj.GLocal;
4 import org.aamm.g4j2ee.rpj.GR;
5 import org.aamm.g4j2ee.rpj.GRemote;
6
7 @GR
8 public class TimeConsumingTask implements GRemote
9 {
10     private int numero = 0;
11     public String timeConsumingMethod(String a) throws Exception
12     {
13         // código executado remotamente (...)
14         return "blabla";
15     }
16
17     @GLocal
18     public int getNumero()
19     {
20         // código executado localmente (...)
21         return numero;
22     }
23 }

```

Figura 35 – Exemplo de classe que envia métodos para execução remota

A Figura 35 mostra um exemplo de uma classe que distribui um dos seus métodos. A classe passa a ter seus métodos remotos pela inclusão da anotação `@GR`. A anotação `@GLocal` informa aos aspectos que este método deve ser ignorado ao fazer a distribuição no grid, fazendo com que esse método seja executado localmente. Assim, o método `timeConsumingMethod` é sempre executado no grid, enquanto que o método `getNumero` é sempre executado localmente.

Procedendo desta forma, a classe `TimeConsumingTask` torna-se um componente vinculado com o grid. Toda classe que chama o método `timeConsumingMethod` dispara, involuntariamente, um processo no grid.

A execução do método, do ponto de vista do elemento que invoca o método, ocorre como se fosse local. Isso significa que a execução do método bloqueia o thread que a solicitou, como ocorre com a execução de qualquer método local.

Assim, para que a aplicação que chama o método não seja bloqueada e para que ela possa usar o tempo em que fica aguardando o processamento do método, é

interessante que o programador que desenvolve a aplicação coloque a execução do método em um thread separado.

Outro detalhe a ser notado na Figura 35 é a necessidade da cláusula “throws Exception” na Linha 11. Ao usar a G4JEE, a eventual falta desta cláusula nos métodos remotos é apontada como um erro durante a codificação. Ou seja, códigos-fonte sem essa cláusula nos métodos remotos simplesmente não compilam.

O apontamento desse tipo de erro e a não compilação auxiliam enormemente o desenvolvimento dessas aplicações distribuídas. Esse efeito não é obtido normalmente pela programação clássica, mesmo quando se programa para a maior parte das plataformas corporativas.

É necessário que exista essa cláusula para que as rotinas que chamem esses métodos remotos façam o tratamento de exceções. As exceções, quando ocorrem remotamente, são propagadas pela rede, inclusive em cascata.

A invocação de métodos no grid pode ocorrer em cascata. Ou seja, um método remoto pode chamar outro método remoto sucessivamente. Isso permite que sejam criados algoritmos paralelos que distribuem o processamento de forma modular.

3.1.2.2 API para a criação de protocolos

Aproveitando a infra-estrutura de comunicação criada sobre JMS, na qual a atribuição e filtragem por IDs já está resolvida, foi criada uma pequena API que permite ao programador desenvolver o seu próprio protocolo de comunicação sobre o protocolo criado para este grid.

O protocolo deste grid prevê um conjunto limitado de tipos de mensagens representados inteiramente por meio de uma enumeração Java (recurso introduzido na versão 1.5 da linguagem). A enumeração prevê um tipo de mensagem chamada

USER_DEFINED_MESSAGE, cujo tipo é aberto. Este tipo aberto pode receber uma outra enumeração criada pelo programador que desenvolve usando a G4JEE.

Para este envio de mensagens, o programador pode usar o mesmo esquema de IDs que o grid usa. Assim, uma aplicação distribuída pode ser criada sem que o programador precise se preocupar com a comunicação entre os nós.

Também é disponibilizado um serviço que lista os IDs das máquinas que estão conectadas ao grid naquele momento. Esse serviço pode ser usado para identificar quais nós da aplicação distribuída podem ser acessados.

3.2 Ambiente de execução

Um aspecto fundamental do desenvolvimento das aplicações é a independência entre o programa criado e o ambiente de execução. Uma vez desenvolvida a aplicação, a mesma pode ser executada usando-se tanto um ambiente de execução local, quanto um que usa a distribuição no grid.

Assim, não é necessária a presença do grid para fazer testes nos algoritmos que estão sendo desenvolvidos para o grid. Os testes podem ser executados com um ambiente de execução local e o ambiente de execução pode ser trocado para que a aplicação seja executada no grid.

3.2.1 Clientes / nós do grid

A plataforma G4JEE considera todo nó do grid ao mesmo tempo como um cliente dos serviços do grid e um fornecedor desses serviços para o grid. Ou seja, o papel dos nós é ao mesmo tempo pedir a execução de tarefas no grid e oferecer seus recursos para o uso de outros nós no grid.

Num arranjo cliente-servidor, o cliente faz pedidos passando parâmetros e o servidor os executa. O cliente geralmente não tem acesso ao método que o servidor está executando. A responsabilidade sobre a execução do método fica toda a cargo do servidor.

Já no arranjo da G4JEE, um nó executor pode executar remotamente um método cuja definição está presente apenas no nó criador. Quando isso ocorre, o nó criador deve transferir pela rede os *bytecodes* que contém essa definição.

O mecanismo de envio de código torna possível que aplicações desenvolvidas para serem executadas no grid não precisem ser realmente instaladas no grid, tornando o ciclo escrita-teste-correções mais rápido, visto que não é necessária a instalação de pacotes (como os arquivos EAR, WAR ou GAR) dentro do servidor para se testar uma aplicação escrita para o grid. Assim, estimula-se que o usuário do grid possa fazer constantes alterações no software sendo executado pelo grid.

3.2.2 Servidor de aplicações

A parte principal do contêiner criado é um servidor de aplicações Java EE. Para os testes feitos, usou-se o servidor de aplicações *open source* JBoss, disponível em (JBoss 2006), mas outros servidores podem ser usados com o mesmo efeito, já que a aplicação criada sobre o servidor de aplicações segue o padrão Java EE.

3.2.3 Sistema de Troca de Mensagens

A troca de mensagens entre os elementos do grid dá-se através de tópicos JMS. Este sistema de troca de mensagens permite que um grande número de aplicações clientes se conecte ao mesmo tópico e se cadastrem como *listeners* de mensagens.

As mensagens são enviadas de forma assíncrona e a entrega das mensagens é confiável. A API da JMS garante que as mensagens sejam entregues somente uma vez e que a não entrega de mensagens seja sempre notificada ao remetente.

Os sistemas de troca de mensagens são extremamente modulares. Assim, novos componentes podem ser facilmente adicionados aos grids baseados na G4JEE através do uso das classes criadas neste trabalho.

Toda mensagem JMS é composta de duas partes principais: o conteúdo principal da mensagem e as propriedades definidas pelo usuário. Cada tipo de mensagem JMS transporta um tipo específico de conteúdo principal.

Cada propriedade pode ser de um tipo primitivo da linguagem Java, do tipo String ou do tipo Object (tipo genérico). Além de transportar informação da origem para o destino, essas propriedades podem ser utilizadas também para filtrar as mensagens JMS.

Essas propriedades podem ser usadas pelos *listeners* para criar seletores que filtram quais mensagens JMS do tópico subscrito são aceitas e quais são rejeitadas.

Por exemplo, uma mensagem JMS que contém uma propriedade chamada “prioridade” do tipo inteiro e com o valor “1” passaria por um seletor definido pela string “prioridade = 1 **or** prioridade = 2”.

Ao se cadastrar em um determinado tópico, os clientes passam a receber todas as mensagens que qualquer outro cliente enviar para o tópico imediatamente após a postagem.

Sobre o sistema JMS foi criado um protocolo de comunicação contendo as primitivas básicas usadas para a comunicação entre os nós do grid. Um dos tipos de mensagem criado nesse protocolo destina-se a trafegar informação específica da

aplicação criada. Desta forma, o programador pode desenvolver seu próprio protocolo de comunicação entre instâncias da sua aplicação ao mesmo tempo em que usa os recursos do grid.

Um EJB do tipo MDB (Message-driven bean) chamado GridControlerMDB fica sempre conectado ao tópico e é executado dentro do contêiner Java EE do servidor de aplicações. Assim como os clientes, o GridControlerMDB se cadastra como *listener* do tópico. O filtro que o GridControlerMDB usa seleciona as mensagens enviadas para um ID reservado especificamente para o servidor.

Ocorre a criação de novas instâncias do MDB no caso de não haver nenhum MDB instanciado, quer seja porque o tópico acabou de ser ligado, quer seja porque cada um dos MDBs que estavam ativos lançou uma exceção que obrigou o contêiner Java EE a encerrá-los.

Tipicamente, o contêiner Java EE reaproveita um MDB já instanciado para receber novas mensagens do tópico. Mas, no caso de muitas mensagens chegarem simultaneamente, o contêiner trata de criar novas instâncias do MDB para receber as mensagens em paralelo.

Cada nó do grid, ao se conectar ao tópico pela primeira vez, envia uma mensagem de pedido de um ID. O MDB que receber esta mensagem cria um novo ID aleatório, certifica-se de que ele é único e o envia ao nó cliente que solicitou o novo ID. A partir deste momento, o nó sempre se identificará através desse ID.

3.2.4 Banco de dados distribuído

O uso de bancos de dados em sistemas grid foi estabelecido por Chevenak (2001).

Com a finalidade de manter o ambiente de desenvolvimento simples, optou-se por não requerer que o programador tivesse que lidar com uma estrutura de banco de dados estruturado. Ao invés disso, desenvolveu-se um banco de dados orientado a objetos simples baseado em padrões JDBC.

O banco de dados orientado a objetos usa o banco de dados relacional Java DB para executar o armazenamento e as consultas aos dados. O banco de dados Java DB foi escolhido para manter os dados por ser extremamente simples de ser distribuído. A distribuição desse DBMS é composta simplesmente de um arquivo JAR cujo tamanho é somente de dois megabytes, aproximadamente.

Este banco de dados é configurado através de chamadas a métodos específicos⁴ e é composto de um conjunto de classes Java que, baseadas nessas configurações, criam arquivos de dados assim que são chamadas pela primeira vez. Como consequência, o banco de dados não necessita de qualquer procedimento de instalação.

A comunicação com as classes do Java DB dá-se por meio de um JDBC que, ao invés de montar uma conexão de rede, que é o uso mais comum do JDBC, chama classes locais que se encarregam de administrar os dados também locais. Apesar de o banco de dados ser, do ponto de vista do programador de grid, orientado a objetos, internamente, as classes da API do grid lidam com um banco de dados relacional. Essa estratégia foi usada para permitir, em uma futura implementação da G4JEE, que o Java DB seja substituído por outros bancos de dados relacionais, conforme será citado na Seção 5.3.3.

⁴ Em inglês, o termo “programmatically” tem sido usado em situações como esta, nas quais rotinas criadas por um programador configuram um componente criado por outro programador. Este termo é usado em contraposição à abordagem na qual se utilizam arquivos de configuração para alterar o comportamento de componentes.

O uso desse DBMS simples permite que todo nó do grid que tenha espaço de armazenamento em disco possa conter um pequeno banco de dados, contribuindo para que o grid tenha um banco de dados virtual do tamanho equivalente à soma das contribuições individuais de cada nó.

Figura 36 – Estrutura do banco de dados distribuído

Na Figura 36, é mostrada esquematicamente a estratégia usada. Nessa estratégia, cada participante do grid contém, embutido dentro de si, um pequeno banco de dados que é acessado diretamente apenas pela aplicação local.

Figura 37 – Esquema de consultas ao banco de dados distribuído

A Figura 37, por sua vez, representa uma consulta ao banco de dados distribuído. Nesta figura, as setas indicam o fluxo de dados durante a consulta. O processo é iniciado pela máquina A, que faz uma consulta a uma tabela do banco de dados distribuído através de um envio de uma mensagem *broadcast* para os demais nós do grid. No exemplo, o nó D estava fora do grid por algum defeito ou simplesmente por que o nó estava desligado.

A consulta (1) é enviada, então, apenas para dois dos quatro nós do grid e para o controlador do grid. A máquina A faz uma consulta ao seu próprio banco de dados (3) e junta o resultado dessa consulta com os resultados das consultas enviados pelas máquinas B e C (2).

O GridControllerMDB, representado por GC, ao receber a mensagem de consulta ao banco de dados, trata de consultar o seu banco de dados de meta-informações sobre as tabelas distribuídas para enviar essas meta-informações (4)

para que o nó A consiga qualificar o resultado da consulta. Por exemplo, é essencial que o nó A saiba que somente 75% das tabelas foram consultadas na pesquisa feita. Dessa forma, cada consulta ao banco de dados tem associada a si um grau de cobertura.

As aplicações grid que usam esses bancos de dados distribuídos devem estar preparadas para lidar com essas imperfeições o tempo todo. É possível, por exemplo, que a consulta feita na Figura 37 não tenha retornado nenhum resultado para o nó A. É possível, também, que se o nó D estivesse disponível, esse nó retornaria algum registro para A. Então, o retorno de nenhum registro ao nó solicitante, não deve ser interpretado como a não existência de qualquer registro que satisfaz as condições da consulta feita. Essa não existência poderia ser concluída somente se o grau de cobertura for 100%, podendo a consulta ser interpretada como se estivesse sendo feita a um banco de dados relacional centralizado numa única máquina.

Por exemplo, se a aplicação cliente deseja obter a média dos valores de uma determinada coluna do banco de dados distribuído e se a cobertura foi como mostrado na Figura 37 (75% de cobertura), a aplicação cliente deve por conta própria calcular um desvio padrão diante dos resultados. Assim, as consultas a esse tipo de banco de dados devem ser vistas como amostragens, ao invés de observações objetivas.

O grid por si só não é capaz de fornecer esses indicadores estatísticos por não conhecer a interpretação dos dados dessas consultas.

A criação do banco de dados dá-se de acordo com o esquema da Figura 38. Neste esquema, o nó A pede (1) para o controlador do grid criar uma tabela distribuída.

Ao receber essa mensagem, o controlador do grid registra a nova tabela no seu registro interno e depois disso envia uma mensagem para todos os nós do grid ordenando que cada um deles crie no seu banco de dados local uma cópia da tabela tal qual especificada pelo nó A.

Figura 38 – Criação de uma tabela no banco de dados distribuído

A Figura 39 mostra um exemplo de classe Java anotada para definir uma tabela no banco de dados orientado a objetos. Nessa classe, a anotação `@Table` da Linha 11 avisa o grid para tratar essa classe como uma tabela distribuída.

A anotação da Linha 12 contém a declaração de uma restrição para toda a tabela. Tal restrição pode referir-se a mais de uma coluna da tabela. O elemento que lê essa classe trata como colunas todos os campos da classe. Neste caso, os campos “nome” e “dataNascimento” são usados como colunas.

Anotações podem ser associadas a cada campo para alterar a forma como a coluna será criada no banco de dados. Por exemplo, criando uma condição de unicidade ou de não-nulidade do valor.

```

1 package sp;
2
3 import java.util.ArrayList;
4
5 import org.aamm.derby.pojodb.Constraint;
6 import org.aamm.derby.pojodb.NotNull;
7 import org.aamm.derby.pojodb.PersistenceManager;
8 import org.aamm.derby.pojodb.Table;
9 import org.aamm.derby.pojodb.Unique;
10
11 @Table
12 @Constraint("dataNascimento < 50000")
13 public class Pessoa
14 {
15     @Unique
16     @NotNull
17     private String nome;
18
19     @Constraint("dataNascimento > 1000")
20     @NotNull
21     private long dataNascimento;
22
23     private static final PersistenceManager p = new
24     PersistenceManager(Pessoa.class);
25
26     public Pessoa()
27     {
28
29     }
30
31     public Pessoa(long nascimento, String nome)
32     {
33         dataNascimento = nascimento;
34         this.nome = nome;
35     }
36
37     @SuppressWarnings("unchecked")
38     public static ArrayList<Pessoa> findByName(String nome)
39     {
40         return p.select("nome like '%" + nome + "%'");
41     }
42
43     public static Pessoa findById(long id)
44     {
45         return (Pessoa)p.select("id = " + id).get(0);
46     }
47
48     public String getNome()
49     {
50         return nome;
51     }
52
53     public long getDataNascimento()
54     {
55         return dataNascimento;
56     }
57 }

```

Figura 39 – Exemplo de POJO que descreve uma tabela

A partir da classe da Figura 39, cria-se automaticamente o comando SQL da Figura 40. Nota-se que não existe um campo ID na classe Pessoa. Esta coluna é criada automaticamente pela G4JEE para toda tabela gerada a partir de uma classe.

A criação e manutenção desses IDs são de responsabilidade do grid, da mesma forma que os endereços dos objetos no *heap* são de responsabilidade da JVM. O paralelismo entre esses dois modelos também é observado no que concerne à transparência das suas responsabilidades.

```
1 CREATE TABLE sp_Pessoa
2 (
3     ID BIGINT PRIMARY KEY,
4     nome VARCHAR(500) UNIQUE NOT NULL,
5     dataNascimento BIGINT NOT NULL
6     CONSTRAINT dataNascimento_CHECK CHECK (dataNascimento > 1000),
7     CONSTRAINT sp_Pessoa_CHECK CHECK (dataNascimento < 50000)
8 )
```

Figura 40 – Comando SQL criado automaticamente a partir da classe da Figura 39

3.2.5 Tolerância a falhas

As falhas em sistemas grid presumivelmente podem ocorrer a qualquer momento, dada a grande quantidade de pontos de falha devida à complexidade da infra-estrutura que comporta o grid.

Do ponto de vista de distribuição de processamento, os sistemas grid devem ser capazes de tolerar a não entrega de tarefas solicitadas. Isso requer que os sistemas grid possam perceber que algo não está ocorrendo de acordo com o esperado. Por exemplo, se uma tarefa solicitada começa a demorar para ser entregue, o solicitante deve pedir informações a respeito do estado da execução da tarefa.

Caso o próprio status não esteja disponível, o solicitante toma a iniciativa de pedir novamente a execução da tarefa para outro servidor.

A tolerância a falhas foi implementada através de threads que são responsáveis por, periodicamente, fazer o salvamento do estado de objetos em arquivos. O salvamento é feito após a conversão dos objetos em um vetor de bytes (processo conhecido como *serialization*).

São feitos salvamentos sucessivos em arquivos diferentes em um esquema de rodízio para evitar que uma eventual falha durante a serialização dos dados consiga corromper o único arquivo que contém o estado. No esquema de rodízio, mesmo que essa falha ocorra, haverá outros arquivos intactos para uma eventual reconstituição do estado após o retorno à execução.

Quando a aplicação volta a ser ligada, o módulo na aplicação cliente responsável pelas tarefas do grid tenta recuperar o último estado salvo. Se este estado estiver corrompido, a aplicação tenta carregar o segundo estado mais recentemente salvo e assim por diante até encontrar algum estado válido. No caso de nenhum estado estar disponível, esse módulo cria um novo objeto, solicitando um novo ID para o servidor.

Esse estado pode conter uma aplicação que foi interrompida no meio de uma execução. Se isso ocorrer, o módulo grid vai tentar continuar a execução da tarefa no ponto em que ela estava quando ocorreu o salvamento do estado.

Este mecanismo serve para que seja evitado que se perca um investimento de tempo na execução de um procedimento demorado.

3.2.6 Monitoramento de recursos

Para usar de forma otimizada os recursos de um grid, é essencial conhecer com detalhes as condições nas quais os elementos do grid estão sendo executados.

Mas, como visto, as características de performance das redes grid dependem de uma quantidade muito grande de fatores. Estudar esses fatores dinamicamente é uma tarefa muito complexa. Além disso, uma vez escolhidas as variáveis que serão monitoradas, essas variáveis devem receber um tratamento específico para redes grid que considere a natureza dinâmica e imprevisível dessas redes.

Os contêineres grid, como o Globus (GLOBUS, 2006), geralmente oferecem alguns recursos avançados para o monitoramento das condições de trabalho servidores grid e reserva de recursos (SMITH et al., 2000). Apesar disso, os serviços grid que usam carga de processamento de tarefas complexas precisam de um conhecimento muito mais aprofundado sobre como um servidor está sendo executado e, o que geralmente é mais importante, saber qual é o perfil de performance que o servidor vem apresentando ao longo do tempo.

Para isso foi criada a PDI (Performance Data Interface) como parte deste trabalho de pesquisa (MATSUI; SATO, 2005). Esta ferramenta visa unificar a troca de informações de performance entre serviços grid e clientes de serviços grid de forma histórica e tolerante a falha.

Outras abordagens para o problema já foram feitas (RIPEANU, 2001) (LEE, 2003) (YANG, 2003), mas o modelo temporal usado aqui ainda não tinha sido estabelecido.

Na PDI, os dados de performance ficam disponíveis em tabelas e os programadores que desenvolvem serviços grid podem desenvolver novas unidades de medida de performance, permitindo assim que algoritmos para resolver problemas específicos sejam desenvolvidos.

Por exemplo, tome-se uma aplicação grid que está sendo desenvolvida para uma organização virtual composta de 50 instituições e na qual cada instituição colabora com um servidor. Nesse caso, certamente a comunicação entre as instituições será um fator importantíssimo para a performance do grid. O programador que desenvolve essa aplicação distribuída pode, usando a PDI, definir uma nova variável que representa a confiabilidade das comunicações entre as 50 instituições. Um algoritmo específico pode ser desenvolvido para procurar caminhos mais confiáveis no grafo formado pela rede de interconexões.

Uma aplicação cliente pode ler periodicamente os valores da variável criada para medir a qualidade da rede e usar essa informação para direcionar a carga de processamento para os ramos mais confiáveis.

Todas as variáveis compartilham algumas características como historicidade, método de acesso, cálculo de médias, desvio padrão entre outras. Implementar essa parte comum a todas as variáveis é papel da API fornecida como parte da PDI.

Na G4JEE foram usados os principais conceitos estudados durante o desenvolvimento da PDI, tais como o conjunto de primitivas necessário para a coleta constante de características de performance e o esquema de predição de carga.

3.2.7 Predição de carga

Uma vez monitorado o valor de uma variável de performance usando esse método, é possível recorrer a procedimentos de identificação de comportamentos para prever os valores futuros das variáveis. A efetividade desse tipo de método de predição foi verificada em (YANG et al., 2003).

Essa predição de carga, aliada a um mecanismo como o proposto em (LEE, 2003) e (RIPEANU, 2001) é capaz de minimizar o tempo ocioso do grid.

Para efeito desta pesquisa, foram adotados os modelos temporais semanal e diário, mas outros modelos de identificação de comportamento mais complexos podem ser desenvolvidos.

Figura 41 – Esquema para cálculo de aderência a modelos temporais

A Figura 41 mostra a abordagem usada para a identificação da periodicidade da carga de cada nó. Na figura, $x(t)$ é o valor de cada medida de carga em cada instante t . Cada modelo se traduz em um valor de T . Por exemplo, um modelo semanal terá um T igual a uma semana. Os intervalos de largura T são alinhados, para efeito de cálculo, conforme mostrado na Figura 42.

Figura 42 – A mesma figura redesenhada para facilitar o entendimento do raciocínio

Neste arranjo, são calculados os desvios padrão de cada corte vertical na pilha de curvas. Cada corte vertical corresponde a um valor de g . Cada g cria uma coleção de valores de x , que na figura é representada à direita da pilha de gráficos. Representa-se esta coleção de valores da variável x tomada a cada T intervalos de tempo, e iniciando a partir do instante g usando o símbolo $c(x, g, T)$.

$$c(x, g, T) = \{x(g), x(g + T), x(g + 2T), \dots\} \quad (7)$$

Quanto mais diferentes entre si forem os valores de x dentro de $c(x, \mathcal{G}, T)$, tanto menor será a chance de esses valores representarem a mesma condição de uso de um determinado computador. Ou seja, tanto menor será a chance de a variável x se repetir a cada T unidades de tempo, considerando a fase $\mathcal{G}$. O cálculo fica completo quando consideramos todos os $\mathcal{G}$ dentro do intervalo de tempo T . Na Figura 42 o gráfico de baixo mostra os desvios padrões dessas coleções em função de cada $\mathcal{G}$.

Mas, para comparar o modelo T com outro modelo (ou seja, com outro valor de T) a fim de descobrir qual modelo melhor representa a periodicidade com que a utilização de um computador se repete, deve-se chegar a um número que sintetize a informação da última curva. Isto é feito usando-se uma integral em função de $\mathcal{G}$ e normalizando-se a curva para que a largura T não influencie na comparação. Ou seja, isto é feito criando-se uma média contínua dos desvios padrões. Portanto, o cálculo de aderência a cada modelo é dado pela expressão:

$$A_T(x) = \frac{\int_{\mathcal{G}=0}^T s(c(x, \mathcal{G}, T)) \partial \mathcal{G}}{T} \quad (8)$$

O desvio padrão s é dado por:

$$s_n(x) = s(\{x_1, x_2, \dots, x_n\}) = \sqrt{\frac{\sum_{j=1}^n (x_j - \bar{x}_n)^2}{n-1}} \quad (9)$$

onde $\bar{x}_n$ é a média aritmética de $\{x_1, x_2, \dots, x_n\}$.

De forma prática, a integral da expressão 8 precisa ser aproximada pela somatória:

$$A_T(x) \approx \sum_{\mathcal{G}=1}^m s(c(x, \mathcal{G}, T)) \quad (10)$$

na qual m é o número de divisões de T feito para a medição, considerando que as medidas sejam feitas em instantes espaçados uniformemente.

A decisão a respeito de qual é o melhor modelo para representar a periodicidade do perfil é feita escolhendo-se o modelo que possui o menor valor de $A_T(x)$.

Escolhido este modelo, o perfil de utilização do computador pode ser dado, por uma curva que apresenta a média de cada coleção em função de $\mathcal{G}$, como mostrado na Figura 43.

Figura 43 – Obtenção do perfil de uso de um computador

O algoritmo usado armazena e recalcula dinamicamente os valores de $s(1,T), s(2,T), \dots, s(m,T)$, assim como o valor de $A_T(x)$ para todas as variáveis x e para todos os modelos T . Assim, todo nó está sempre pronto a fornecer dados sobre sua operação a qualquer outro nó que os requeira. A necessidade dessa abordagem para o monitoramento de aplicações grid foi demonstrado pelos resultados do estudo (ZHANG 2003).

Já que os nós do grid devem ficar constantemente avaliando os modelos, as médias e os desvios padrões precisam ser recalculados sempre. Para isso, os cálculos usados foram recursivos, respectivamente, nas seguintes formas:

$$\bar{x}_{n+1} = \frac{n\bar{x}_n + x_{n+1}}{n+1} \quad (11)$$

$$s_{n+1}(c(x, \mathcal{G}, T)) \approx \sqrt{\frac{ns_n^2(c(x, \mathcal{G}, T)) + [x(\mathcal{G} + nT) - \bar{x}_{n+1}]^2}{n}} \quad (12)$$

Nota-se que o cálculo do desvio padrão é aproximado, pois usa médias antigas. Ou seja, o cálculo desse desvio padrão é tanto menos acurado quanto mais a média das medidas variar. Assim, o erro do desvio padrão pode ser estimado sem precisão, para efeito de comparação, por:

$$E(s) \propto \int \frac{\partial \bar{x}}{\partial t} dt \quad (13)$$

No caso de ocorrerem erros nas medidas como na área hachurada da Figura 44, pelo mau funcionamento de algum agente que é responsável pela medida ou pelo fato de o computador ter sido desligado, por exemplo, objetos representando medidas nulas são inseridos na tabela de medidas.

Figura 44 – Erros nas medidas

Para os cálculos de médias e desvios padrões, essas medidas de erro são ignoradas. O cálculo usado para $A_T(x)$ não contabiliza as medidas nulas, mas essas medidas de erro são usadas para julgar a respeito da confiabilidade do nó.

3.2.8 Exemplo

É interessante analisar um exemplo prático para ilustrar como a predição de carga auxilia na tarefa de aproveitar da melhor forma possível um grid. Tome-se então um pequeno grid composto por três servidores M1, M2 e M3 que são compartilhados com aplicações para usuários finais durante as oito horas de um turno de trabalho. Esse grid será utilizado para executar oito tarefas dependentes entre si.

Figura 45 – Dependências entre tarefas para o exemplo

A Figura 45 apresenta as dependências entre as tarefas. Por exemplo, só é possível iniciar T3 depois que T1 e T2 estiverem finalizados.

Figura 46 – Perfis de carga dos servidores considerados

Os perfis de carga simulariam três servidores em três fusos horários diferentes e distantes oito horas entre si, de forma a não coincidir seus períodos de carga de trabalho conforme a Figura 46.

A Figura 47 apresenta uma simulação para três casos de execução: (a) sequencial, no qual somente o servidor M1 sem carga adicional é usado; (b) paralelo sem histórico, no qual os três servidores são usados em conjunto, mas ignora-se quando cada servidor será utilizado pela carga adicional; e (c) paralelo com histórico, no qual os três servidores são utilizados simultaneamente e as informações históricas sobre a carga adicional é usada para tornar a distribuição de tarefas mais eficiente.

Figura 47 – Comparação entre os desempenhos seqüencial e paralelo

No caso paralelo sem histórico (b), desconhecendo que M2 receberia carga logo após, o escalonador atribuiu a tarefa F5 a essa máquina, causando um atraso em T7 e T8.

Já o escalonador que usa o histórico percebe que atribuir qualquer tarefa a M2 logo após o processamento de T3 acarreta um atraso. Assim, ao invés de passar qualquer tarefa a M2, o escalonador decide enviar T5 a M3 e esperar até que M1 fique livre para enviar M6, eliminando a situação de espera.

3.3 Balanceamento de carga usando a teoria Fuzzy

O problema de executar aplicações de alta performance num ambiente dinâmico e bastante imprevisível, como o grid, é abordado pela G4JEE através de um escalonador fuzzy. Este escalonador executa como parte do Message Driven Bean que fica acoplado ao tópico JMS dentro do servidor de aplicações. Toda vez que um nó cliente deseja pedir para o grid executar um de seus métodos, este nó recorre ao escalonador para solicitar uma sugestão de nó para o qual o pedido deve ser enviado.

3.3.1 Controle fuzzy

O controle fuzzy, tal qual estabelecido por (DRIANKOV et al., 1995) e (JANTZEN, 1998), pode ser feito de forma similar ao controle em feedback, como nos controladores PID (Proporcional Integral Diferencial), por exemplo. O controle do G4JEE é feito segundo mostrado na Figura 48 e é inspirado no trabalho (LEE, 1990/I) (LEE, 1990/II).

Figura 48 – Diagrama do controle fuzzy

As medidas de desempenho chegam como valores numéricos de variáveis escalares. Esses valores numéricos são convertidos para valores de variáveis lingüísticas pelo bloco de fuzificação. Uma vez representada como variável lingüística, a informação está no domínio fuzzy, dentro do qual toda variável é do tipo lingüística e as operações são feitas sobre conjuntos fuzzy, os quais serão introduzidos a seguir.

O bloco seguinte combina as medidas de desempenho com a saída do sinal de controle fuzzy. Essas variáveis combinadas refletem o quão longe de uma meta está o controle fuzzy feito.

A seguir, o bloco fuzzy avalia esta medida de erro com a ajuda dos parâmetros configurados pelo operador do grid. Esses parâmetros são a interpretação do operador do grid para cada uma das variáveis lingüísticas que ele controla. A avaliação deste bloco tem como resultado um conjunto de parâmetros para o escalonador.

O feedback encarrega-se de fechar o ciclo, permitindo que novos valores de medidas de desempenho do grid possam ser combinados com os parâmetros que o escalonador usa, a fim de criar novos parâmetros para o escalonador a cada ciclo.

3.3.2 Modelo de carga usado

Para o modelo, admitiu-se que organização virtual é um conjunto de servidores em domínios administrativos distintos entre si. A rede que interconecta as instituições apresenta erros aleatórios. Os recursos são compartilhados com aplicações de maior prioridade que interrompem aleatoriamente o trabalho do grid.

Já que a comunicação é instável, um determinado nó pode continuar processando uma tarefa mesmo que a comunicação seja interrompida. Quando a comunicação for restabelecida, o nó que estava processando a requisição atualiza o status e o cliente do processamento, caso já tenha tentado fazer o repasse para outro servidor, pode liberar esse outro servidor, poupando esforço repetitivo.

3.3.3 Algoritmo Fuzzy para escalonamento

O escalonamento das aplicações grid equivale à decisão entre um ou outro servidor que receberá uma carga. Para isso, métodos usualmente empregados em computação distribuída não podem ser utilizados devido à grande quantidade de variáveis fora do controle que influenciam na performance, como já visto.

Por outro lado, não é factível considerar todas as variáveis que têm influência no desempenho de cada servidor, pois as variáveis são em grande quantidade e freqüentemente não podem ser medidas por uma aplicação que está sendo executada no grid. Não é possível, por exemplo, prever qual é a taxa de utilização de um determinado servidor que é usado também compartilhado com outras aplicações servidoras além das tarefas do grid. A qualquer momento, um novo cliente pode

começar a fazer muitas solicitações às outras aplicações com as quais as aplicações grid concorrem, fazendo com que o tempo disponível para a execução das tarefas do grid se torne muito menor.

3.3.3.1 Expressão para decisões de carga

Na solução desenvolvida, usou-se a seguinte expressão:

$$N_s = d_s \times \sigma_d + c_s \times \sigma_c + t_s^{-1} \times \sigma_{t^{-1}} \quad (14)$$

onde:

- d_s representa o nível de dedicação do nó S e é dado pela fração de tempo que o nó fica dedicado exclusivamente para tarefas do grid;
- c_s é a confiabilidade do servidor S , proporcional à fração de serviços bem entregues em relação ao número de serviços pedidos;
- t_s^{-1} é o inverso do tempo de resposta médio do servidor S obtido por observações históricas (período de resposta médio);
- σ_d , σ_c e $\sigma_{t^{-1}}$ são, respectivamente, os pesos de d_s , c_s e t_s^{-1} para o cálculo. Estes pesos variam com o tempo e são fornecidos pelo grid para o nó que executará a escolha.

É importante notar que os pesos σ_d , σ_c e $\sigma_{t^{-1}}$ são independentes do nó. Ou seja, os mesmos pesos são usados para todos os nós do grid e refletem a estratégia do escalonador fuzzy para o balanceamento de carga.

Procedendo conforme (14), quanto maior N_s , tanto melhor são as condições do nó S para processar tarefas. Ou seja, o nó que possui o maior N_s deve ser o nó que recebe carga.

3.3.3.2 Estimação de d (dedicação)

O nível de dedicação d pode ser obtido pelo cálculo:

$$d = \int_{g-T}^g n(\rho) d\rho \quad (15)$$

onde T é o período considerado, g é o instante do cálculo, $n(\rho)$ é a quantidade de processos concorrentes no instante ρ .

Na prática a integral é aproximada para a seguinte somatória:

$$d \approx \sum_{i=m-p}^m n_i \quad (16)$$

onde m é o índice da última medida feita, p é o número de medidas que serão consideradas no cálculo (o cálculo é feito, então, com base nas p últimas medidas) e n_i é a i -ésima medida feita. p é tal que todas as medidas n_i consideradas foram obtidas entre o momento do cálculo e T unidades de tempo no passado.

3.3.3.3 Estimação de c (confiabilidade)

Para estimar c , deve-se estimar a taxa de erros, que são eventos pontuais. Diferente do cálculo da dedicação, o cálculo da confiabilidade não pode se valer de uma integral no tempo, pois a confiabilidade só pode ser medida quando existe demanda. A expressão usada para c é, então:

$$c = \frac{k(T)}{k(T) + l(T)} \quad (17)$$

onde $k(T)$ e $l(T)$ são, respectivamente, o número de pedidos respondidos corretamente e incorretamente durante as últimas T unidades de tempo. Mesmo em situações de baixa demanda, os pedidos teste, que serão descritos a seguir, podem ser usados para estimar c .

3.3.3.4 Estimação de t^{-1} (tempo de resposta)

A estimação de t^{-1} é obtida fazendo-se pedidos periódicos de serviços, os quais serão chamados de serviços-teste, com tempo de processamento interno nulo e considerando-se a média aritmética desses tempos de resposta. Assim, tem-se:

$$t^{-1} = \left(\frac{\sum_{i=m-p}^m t_i}{p} \right)^{-1} \quad (18)$$

onde t_i é o i -ésimo tempo de resposta medido.

3.3.4 Variáveis Lingüísticas de Entrada

Serão tomadas como variáveis fuzzy de entrada a flutuação na taxa de pedidos de tarefas e a predominância de pedidos de transferência de dados.

3.3.4.1 E_1 - Flutuação na taxa de pedidos de tarefas

Obtida através da fuzificação da variável e_1 , desvio padrão da taxa de pedidos, dada por:

$$e_1 = \frac{\sum_{i=m-p}^m |f_i - \bar{f}|}{p} \quad (19)$$

onde f_i são as medidas de quantidade de pedidos por unidade de tempo e $\bar{f}$ é a média dessas taxas no intervalo entre as medidas $m - p$ e m .

Serão utilizados os níveis de quantização:

“Alto” – Imprevisível

“Médio” – Instável

“Baixo” – Estável

3.3.4.2 E_2 - Predominância de pedidos de transferência de dados ou processamento de dados

E_2 será traduzida para a quantidade de dados que devem ser trafegados (quantidade alta ou baixa).

Para se obter esta variável, cada serviço será associado previamente com uma porcentagem de tempo g que na execução deste se gasta para fazer a transferência de dados. A porcentagem de tempo gasto para o processamento dos dados será, então, $(1 - g)$.

A seguir, observa-se a demanda e registram-se todos os pedidos feitos, obtendo-se a média dessas porcentagens.

Assim, a expressão de e_2 , variável que será usada para gerar E_2 , será:

$$e_2 = \frac{\sum_{i=m-p}^m g_i}{p} \quad (20)$$

Para a variável E_2 , serão usados os níveis:

- “Alto” – Serviço apenas de transferência de dados;
- “Médio” – Serviço de transferência e busca ou filtragem de dados.;

- “Baixo” – Serviço no qual se transfere uma quantidade não apreciável de dados, mas no qual se pede o processamento dos mesmos;
- “Muito Baixo” – Serviço no qual se pede o processamento de alto desempenho e a quantidade de dados é pequena.

Para as variáveis linguísticas de entrada, usam-se respectivamente as curvas da Figura 49.

Figura 49 – Curvas de partição fuzzy das variáveis de entrada E_1 (a) e E_2 (b).

3.3.4.3 Regras Fuzzy

A representação das regras fuzzy será feita por meio das Tabelas 1, 2 e 3. Nelas, os valores das variáveis de entrada E_1 e E_2 são relacionados com o valor v da variável de saída S . Assim, a leitura das tabelas deve ser:

$$\text{“Se } E_1 \text{ é e } E_2 \text{ é, então } S \text{ é”} \quad (21)$$

Tabela 1 – Regras para o nível de dedicação σ_d

σ_d		E_1		
		Alto	Médio	Baixo
E_2	Alto	Médio	Médio	Médio
	Médio	Médio	Médio	Médio
	Baixo	Alto	Médio	Baixo
	Muito Baixo	Alto	Alto	Médio

Tabela 2 – Regras para a confiabilidade σ_c

σ_c		E_1		
		Alto	Médio	Baixo
E_2	Alto	Médio	Alto	Alto
	Médio	Baixo	Médio	Alto
	Baixo	Médio	Médio	Médio
	Muito Baixo	Médio	Alto	Alto

Tabela 3 – Regras para o tempo de resposta $\sigma_{t^{-1}}$

$\sigma_{t^{-1}}$		E_1		
		Alto	Médio	Baixo
E_2	Alto	Alto	Alto	Alto
	Médio	Médio	Médio	Alto
	Baixo	Baixo	Baixo	Baixo
	Muito Baixo	Baixo	Baixo	Baixo

Por exemplo, na

Tabela 1, a seguinte regra poderia ser lida:

“Se a flutuação na taxa de pedido de serviços (E_1) for alta, e a predominância de pedidos de transferência de dados (E_2) for média, então o nível de dedicação d deve ser considerado de forma média.” (22)

Cada uma das três Seções seguintes explica os princípios usados para a obtenção das Tabelas 1, 2 e 3, respectivamente.

3.3.4.4 Princípio para a Dedicção (*d*)

O nível de dedicação de um servidor deve ser considerado com grande importância quando os pedidos são predominantemente relacionados com alto processamento. Neste caso, quanto maior for a imprevisibilidade com que os pedidos chegam, tanto mais a dedicação do servidor deve ser considerada, pois muitos pedidos simultâneos causam um grande *overhead* ao serem instanciados.

Por outro lado, se a demanda por tarefas for de alta transferência de dados, a dedicação do servidor não é um fator tão preponderante para a sua escolha, visto que altas transferências de dados têm geralmente como fator limitante a rede e, em alguns casos, o acesso a disco, ficando a disponibilidade do processador, claramente, como um fator de pouca importância.

3.3.4.5 Princípio para a Confiabilidade (*c*)

Quanto à confiabilidade, tem-se que para volumes muito grandes de dados transferidos, um servidor confiável é estritamente necessário. Visto que a transferência de volumes grandes de dados dá-se durante um período de comunicação entre origem e destino dos dados, se o servidor envolvido não for confiável, a transferência dos dados torna-se virtualmente impossível.

Para pedidos de execução de rotinas que consomem muito poder de processamento, também se deseja que o servidor que a processe tenha alta confiabilidade, pois se um dos sub-processos não for entregue, este deverá ser pedido novamente, provavelmente para outro servidor, o que causa uma redução no desempenho de todo o processo.

Com relação à previsibilidade do fluxo de pedidos, quanto mais alta for a imprevisibilidade dos pedidos, tanto menos sentido existe em se cobrar uma alta

confiabilidade dos servidores, já que estes podem diminuir sua confiabilidade de acordo com a taxa de pedidos simultâneos.

3.3.4.6 Princípio para o Tempo de Resposta (t^{-1})

Quanto ao tempo de resposta, se a demanda é por muita transferência de dados, um tempo de resposta alto pode significar uma conexão de rede lenta. Isso torna o tempo de resposta lento para toda a transferência de dados.

Para transferência maciça de dados, não se deve considerar o fator previsibilidade, pois as transferências podem durar um tempo suficiente para que o grau de previsibilidade mude.

Se a demanda é por tarefas de alto processamento, o tempo gasto com a transferência de dados, processos de negociação inicial (*handshaking*) inerentes ao protocolo de comunicação são irrelevantes, principalmente quando não se pode prever qual será a taxa de pedidos. Se não se pode prever qual será a taxa de pedidos, faz ainda menos sentido considerar este fator para discernir entre servidores.

3.3.4.7 Curvas para as Partições Nebulosas

Figura 50 – Partição fuzzy das variáveis de saída

Para todas as variáveis de saída, a partição fuzzy usada teve o contorno dado pela Figura 50. Cabe ao administrador do grid desenhar as curvas utilizadas na prática para o grid que ele está administrando.

Assim, o administrador executará as funções de um especialista, e fará o ajuste fino dos parâmetros quando o grid for posto em funcionamento pela primeira vez ou quando a rede sofrer uma mudança estrutural drástica.

Através da observação do desempenho da rede, e acompanhando-se a coleta de dados para os parâmetros σ_d , σ_c e $\sigma_{t^{-1}}$, o administrador pode inferir como serão as curvas que dizem o que é uma dedicação, uma confiabilidade ou uma frequência de resposta muito baixa, baixa, média ou alta.

4 Implementação e resultados

Apresentada a estratégia escolhida para o sistema criado, pode-se discutir agora em detalhes os problemas que foram resolvidos para se obter o produto deste trabalho. Este capítulo prepara a discussão deste texto para o capítulo final, no qual se faz a conclusão do trabalho.

4.1 Implementação

Nesta seção, serão mostrados os detalhes mais importantes do sistema desenvolvido. O objetivo será elucidar como, de forma prática, foi desenvolvida a plataforma desse trabalho, quais foram os problemas encontrados ao se escrever a plataforma e como esses problemas foram corrigidos.

4.1.1 Orientação a Aspectos

A versão atual da linguagem AspectJ é fortemente voltada para as situações de desenvolvimento nas quais as classes Java e os aspectos são escritos simultaneamente. Um uso diferente consistiria em preparar aspectos para serem aplicados em classes que serão escritas por outros programadores. Essa segunda abordagem pode ser usada para criar componentes reutilizáveis baseados em aspectos. Esses componentes podem ser distribuídos para serem usados em outros projetos, alterando classes que serão escritas no futuro.

A falta de prioridade no segundo enfoque trouxe dificuldades durante o desenvolvimento da G4JEE, uma vez que as ferramentas de desenvolvimento dessa plataforma são baseadas em aspectos cuja finalidade é alterar uma aplicação genérica, fazendo com que ela possa ser executada pela G4JEE.

A linguagem AspectJ não permite a manipulação das definições dos *pointcuts* via programação. Ou seja, as definições dos *pointcuts* só podem ser alteradas durante a escrita dos mesmos, da mesma forma como o nome de uma classe só pode ser alterado durante a escrita da classe. Isso torna impraticável, por exemplo, desenvolver um aspecto que transforme automaticamente o código-fonte da Figura 51 no da Figura 52. Também é impraticável, usando AspectJ, verificar, durante a escrita de uma aplicação, as falhas da Figura 51, ou seja, não é possível verificar a falta dos quatro métodos da Figura 52.

É importante salientar que essas verificações podem ser feitas em tempo de execução, usando reflexão sobre as classes. Mas o intuito da ferramenta que se desenvolve nesse projeto é auxiliar o programador ao longo da escrita do código. A ferramenta deve mostrar para o programador as inconsistências do seu código ou completar o seu código automaticamente tornando-o consistente antes que ele seja compilado.

```
1 @JavaBean
2 public class Registro
3 {
4     private long id;
5     private String descricao;
6 }
```

Figura 51 – Java Bean incompleto

```
1  @JavaBean
2  public class Registro
3  {
4      private long id;
5      private String descricao;
6
7      public String getDescricao()
8      {
9          return descricao;
10     }
11     public void setDescricao(String descricao)
12     {
13         this.descricao = descricao;
14     }
15     public long getId()
16     {
17         return id;
18     }
19     public void setId(long id)
20     {
21         this.id = id;
22     }
23 }
```

Figura 52 – JavaBean da Figura 51 completado

Da mesma forma, a linguagem AspectJ não é capaz de criar um novo método em cuja assinatura exista referência à classe do método. Essa dificuldade tornou o sistema criado menos funcional do que inicialmente previsto ao requerer do programador a escrita de métodos que poderiam ter sido implementados automaticamente pelos aspectos.

Assim, não foi possível desenvolver um aspecto que cuidasse de inserir automaticamente todos os métodos necessários para uma classe Java que representa uma tabela. Por exemplo, não foi possível desenvolver um método `findByID()` conforme destacado na Figura 53, pois, conforme visto, a linguagem AspectJ obriga que os métodos criados tenham toda a sua assinatura definida durante a escrita dos aspectos.

```

1  @JavaBean
2  public class Registro
3  {
4      private long id;
5      private String descricao;
6
7      public String getDescricao()
8      {
9          return descricao;
10     }
11     public void setDescricao(String descricao)
12     {
13         this.descricao = descricao;
14     }
15     public long getId()
16     {
17         return id;
18     }
19     public void setId(long id)
20     {
21         this.id = id;
22     }
23
24     public static Registro findById(long id)
25     {
26         return PersistenceManager<Registro>.findById(id);
27     }
28 }

```

Figura 53 – O método em destaque não pode ser inserido numa classe automaticamente pelo uso de aspectos

Desta forma, se um método chamado `findById()` fosse inserido nas classes-tabela usando *inter-type declarations*, obrigatoriamente, todos esses métodos deveriam ter a mesma assinatura. Esse efeito é indesejado, pois o método `findById()` de cada classe deve retornar um valor do tipo da própria classe. Ou seja, uma classe chamada `Registro`, deveria ter o método com a assinatura

```
public static Registro findById(long id)
```

enquanto que uma classe chamada `Texto` deveria ter um método com a assinatura

```
public static Texto findById(long id)
```

Ao invés do método destacado na Figura 53, no qual o tipo de retorno depende do tipo da classe no qual o método é criado, a implementação do método teve que ser

aquela exibida na Figura 54. A implementação final retorna um objeto do tipo `Object`, o que obriga a classe que chama o método `findById()` a fazer um *cast* para o tipo correto. Ou seja, essa implementação obriga o invocador do método a correr o risco de fazer um *cast* para o tipo errado, já que a linguagem Java não tem meios de verificar se o *cast* feito disparará uma exceção do tipo `ClassCastException` ou não.

```
1  @JavaBean
2  public class Registro
3  {
4
5      // (...)
6
7      public static Object findById(long id)
8      {
9          return PersistenceManager.findById(this.getClass(),id);
10     }
11 }
```

Figura 54 – Método tal qual usado na implementação deste projeto

O mesmo tipo de problema foi encontrado em outras partes do sistema. Apesar disso, o uso da orientação a aspectos ainda foi muito vantajoso para a captura de chamadas a métodos e acessos a variáveis.

O suporte à edição de código-fonte provido pela AJDT também ainda é limitado. Mesmo com essa limitação, ainda foi possível desenvolver alguns aspectos que avisam ao programador que o seu código pode estar usando uma prática não muito recomendável. Mensagens de aviso foram inseridas por meio da declaração “warning”, conforme pode ser visto na Figura 55. As mensagens entre aspas são exibidas dentro do Eclipse durante a edição do código fonte, advertindo o programador a respeito de algumas consequências do seu estilo de programação.

As declarações da Figura 55 fazem com que o Eclipse mostre as mensagens entre aspas, ao lado das linhas nas quais são feitas alterações nos valores de variáveis estáticas ou de instância, respectivamente. Esses alertas também são exibidos durante

a fase de compilação, mesmo que o programador não esteja usando o IDE Eclipse e decida realizar a compilação manualmente.

```

1 declare warning: set(static * (@GR *).*) && !withincode(@GLocal * (@GR
→ *.*)*(..)) && withincode(* (@GR *).*(..)): "Explicit writing of a static
→ variable within a remote method causes the network to transfer this value.";
2
3 declare warning: set(* (@GR *).*) && !set(static * (@GR *).*)
→ && !withincode(@GLocal * (@GR *).*(..)) && withincode(* (@GR *).*(..)):
→ "Explicit writing of a creator's variable within a remote method causes the
→ network to transfer this value.";

```

Figura 55 – Declarações que criam avisos para o programador

Para fazer com que os métodos executem remotamente, foi usado um *pointcut* do tipo *around*, que executa um *advice* no lugar da execução do método.

Além disso, a AOP foi bastante útil para executar a inclusão de objetos no registro. A inclusão de objetos no registro foi feita através de um aspecto do tipo “after” aplicado ao construtor desses objetos. Ou seja, sempre que um objeto de uma classe que implementa *GRemote* é instanciado, o seu próprio construtor se encarrega de incluí-lo no registro. Novamente, esse efeito ocorre sem a intervenção do programador.

Sem a AOP, o registro desses objetos teria que ser feito durante a criação de métodos remotos. Isso implicaria em uma série de verificações dentro do registro para evitar duplicidades, o que tornaria o algoritmo muito mais complicado.

4.1.2 Classloader para grid

Um dos recursos da linguagem Java que a torna prática para o uso na computação distribuída é a modularidade dos *classloaders*, conforme pode ser entendido em (MEYER, 1997) ou (TRAVIS). Um *classloader* é um componente que

é responsável em encontrar definições de classes para software sendo executado numa JVM.

Conforme visto na Seção 3.1.1, um cliente que usa um grid baseado na G4JEE envia para o grid os métodos que devem ser executados. Por esse motivo, foi necessário implementar um novo *classloader* para o ambiente de execução da G4JEE. Esse *classloader* deveria ser capaz de fazer com que instâncias da JVM conseguissem executar métodos de classes que estão definidas somente remotamente em outros nós do grid.

Basicamente, um *classloader* recebe pedidos de definições de classes e retorna objetos do tipo `Class` para o requerente. Para simplificar a tarefa de se escrever um *classloader*, a linguagem Java permite o uso do *classloader* básico que já vem nela embutido. Assim, um novo *classloader* pode ser construído se interpondo entre o *classloader* básico, que usa métodos nativos e se comunica de forma íntima com a JVM, e as classes da aplicação sendo executada. O novo *classloader* pode reimplementar a forma com que os *bytecodes* são encontrados.

Toda vez que o *classloader* da G4JEE recebe um pedido de definição de uma classe que ainda não está dentre as classes já carregadas, este *classloader* verifica primeiro o seu `CLASSPATH` à procura dessa classe. Caso a classe solicitada também não esteja presente no `CLASSPATH`, o *classloader* envia uma mensagem para o criador da execução do método repassando a solicitação.

O nó criador do método, ao receber a solicitação encaminhada trata de procurar o arquivo de terminação “.class” contendo os *bytecodes* da classe em questão, empacota esses bytes e os envia de volta para o *classloader* solicitante. O *classloader* da G4JEE, então, usando um método do *classloader* básico da Java SE, carrega os

bytecodes na JVM e passa uma referência a um objeto do tipo *Class* para a aplicação sendo executada remotamente.

Internamente, o *classloader* tem um cache de *bytecodes* de classes identificados pelos nomes das classes respectivas. Ao definir uma classe qualquer, o *classloader* aproveita a oportunidade de ter acesso aos *bytecodes* e guarda esses *bytecodes* no seu cache, evitando que da próxima vez que ele precisar desses *bytecodes* ele tenha que recorrer novamente ao nó que possui a definição da classe.

O cache impede, no entanto, que novas versões de uma classe que eventualmente apareçam durante a execução do grid sejam usadas pela JVM que executa os métodos remotamente. Essa situação é comum principalmente durante o desenvolvimento das classes, quando o programador está alterando constantemente as classes e executando testes. Para resolver esse problema, a G4JEE permite que o operador do grid desligue o cache.

4.1.3 Uso de POJO

POJO (Plain-old Java object) é o nome da abordagem usada atualmente pela Java EE, para simplificar o desenvolvimento de aplicações corporativas. Essa abordagem contrapõe-se à abordagem inicial da Java EE que usava arquivos descritores em XML. Esses arquivos XML descreviam os componentes EJB.

Essa forma de desenvolvimento de aplicativos demonstrou ser na prática pouco eficiente para o desenvolvimento de aplicativos, pois obrigava o programador a criar manualmente os arquivos XML compatíveis com os nomes das classes, com os nomes dos métodos e com as assinaturas dos métodos. Isso ocorre, pois num ambiente de desenvolvimento de aplicações para produção, as definições dos

componentes corporativos mudam freqüentemente durante o transcorrer do desenvolvimento.

Não demorou para que formas mais eficientes de desenvolvimento de componentes EJB (como o XDOCLETS, por exemplo) aparecessem. As abordagens que usam um único arquivo como fonte para todo o componente reduzem a margem de erro que atrasa o desenvolvimento de componentes e dificulta a sua alteração posterior.

A Sun decidiu, então, corrigir uma falha de design do processo de desenvolvimento e adotar a centralização de código, criando o conceito de POJO, divulgado por meio de (DEMICHIEL; RUSSEL, 2006), ao mesmo tempo que a nova versão da linguagem passou a permitir o uso de anotações nas classes e nos seus membros. A nova versão da especificação EJB (EJB, 2005) já incorpora esses conceitos.

Na nova versão da plataforma Java, as anotações foram incorporadas como um novo recurso nativo da própria linguagem. Isso permite que novas anotações sejam criadas para a linguagem. O acesso a essas anotações pode ser feito por meio de uma extensão do mecanismo de reflexão da linguagem Java.

A mesma abordagem foi usada na plataforma de desenvolvimento desse projeto. Foram criadas novas anotações específicas para marcar métodos remotos. A presença de anotações é detectada pelos aspectos desenvolvidos para este sistema.

4.1.4 GridControlerMDB

O GCMDB (GridControlerMDB) é um MDB (Message-Driven Bean). Ou seja, é um EJB que fica conectado ao tópico JMS e é administrado pelo contêiner J2EE. Os GCMDBs recebem mensagens enviadas para um ID reservado ao servidor.

Visto que o contêiner do servidor de aplicações tem autonomia para criar novas instâncias do GCMDB, esse EJB não pode basear o seu funcionamento em membros de instância. Ao invés disso, é necessário armazenar quaisquer dados num local único, compartilhado por todas as instâncias do GCMDB. O armazenamento está sendo exibido na Figura 56, com o nome de “diretório do grid” e foi implementado usando o DBMS Java DB.

Figura 56 – Arquitetura do tópico JMS e o papel do GridControlerMDB

A Figura 56 também mostra a estrutura da comunicação JMS. Na figura, três máquinas A, B e C estão conectadas ao tópico JMS. Dentro do servidor de aplicações, existem duas instâncias do GCMDB. CCT refere-se ao ClientCheckThread, um thread cuja função é verificar periodicamente quais clientes estão conectados ao tópico.

A única forma de identificar se um cliente está ou não conectado ao tópico é enviar uma mensagem para este cliente e esperar uma resposta desse cliente. Essa verificação é feita por um thread chamado `ClientCheckThread`, instanciado pelo `GridControlerMDB`. É importante que exista somente um `ClientCheckThread` por servidor de aplicações. Se houver mais de um desses threads, os clientes do grid serão verificados mais do que o necessário. Quanto maior o número desses threads, maior é a quantidade de verificações que seriam feitas a cada cliente.

Para assegurar a unicidade desse thread, assim que uma instância do thread é criada, essa instância faz um teste para verificar se existe algum outro thread escrevendo no espaço de memória compartilhada. Se houver, então essa nova instância encerra a sua atividade, ficando disponível para o *garbage collector* da JVM. Se não houver, essa instância se assume como sendo a instância única e passa a varrer o registro de nós periodicamente testando se existe conexão dos nós com o tópico ou não.

4.1.5 Sistema de troca de mensagens

Nessa estrutura de comunicação, toda mensagem enviada por um nó potencialmente pode chegar a todos os outros nós. Cada nó cria um seletor de mensagens que trata de descartar as mensagens que não são destinadas a si mesmo. O seletor é criado durante a construção de um objeto do tipo `TopicSubscriber` (classe da Java EE). Não é possível, depois de criado este objeto, alterar o seletor. Outro seletor só pode ser criado se outro `TopicSubscriber` for criado. Esta limitação permite que o servidor que mantém o tópico aplique os filtros ele mesmo, reduzindo a quantidade de mensagens trafegadas no tópico.

O servidor de aplicações JBoss, usado neste trabalho para administrar o tópico JMS, sempre desmonta toda mensagem enviada para analisá-la e tomar decisões sobre o seu destino e consistência. No início do desenvolvimento da G4JEE a abertura das mensagens pelo servidor que intermedeia a troca de mensagens causou erros na comunicação.

Esses erros ocorriam, pois ao abrir uma mensagem que contém um objeto, a JVM do JBoss pode não reconhecer a classe do objeto pelo fato de ela ter sido definida em um dos nós do grid. De fato, isso ocorre frequentemente, já que geralmente o grid é usado para executar algoritmos criados pelo usuário. Como não é possível alterar o *classloader* do JBoss para que ele use a mesma estratégia usada no *classloader* da G4JEE, a única forma de trafegar em mensagens do tipo Object os objetos de classes definidas pelo usuário é fazendo com que ele seja transferido serializado na forma de um vetor de bytes.

Sendo um vetor de bytes, nenhuma JVM tentará fazer o carregamento desses bytes interpretando-os como *bytecodes* Java a não ser que a JVM seja explicitamente instruída a fazê-lo. Assim, não haverá tentativa de chamar o *classloader* para carregar a classe do objeto e o erro não ocorre.

O vetor de bytes é criado assim que a invocação a um método é empacotada para ser enviada pela rede. A invocação ao método enviada pela rede é um objeto do tipo PackedMethod e o vetor de bytes é um dos campos desse objeto.

4.2 Resultados obtidos e análises

Durante os testes feitos usando o esquema de desenvolvimento criado neste trabalho, verificou-se que o processo de programação fica muito mais simples ao se usar a ferramenta da G4JEE. Como esperado, o ciclo de teste-execução-correção se

mostrou bastante rápido e tão prático quanto o mesmo ciclo para o desenvolvimento de qualquer aplicação Java SE.

4.2.1 Testes unitários

Para o desenvolvimento da G4JEE foram usados testes unitários JUnit (HUNT; THOMAS, 2003) (JUNIT, 2006). Desenvolveram-se testes de dois tipos:

1. **testes do funcionamento das classes principais** – testes desenvolvidos para testar isoladamente o comportamento de cada classe principal da plataforma;
2. **testes do funcionamento do grid** – testes que fazem chamadas ao grid e testam as reações do grid a cada chamada. Estes testes foram executados em um grid composto de três nós. Os testes são:
 - a. um método local invocando um método remoto
 - b. um método remoto lendo uma variável de instância compartilhada
 - c. um método remoto alterando o valor de uma variável local compartilhada
 - d. um método remoto invocando um método local
 - e. um método remoto alterando o valor de uma variável estática local
 - f. um método remoto invocando um método estático local
 - g. um método remoto lendo o valor de uma variável estática
 - h. um método remoto disparando uma exceção
 - i. um método remoto disparando uma exceção em cascata pelo efeito do disparo de uma exceção em outro método remoto

A versão final da G4JEE passou em todos os testes unitários desenvolvidos tanto na plataforma Windows quanto usando Linux. Os testes também foram executados usando um esquema de *mock objects* (THOMAS; HUNTE, 2002) e o resultado deste segundo teste também foi bem sucedido para todos os testes.

4.2.2 Teste de estresse

Além destes testes unitários, foi executado um teste no qual um cálculo de soma de progressão aritmética foi feito de forma recursiva entre os valores 1 até 1000, obrigando o grid a manter 1000 threads executando numa pilha de chamadas.

O teste foi executado com três nós no grid, cada um dos nós solicitou simultaneamente que o grid fizesse o cálculo, ocupando o grid com até 3000 threads em espera.

A funcionalidade do grid para este teste foi verificada com sucesso. O teste também foi capaz de avaliar o custo da comunicação entre os nós, uma vez que o cálculo solicitado foi simples a ponto de praticamente não consumir tempo do grid. Verificou-se que existe um custo na comunicação e que este custo é elevado quando a aplicação executa muitas trocas de mensagens.

Portanto, grids criados com a plataforma G4JEE servirão para executar métodos que requerem muito processamento e nos quais ocorre pouca comunicação entre processos executando paralelamente.

4.2.3 Testes de performance

Foram executados testes de performance para verificar a efetividade da estratégia de escalonamento deste trabalho. Estes testes foram executados alternadamente em grids homogêneos e heterogêneos. Também foram feitos testes de performance que medem o overhead da comunicação dentro do grid.

O algoritmo escolhido para o teste foi o algoritmo de criação de mapas temáticos. O objetivo desse algoritmo é gerar uma imagem que representa, em cada ponto, a intensidade da influência de um determinado fator que ocorre aleatoriamente e de forma pontual.

Por exemplo, um mapa temático de densidade demográfica associa a cada ponto dentro de uma área retangular um escalar que representa a ocorrência de pessoas morando ao redor daquele ponto. Para se obter o escalar de um determinado ponto, toma-se todos os pontos a uma distância inferior a um raio r . De posse desse conjunto de pontos, calcula-se o escalar por uma fórmula do tipo:

$$f(p) = \sum_{i=1}^n \frac{1}{d(p, p_i)^2 + \varepsilon} \quad (23)$$

onde $f(p)$ é o valor do escalar referente ao ponto p , $d(p, p_i)$ é a distância entre os pontos p e p_i , $\{p_1, p_2, \dots, p_n\}$ é o conjunto dos n pontos que influenciam o ponto p (ou seja, são os pontos que estão a uma distância de p inferior ao raio r).

Já que o cálculo é feito com base em pixels, ou seja, já que as coordenadas são quantizadas, sempre ocorre de o mapa temático ser calculado sobre pontos nos quais existe a ocorrência do evento estudado, o que faz com que a distância $d(p, p_i)$ seja nula. Assim, de forma prática, é usado um ε pequeno e arbitrário para evitar que a fração tenha um valor infinito nesses casos.

Figura 57 – Resultados de um teste com 4 frames em um grid composto de computadores Intel P4 3GHz 1GB RAM

A Figura 57 mostra um teste feito usando um grid formado por quatro computadores idênticos. Os quatro computadores eram equipados com o processador Pentium 4, de 3GHz e contavam com 1GB RAM.

O teste executado foi o cálculo de mapas temáticos de 1000 por 1000 pixels com 4 frames a respeito de 200 ocorrências espalhadas aleatoriamente dentro da área. Cada frame é um mapa temático independente dos demais, cada um com um raio de influência diferente dos demais e igual a 50, 100, 150 e 200 pixels.

Figura 58 – Desempenho de um cálculo recursivo local executado em Athlon XP 2400+ 1.99 GHz 768MB RAM

A Figura 58 mostra um teste diferente daquele que se acabou de mostrar. O teste cujo resultado está mostrado na Figura 58 consistiu em um cálculo recursivo de uma progressão aritmética de 1000 termos sendo executado usando um único computador que era ao mesmo tempo cliente e servidor do grid. Esse computador contribuiu com 1 a 5 nós virtuais para o grid.

Pelo fato de o cálculo ser extremamente simples, o teste é capaz de mostrar o overhead da comunicação no grid. Nota-se pelo resultado que existe um overhead mínimo de aproximadamente 25000ms nesse cálculo e que o aumento no número de nós causa um aumento proporcional no tempo gasto com a comunicação.

Ou seja, o custo do envio e recepção de cada mensagem é de aproximadamente 25ms.

Figura 59 – Desempenho de um cálculo em um grid heterogêneo calculando 24 frames

Por fim, a Figura 59 mostra a performance de um teste no qual um grid composto de 3 computadores de 64 bits dual core foi usado para calcular mapas temáticos. No caso desse teste, para aumentar o tempo de processamento, foram calculados 24 mapas temáticos com raios de influência variáveis.

Já que todos os processadores possuem dual core, o desempenho da aplicação melhorou até o ponto em que cada computador contribuiu com dois nós virtuais para o grid (6 nós virtuais no total). Essa interpretação dos resultados explica o resultado do último teste, no qual cada computador contribui com 3 nós virtuais para o grid.

Nesse último teste, o desempenho de cada computador foi inferior, pois os dois cores eram obrigados a administrar 2 nós virtuais.

Também é interessante notar que nesse teste o desempenho do grid foi, no segundo e no terceiro testes, um pouco melhor do que o resultado esperado. Isso ocorre, pois o resultado esperado leva em conta a experiência do usuário da máquina que executou o primeiro teste. No segundo e no terceiro testes foram inseridas no grid máquinas mais potentes do que a máquina do primeiro teste.

5 Conclusões

5.1 Conclusões do trabalho

Concluiu-se que existem grandes vantagens em se usar uma plataforma corporativa, como o Java EE, para o desenvolvimento de grids. Além disso, a escolha de se utilizar uma linguagem de programação específica para todas as aplicações de um grid apresentou uma série de vantagens com relação à abordagem que usa interoperabilidade de aplicações com um protocolo genérico, como o WSDL.

A primeira vantagem clara é a facilidade de se desenvolver a aplicação em uma única linguagem de programação. Se essa linguagem de programação for orientada a objetos, como no caso da escolha feita nesse trabalho, a transferência de dados pelo grid se dá sem a necessidade de conversão de tipos, remontagem de estruturas de dados ou limitações causadas por uma estrutura de dados fixa.

Sendo possível trafegar no grid um objeto Java pertencente a qualquer classe, cria-se um ambiente flexível, no qual as tarefas que são nele executadas podem ser criadas durante a execução do próprio grid.

A desvantagem dessa abordagem de se usar uma linguagem única é justamente a dificuldade dos grids baseados na G4JEE de trabalharem em conjunto com outros grids. Assim, os grids G4JEE devem ser considerados grids limitados aos casos em que o desenvolvimento das aplicações grid é feito usando a linguagem Java.

5.2 Contribuições do trabalho

Neste trabalho, criou-se um ambiente funcional para o desenvolvimento de aplicações distribuídas em um grid. O ambiente de criação de aplicações é totalmente acoplado com a infra-estrutura de grid criada neste projeto.

Assim mesmo, a abordagem usada no ambiente de programação pode ser aproveitada para a criação de aplicações para outros ambientes de execução de aplicações distribuídas em grids ou clusters em redes locais.

Neste trabalho, criou-se uma estratégia de escalonamento que procura ser realista quanto aos problemas encontrados pelas aplicações grid utilizadas efetivamente (em contraposição às aplicações escritas com finalidade de desenvolver e discutir academicamente o próprio conceito de computação grid). O escalonador também é realista com relação ao número de variáveis que interferem na execução do grid, provendo uma abordagem baseada na teoria fuzzy para tratar esse problema.

Foi criada, também, uma estratégia baseada na AOP para o desenvolvimento rápido de aplicações para o ambiente de execução proposto, permitindo que aplicações grid sejam desenvolvidas de forma ágil, característica que corresponde às necessidades muito dinâmicas dos usuários dos grids.

O ambiente de desenvolvimento criado neste trabalho entrega todos os demais resultados deste trabalho para o programador, sem requerer que esse programador tenha que aprender um novo tipo de tecnologia. Ao mesmo tempo, esse ambiente de desenvolvimento fornece ao programador ferramentas para construir aplicações distribuídas contando com uma persistência também distribuída.

5.3 Trabalhos futuros

Durante o desenvolvimento desse trabalho, novas oportunidades de melhoria na plataforma G4JEE foram detectadas. Essas oportunidades não foram exploradas, pois o esforço deste trabalho foi concentrado na exploração da AOP e da teoria fuzzy visando solucionar os problemas infra-estruturais básicos dos grids G4JEE.

Continuando o desenvolvimento iniciado por esta pesquisa, trabalhos futuros podem explorar essas lacunas, as quais serão enumeradas a seguir:

5.3.1 Escalonador

O escalonador usado deste trabalho poderia ser modular, permitindo que outros escalonadores com outras abordagens sejam usados. Novos escalonadores se beneficiariam de toda a infra-estrutura criada pela G4JEE, como a alteração automática de código e o subsistema de troca de mensagens, por exemplo.

Outra alteração que pode ser feita no escalonador pode ser uma melhoria no valor de E_2 usando a abordagem para prever o tráfego de dados que foi proposta em (SCHOPF; VAZHKUDAI, 2002).

5.3.2 Sistema de troca de mensagens

A comunicação utilizada se baseia em apenas um servidor JMS, criando um ponto central de falha. Implementações que conectam vários tópicos JMS criando uma estrutura de troca de mensagens mais tolerante a falhas poderiam ser construídas futuramente. Essa abordagem também pode ser usada para criar *middlewares* entre diferentes grids.

Outra abordagem possível utilizaria tópicos JMS redundantes para eliminar o ponto central de falha.

5.3.3 Banco de dados

A implementação do banco de dados distribuídos deste trabalho foi criada visando somente o uso do princípio de localidade e o estudo das conseqüências da distribuição num grid de um banco de dados orientado a objetos. Assim, o banco de dados criado ainda carece de muitas características novas para se tornar completamente funcional.

As tabelas distribuídas ainda não possuem várias funcionalidades que as tornariam virtualmente uma única tabela. Por exemplo, uma coluna marcada com “unique” não apresenta uma restrição de unicidade entre diversos nós do grid. A unicidade, nesta implementação, funciona simplesmente no contexto de cada nó isoladamente. Uma implementação mais completa demandaria algum tipo de sincronismo entre tabelas. Este sincronismo entre tabelas pode se tornar bastante complexo num ambiente grid, pois cada nó pode ficar ligado ou desligado aleatoriamente.

Ainda com relação às tabelas distribuídas, a abordagem deste trabalho usa o banco de dados Derby (Java DB) somente. Sabe-se, no entanto, que este banco de dados é bastante simples se comparado com outras alternativas comerciais como o DBMS Oracle, por exemplo. Assim, trabalhos futuros nessa linha de pesquisa podem tornar possível o uso de outros DBMSs através de um arquivo de configuração, por exemplo.

O uso de outros DBMSs tornaria os grids montados usando este sistema mais interessantes, pois o DBMS Derby, da forma como foi usado neste projeto, permite que apenas o grid acesse seus dados, bloqueando os *datafiles* para outros processos. Se for usado um DBMS de rede (como por exemplo, o banco de dados Oracle,

MySQL ou mesmo o Derby DB no modo servidor de rede), outros processos poderão inserir dados nas tabelas do grid enquanto o grid está em funcionamento. As tabelas usadas pelo grid poderiam ser, também, *views* de outras tabelas, apresentando para o grid dados consolidados provenientes de processos complexos de obtenção desses dados.

Como já mencionado neste texto, grids são naturalmente heterogêneos. Essa heterogeneidade certamente é também observada nos DBMSs disponíveis. Assim, um mecanismo de suporte a vários tipos de DBMSs deve ser implementado futuramente para tornar a solução de grid mais completa.

Figura 60 – O grid pode conter vários tipos de arranjo de banco de dados, inclusive concorrentes com outras aplicações fora do grid.

Esse cenário é mostrado pela Figura 60, na qual é representada uma situação heterogênea em relação aos DBMSs. Na figura, o nó A contém um DBMS interno,

como o Java DB. Os nós B, C, D e E usam um DBMS exterior ao grid. Os nós C e D, na figura, compartilham um mesmo DBMS. Finalmente, o nó E compartilha um DBMS com outra aplicação não pertencente ao grid.

Além da persistência dos dados em um DBMS, pode ser criada futuramente uma persistência de dados usando arquivos contando com algo como um `GridInputStream` para leitura de arquivos remotos.

5.3.4 Atualização das ferramentas acompanhando as mudanças que eventualmente ocorrerão na linguagem AspectJ

Tanto a linguagem AspectJ quanto a ferramenta AJDT estão ainda em evolução. Ao se experimentar usar a AJDT, torna-se claro que esta ferramenta ainda está sendo desenvolvida e a documentação disponível deixa claro que há intenção de continuar-se a desenvolvê-la.

Novas versões destas ferramentas devem surgir com um suporte mais refinado à criação e ao uso dos aspectos. Então, é interessante num trabalho posterior aproveitar as mudanças futuras nessas ferramentas para ampliar a atuação da orientação a aspectos durante o desenvolvimento das aplicações grid.

5.3.5 Segurança

O enfoque deste projeto foi a distribuição da computação e pouca atenção foi dada ao aspecto de segurança da troca de mensagens. Ainda assim, por usar tópicos JMS, ao subsistema de troca de mensagens pode ser incluído um mecanismo de segurança JAAS. O mecanismo de autenticação pode também ser usado para identificar usuários do grid. Uma futura implementação da G4JEE pode ter um serviço de diretório que mapeie os IDs de cada nó com um usuário que pode estar autenticado nele.

5.3.6 Compatibilidade com a OGSA

Um aspecto negativo da abordagem escolhida é o não uso dos padrões da OGSA. Uma forma de aproveitar melhor o desenvolvimento de *engines* grid que tem sido feito em torno da OGSA, a comunicação JMS poderia ser trocada pelos serviços grid. Para isso, a troca de objetos serializados, que é a base da distribuição de tarefas deste trabalho, deveria ser feita usando um serviço grid específico.

5.3.7 Mecanismos de notificação e reserva de recursos

Os mecanismos de notificação e de reserva de recursos não foram implementados na plataforma criada neste trabalho, embora, como visto durante a discussão sobre o conceito de grid, estes mecanismos devam estar presentes nas estruturas consideradas como grid. Mesmo não contendo esses dois mecanismos, a plataforma deste trabalho foi concebida visando o desenvolvimento desses dois mecanismos em desenvolvimentos futuros.

O mecanismo de notificação pode ser implementado baseado no esquema de comunicação que já existe, pelo fato de esse mecanismo já ser bidirecional e baseado em *listeners*.

A reserva de recursos seria mais difícil de implementar que a notificação, pois o controle dos recursos não é feito de forma centralizada, mas sim feita individualmente por cada nó.

5.3.8 Otimização do uso da memória

A implementação atual da G4JEE usa um registro contendo referências a todos os objetos GRemote criados na JVM. Este registro serve para encontrar objetos a partir dos quais são executados métodos locais chamados remotamente e dos quais

são acessadas variáveis de instância. Essa execução e esse acesso permitem que ocorra o efeito feedback e que o nó criador mantenha a cópia oficial das variáveis.

Apesar desse efeito positivo para o desenvolvimento do grid, esse registro cria também um risco no nó criador. Como o registro mantém uma referência a todo objeto GRemote, o programador que use a G4JEE e que queira enviar algum objeto GRemote para o *garbage collector* tornando-o não-referenciado não será bem sucedido. Em resumo, o registro torna todo objeto do tipo GRemote não-descartável pelo *garbage collector*.

Há, então, o risco de que muita memória seja consumida para a manutenção desses objetos. Esse risco é pequeno se o programador não tráfegar objetos muito grandes pelo grid.

Uma implementação futura do G4JEE deve ter o seu próprio mecanismo de *garbage collector* que verifique se algum nó do grid contém uma referência para o objeto e que remova este objeto do registro caso não haja qualquer referência.

5.3.9 Cache do *classloader*

O cache do *classloader* também necessita de melhorias, para ser capaz de redefinir dinamicamente uma classe já usada. É possível, e bem provável nos ambientes de desenvolvimento, que uma mesma classe seja alterada durante a execução do grid. O grid deve ser capaz de executar essa classe, mesmo que ela mude ao longo do tempo.

Para que isso seja possível, o grid deve ser capaz de limpar os *bytecodes* que contém uma definição antiga de uma classe. Limpo o cache, um pedido para a mesma classe obrigará o *classloader* a buscar a definição de classe novamente no nó que a contém.

Uma implementação dessa política de descarte de *bytecodes* pode fazer com que todo nó mantenha um banco de dados contendo as datas das definições de classe. Assim que essas datas forem alteradas, o nó se encarregaria de enviar para os demais nós do grid uma mensagem informando que existe uma nova versão da classe.

5.3.10 Codificação

O processo de desenvolvimento pode ser melhorado através da criação de modelos. A plataforma Eclipse permite que se criem *wizards* facilmente. Seria, então, simples criar dentro do Eclipse um comando que crie um projeto de aplicação escrita usando a G4JEE. Este projeto automaticamente criado pode já conter suporte à linguagem AspectJ e incluir o JAR contendo as definições dos aspectos da G4JEE.

Outra melhoria pode ser feita incluindo-se instrumentação no grid. Isso permitiria que o programador verificasse se o seu código está usufruindo o grid conforme esperado.

5.3.11 Objetos remotos

Pode ser desenvolvido um novo recurso para a distribuição da computação através da criação de objetos em um nó remoto no grid. Pode ser criada uma marcação específica que envia pelo grid um pedido de instanciação de um objeto.

Este objeto, uma vez instanciado, poderia ter seus métodos chamados remotamente. Essa abordagem é interessante, pois permite criar agentes remotos, ou usar a memória de outro nó, complementando a memória local.

Por exemplo, o nó remoto pode executar um cálculo e guardar o seu resultado para si. A chamada de um método pode obter esse resultado toda vez que esses dados forem necessários.

Referências Bibliográficas

AJDT Project. Disponível em <<http://www.eclipse.org/ajdt>>. Acesso em: 12 de junho de 2006

ASPECTJ. Disponível em: <<http://aspectj.org>>. Acesso em: 12 de junho de 2006

The ASPECTJ5 Development Kit Developer's Notebook. AspectJ team. Disponível em: <<http://eclipse.org/aspectj/doc/next/adk15notebook>>. Acesso em: 12 de junho de 2006

CHEVENAK, A.; FOSTER, I.; KESSELMAN, C.; SALISBURY, C.; TUECKE, S. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets JOURNAL OF NETWORK AND COMPUTER APPLICATIONS, 23:187-200, 2001

DEMICHIEL, L.; RUSSEL, C. A Letter to the Java Technology Community. Sun Microsystems, JSR-220 and JSR-243, Disponível em: <<http://java.sun.com/j2ee/letter/persistence.html>>. Acesso em: 12 de junho de 2006

DRIANKOV, D.; HELLENDORF; REINFRANK, M. An Introduction fo Fuzzy Control, SPRINGER-VERLAG, 1995

ECLIPSE Project. Disponível em: <<http://www.eclipse.org>>. Acesso em: 12 de junho de 2006

EJB 3.0 Expert Group. JSR 220: Enterprise JavaBeans™, Version 3.0 – Java Persistence API. Sun Microsystems. 2005

FOSTER, I.; KESSELMAN, C.; LEE, C.; LINDELL, R.; NAHRSTEDT, K.; ROY A. A distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In: INTERNATIONAL WORKSHOP ON QUALITY OF SERVICE, 1999

FOSTER, I.; KESSELMAN, C.; TUECKE, S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. In: INTERNATIONAL JOURNAL OF SUPERCOMPUTER APPLICATIONS, 15(3), 2001-A

FOSTER, I.; KESSELMAN, C.; NICK, J.; TUECKE, S. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. In: INTERNATIONAL JOURNAL OF SUPERCOMPUTER APPLICATIONS, 2001-B

FOSTER, I. What is the Grid? A Three Point Checklist. Argonne National Laboratory & University of Chicago, 20 de Julho de 2002

FOSTER, I. Service-Oriented Science. In: SCIENCE, vol. 308, 6 de Maio de 2005

GAF4J. Disponível em: <<http://www.alphaworks.ibm.com/tech/GAF4J>>. Acesso em: 12 de Junho de 2006

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. Design Patterns – Elements of Reusable Object-Oriented Software, Addison Wesley, 1994

GLOBUS Alliance. Disponível em: <<http://www.globus.org>>. Acesso em: 12 de junho de 2006

HAVINGA, W.; NAGY, I.; BERGMANS, L. Introduction and Derivation of Annotations in AOP – Applying Expressive Pointcut Languages to Introductions

HUNT, A.; THOMAS, D. Pragmatic Unit Testing in Java with JUnit, 2003 The Pragmatic Programmers

JANTZEN, J. Design Of Fuzzy Controllers. Technical University of Denmark, Department of Automation, DENMARK Tech. report no 98-E 864 (design), 19 Aug 1998

JAVA. Disponível em: <<http://www.java.sun.com>>. Acesso em: 12 de junho de 2006

JAVA 1.5 Documentation. Sun Microsystems. Disponível em <<http://java.sun.com/j2se/1.5.0/docs>>. Acesso em: 12 de junho de 2006

JBOSS Project. JBoss Organization. Disponível em: <<http://jboss.com>>. Acesso em: 12 de junho de 2006

JUNIT. Disponível em: <<http://www.junit.org>> Acesso em: 12 de junho de 2006

LASZEWSKI, G.; FOSTER, I.; GAWOR, J.; LANE, P. A Java Commodity Grid Toolkit. Concurrency: Practice and Experience, 13, 2001

LEE, C. C.; Fuzzy Logic in Control Systems: Fuzzy Logic Controller – Part I. In: IEEE TRANSACTIONS ON SYSTEMS, MAN AND CYBERNETICS, vol. 20, no. 2, pp. 404-418, Março/Abril de 1990

LEE, C. C.; Fuzzy Logic in Control Systems: Fuzzy Logic Controller – Part I. In: IEEE TRANSACTIONS ON SYSTEMS, MAN AND CYBERNETICS, vol. 20, no. 2, pp. 419-435, Março/Abril de 1990

LEE, B.; SCHOPF, J. Run-time Prediction of Parallel Applications on Shared Environments. Poster Paper In: PROCEEDINGS OF CLUSTER2003, Dezembro de 2003

MATSUI, A. A. M.; SATO, L. M. Performance Data Interface: Obtenção de Performance Estendendo a Implementação OGSi para Redes Grid. In: VI WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, Sociedade Brasileira de Computação, 2005

MEYER, J.; DROWNING, T. Java Virtual Machine. O'Reilly, 1997

PRODAN, R.; FAHRINGER, T. ZENTURIO: A Grid Service-based Tool for Optimising Parallel and Grid Applications

RIPEANU, M.; IAMNITCHI, A.; FOSTER, I. Cactus Application: Performance Predictions in Grid Environments. In: EUROPAR 2001, Manchester, UK, Agosto de 2001

SCHOPF, J. M.; VAZHKUDAI, S. Predicting Sporadic Grid Data Transfers. In: 11TH IEEE INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE DISTRIBUTED COMPUTING (HPDC-11), IEEE Press, Edinburg, Scotland, Julho de 2002

SULISTIO, A.; YEO, C. S.; BUYYA, R. A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools

SMITH, W.; FOSTER, I.; TAYLOR, V. Scheduling with Advanced Reservations. In: PROCEEDINGS OF THE IPDPS CONFERENCE, Maio de 2000

THOMAS, D.; HUNTE, A. Mock Objects, IEEE SOFTWARE, vol. 19, no. 3, pp. 22-24, Maio/Junho de 2002

TRAVIS, G. Understanding the Java ClassLoader, IBM developer Works

TUECKE, S.; CZAJKOWSKI, K.; FOSTER, I.; FREY, J.; GRAHAM, S.; KESSELMAN, C.; MAGUIRE, T.; SANDHOLM, T.; VANDERBILT, P.; SNELLING, D. Open Grid Services Infrastructure (OGSI) Version 1.0. – Global Grid Forum Draft Recommendation, 26 de junho de 2003

YANG, L.; SCHOPF, J. M.; FOSTER, I. Conservative Scheduling: Using Predicted Variance to Improve Scheduling Decisions in Dynamic Environments. In: SUPERCOMPUTING 2003, Novembro de 2003

ZADEH, L. A. Fuzzy Sets. In: INFORMATION AND CONTROL, vol. 12, pp. 94-102, 1968

ZADEH, L. A. Fuzzy Logic = Computing with Words. In: IEEE TRANSACTIONS ON FUZZY SYSTEMS, vol. 4, no. 2, pp. 103-111, Maio de 1996

ZADEH, L. A. Fuzzy Logic, IEEE 1998

ZHANG, X.; FRESCHL, J.; SCHOPF, J. A Performance Study of Monitoring and Information Services for Distributed Systems. In: PROCEEDINGS OF HPDC, Agosto de 2003

Apêndice A - Mecanismo para a criação de protocolos definidos pelo usuário

A criação de protocolos de usuário é possível através do uso da mensagem do tipo `USER_DEFINED_MESSAGE` que faz parte do protocolo da G4JEE. A cada tipo de mensagem desse protocolo, está associada uma classe. Antes de qualquer mensagem ser enviada, é feita uma verificação a respeito da consistência entre o conteúdo da mensagem e a classe especificada pelo tipo da mensagem.

No caso de mensagens do tipo `USER_DEFINED_MESSAGE`, a classe do conteúdo da mensagem é `UserMessage`. Essa classe destina-se a permitir que um programador crie seu próprio tipo de mensagem.

A construção de um protocolo definido pelo usuário consiste em criar um *enumeration* que implemente a interface `UserMessageType`, mostrada na Figura 61. O método `type()` informa qual deve ser a classe do conteúdo da mensagem. Já o método `validate(UserMessage message)` é chamado para que o G4JEE valide a mensagem antes de enviá-la pela rede. Este método dá ao programador a oportunidade de verificar a consistência das suas mensagens de forma centralizada.


```
1 package org.aamm.g4j2ee.processnode;
2
3 import java.io.Serializable;
4
5 import org.aamm.g4j2ee.datastructure.ValidationResp;
6
7 public interface UserMessageType extends Serializable
8 {
9     Class type();
10
11     ValidationResp[] validate(UserMessage message);
12 }
```

Figura 61 – Interface UserMessageType

Um exemplo de protocolo definido pelo usuário é mostrado na Figura 62. Neste protocolo estão definidos dois tipos de mensagens: CHAT e FILE. Nos dois casos, o método `validate` retorna um valor `null`, significando que a validação sempre informa que o conteúdo é válido. A classe da mensagem CHAT é a classe `String`, o que significa que uma mensagem de usuário do tipo CHAT deve ter como conteúdo uma `String`.

```

1 package org.aamm.g4j2eesampleclients;
2
3 import org.aamm.g4j2ee.datastructure.ValidationResp;
4 import org.aamm.g4j2ee.processnode.UserMessage;
5 import org.aamm.g4j2ee.processnode.UserMessageType;
6 import org.aamm.g4j2ee.userutil.ByteArray;
7
8 public enum MyMT implements UserMessageType
9 {
10     /**
11      * Chat message
12      */
13     CHAT(String.class),
14     /**
15      * File transfer message
16      */
17     FILE(ByteArray.class);
18
19     private Class c;
20
21     private MyMT(Class c)
22     {
23         this.c = c;
24     }
25
26     public Class type()
27     {
28         return c;
29     }
30
31     public ValidationResp[] validate(UserMessage message)
32     {
33         return null;
34     }
35 }

```

Figura 62 – Exemplo de protocolo definido pelo usuário

O tipo `ByteArray` escolhido pelo tipo de mensagem `FILE` é um `JavaBean` definido pela `G4JEE` que encapsula um vetor de bytes para trafegá-los pela rede.

O envio de uma mensagem por uma aplicação criada pelo programador é feito conforme mostrado no código-fonte da Figura 63. Primeiro deve ser criado um objeto do tipo `UserMessage`. Ao criar um objeto, deve ser informado o tipo da mensagem no construtor. O ID 4974 é o ID do nó para o qual a mensagem será enviada.

```

1 UserMessage<String> message = new UserMessage<String>(MyMT.CHAT);
2 message.setTo(4974);
3 message.setValue("Bom dia.");
4 ProcessNode.getProcessNode().sendMessage(message);

```

Figura 63 – Envio de uma mensagem definida pelo usuário

O conteúdo da mensagem é atribuído na linha 3, seguindo a orientação do *enumeration* MyMT. O tipo UserMessage é parametrizado e o tipo-parâmetro é usado para o tipo do argumento do método setValue(). Procedendo-se desta forma, o programador que utiliza a classe UserMessage é desonerado de ter que executar casts de tipos e correr o risco de escrever código-fonte que cause erros durante a sua execução. Ao invés disso, a parametrização mostrada na Figura 64 faz com que o programador descubra os erros durante a edição do código-fonte.

```

...  (...)
12
13  public class UserMessage<T extends Serializable> implements Serializable
14  {
...      (...)
17      private ByteArrayObject<T> value;
...      (...)
44      public void setValue(T value) throws CastException, ClassPendingException
45      {
...          (...)
59      }
60
61      public T getValue() throws ClassPendingException
62      {
63          return value.getValue();
64      }
...      (...)
136 }

```

Figura 64 – Parametrização do tipo UserMessage

O ProcessNode segue o *design pattern singleton* (GAMMA et al. 1994), ou seja, é uma classe que garante que ocorre em cada JVM apenas instância de si mesma. O fato de ser um *singleton* simplifica o controle da conexão com o tópico JMS. O envio da mensagem é feito por este *singleton*.

O singleton ProcessNode também registra listeners para receber as mensagens de usuário enviados para o nó local. Para que um objeto seja registrado como listener para receber mensagens, este objeto deve ser de um tipo que implementa a interface ProcessNodeListener, mostrada na Figura 65. O registro de um desses objetos é feito

através do método `addListener(ProcessNodeListener listener)` definida na classe `ProcessNode`. O método `messageReceived` é chamado em todos os listeners cadastrados assim que uma mensagem definida pelo usuário é recebida pelo nó.

```

1 package org.aamm.g4j2ee.processnode;
2
3 public interface ProcessNodeListener
4 {
5     public void messageReceived(UserMessage message);
6 }

```

Figura 65 – Interface ProcessNodeListener

Assim, por exemplo, uma aplicação que simplesmente recebe mensagens remotamente e as exibe no console poderia ser escrita conforme a Figura 66. O cadastramento do listener ocorre na linha 12. Assim que uma mensagem é recebida pelo grid, o `ProcessNode` passa essa mensagem para o listener cadastrado através da chamada ao método `messageReceived()`.

```

1 package org.aamm.g4j2eesampleclients.text;
2
3 import org.aamm.g4j2ee.datastructure.ClassPendingException;
4 import org.aamm.g4j2ee.processnode.ProcessNode;
5 import org.aamm.g4j2ee.processnode.ProcessNodeListener;
6 import org.aamm.g4j2ee.processnode.UserMessage;
7
8 public class PrintListener implements ProcessNodeListener
9 {
10     public static void main(String[] args)
11     {
12         ProcessNode.getProcessNode().addListener(new PrintListener());
13     }
14
15     public void messageReceived(UserMessage message)
16     {
17         try
18         {
19             System.out.print("Mensagem recebida: ");
20             System.out.println(message.getValue().toString());
21         } catch (ClassPendingException e)
22         {
23             e.printStackTrace();
24         }
25     }
26 }

```

Figura 66 – Exemplo de programa que recebe mensagens definidas pelo usuário

Apêndice B – Manual de programação sucinto

Neste apêndice, mostrar-se-á um procedimento simples para criar uma aplicação que use o grid para executar um método remotamente. Os pré-requisitos para o desenvolvimento dessa aplicação são:

1. o IDE Eclipse instalado e com o plugin AJDT (ambos podem ser obtidos gratuitamente pelo site <http://www.eclipse.org>)
2. acesso ao projeto modelo, empacotado em um arquivo ZIP
3. disponibilidade de um servidor previamente configurado
4. um nó já conectado a esse servidor configurado

Figura 67 – IDE Eclipse sem projetos

A criação de um projeto para o grid deve ser feita a partir do IDE Eclipse. A Figura 67 mostra o IDE Eclipse sem projetos. Esta instância do Eclipse já contém o plugin AJDT instalado. A presença desse plugin no Eclipse pode ser identificada na figura pela presença dos ícones contendo a sigla “AJ”.

Apesar de esse plugin ser essencial para o desenvolvimento de aplicativos G4JEE, não é necessário que o programador saiba usar orientação a aspectos para desenvolver aplicações para o grid. O plugin é usado para aplicar os aspectos previamente criados (parte da G4JEE) e para exibir a aplicação desses aspectos, conforme será mostrado a diante.

A primeira etapa é a importação do projeto modelo criado previamente. Este projeto já contém os arquivos JAR necessários para que o projeto seja executado

usando o servidor de aplicações JBoss, com suporte ao banco de dados Java DB e com todos os elementos da G4JEE para o cliente (aspectos e classes).

Figura 68 – Wizard de importação de projeto

Figura 69 – Seleção do arquivo template_g4j2ee.zip

Figura 70 – O projeto modelo é criado dentro do IDE

Uma vez criado dentro do Eclipse, o projeto já conta com a estrutura necessária para iniciar o desenvolvimento de classes que usam o grid.

Figura 71 – Criação de uma classe executável

A criação de uma classe executável para o grid não é diferente de qualquer classe executável comum da linguagem Java.

Figura 72 – A classe PrimeiroPrograma logo após a sua criação

Figura 73 – Edição da classe para incluir elementos remotos

A classe deve ser alterada contendo a anotação `@GR` e deve implementar a interface `GRemote`. É importante notar também que a aplicação deve ser finalizada explicitamente (conforme a Figura 73). Caso contrário, um *listener* da G4JEE fica ativo e não permite que a aplicação termine. Chamadas ao método `System.exit(int)` são detectadas por um point cut que trata de executar o desligamento do `ProcessNode` do nó. Assim, o programador não precisa conhecer a API do grid para iniciar e finalizar corretamente o nó no grid.

Se, mesmo o grid apresentando esse comportamento automático, o programador decidir controlar explicitamente o ciclo de vida do nó do grid, a API do grid fornece métodos simples para esse propósito.

Esse comportamento serve para que essa aplicação fique disponível para receber tarefas do grid. Ou seja, a G4JEE entende que uma aplicação que não é terminada explicitamente deve ser automaticamente considerada disponível para uso do grid.

Nota-se na Figura 73 que a linha “pa.remoto()” é marcada com uma seta dupla. Essa marcação é feita pela ferramenta AJDT para informar ao programador que esta linha está sendo afetada por um aspecto do tipo “around”. Sem essa marcação, o programador pode acabar por se perder ao tentar encontrar algum erro nos seus programas.

Figura 74 – Edição do arquivo jndi.properties

A única tarefa de configuração necessária para a execução de uma aplicação usando o G4JEE é a edição do arquivo `jndi.properties`, que já vem criado dentro do projeto modelo.

O arquivo já vem preparado para indicar ao G4JEE para se conectar a um servidor local. Então, se o servidor estiver sendo executado localmente, não há necessidade de alterar este arquivo.

Figura 75 – Execução do programa

Ao ser executado, os comandos do método `main()` começam a ser executados. A chamada ao método “`remoto()`” é trocada por uma chamada a um método da G4JEE sem que o programador tenha que, explicitamente, chamar este método.

Nas linhas seguintes da Figura 75 mostram o engine do G4JEE tentando ler um estado anterior do nó. Como a execução mostrada na figura é a primeira feita neste nó, o engine cria um novo singleton do zero e inicia a comunicação JMS usando o id 2, que é um id especial usado pelos nós que ainda não obtiveram um id válido. A seguir, o nó recebe um id escolhido pelo servidor.

Depois, o cliente envia para o servidor um pedido de sugestão de nó para a execução do método “remoto()”. O servidor sugere que o nó de ID 1025 seja acionado. O programa, então, envia a requisição para o nó 1025.

Ao finalizar a execução desse método remoto, a execução do método “main()” continua e a linha “Fim do programa” é impressa no console.

```
1 ProcessNode :: task run message received
2 *** ByteArrayObject :: Class name: PrimeiroPrograma
3 ProcessNode : Class data received
4 ProcessNode :: getClassData() :: continuando
5 Executando remotamente
```

Figura 76 – Resultado da execução do método remotamente

Na aplicação remota, executada no nó 1025, o console exibe a mensagem mostrada na Figura 76. Nota-se que a mensagem “Executando remotamente” é impressa nesse console remoto, e não no console local onde o programa PrimeiroPrograma está sendo executado.

As demais mensagens exibidas nos consoles são presentes meramente para efeito de debug.

Após a execução do programa, nota-se que dois arquivos chamados “processNode1.state” e “processNode2.state” foram criados no diretório *home* do usuário. Estes arquivos são contêm estados de um objeto da G4JEE e servem para manter o id deste nó e o estado da execução de algum método remoto. O tempo

necessário para a execução desse programa foi suficiente para a escrita de apenas dois desses arquivos, mas se a aplicação tivesse ficado executando por mais alguns segundos, outros arquivos de estado teriam sido criados.

A criação desses arquivos se dá sempre no diretório indicado pela variável “user.home”, que pode ser alterada pela linha da Figura 77. Este diretório contém, também, os arquivos de dados do DBMS Java DB.

```
1 System.setProperty("user.home", "~/novo_home");
```

Figura 77 – Alteração do diretório padrão para salvamento de dados