CALEBE DE PAULA BIANCHINI

UM AMBIENTE PARA PROGRAMAÇÃO ORIENTADA A OBJETOS DISTRIBUÍDOS E PARALELOS EM GRADES COMPUTACIONAIS

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Doutor em Engenharia Elétrica.

São Paulo
2009

CALEBE DE PAULA BIANCHINI

UM AMBIENTE PARA PROGRAMAÇÃO ORIENTADA A OBJETOS DISTRIBUÍDOS E PARALELOS EM GRADES COMPUTACIONAIS

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Doutor em Engenharia Elétrica.

Área de Concentração:
Sistemas Digitais

Orientador: Profa. Livre-Docente Liria Matsumoto Sato

São Paulo
2009

AGRADECIMENTOS

Agradeço a Deus pelo milagre da minha vida, que está completa em Cristo Jesus: “Porque Dele, por Ele e para Ele são todas as coisas”. *Soli Deo gloria*.

Agradeço a minha orientadora, Profa Dra. Liria Matsumoto Sato, pelo apoio durante esses anos de estudos e pelo incentivo nos momentos de fraqueza. E, em especial, pela ajuda incondicional nos últimos dias de escrita deste trabalho.

Em especial, quero agradecer à minha amada esposa Grazielle de Arruda Werneck Bianchini que, durante boa parte do desenvolvimento deste trabalho, se mostrou prestativa em ouvir meus problemas técnicos, me consolar quando os defeitos apareciam, e me incentivar nos estudos, mesmo não conhecendo profundamente o tema deste trabalho.

Agradeço também minha família (Syrthes, Abiail, Lorise e Cinira), pelo suporte emocional durante os primeiros anos de meus estudos. Agradeço também àqueles que se juntaram a essa família tão querida (Atacy e Louise) e fizeram diferença nesse percurso.

Quero agradecer também aos meus dois irmãos, Francisco Isidro Massetto e Augusto Mendes Gomes Jr, que dividiram comigo momentos de alegria e angústia, mostrando o verdadeiro valor de uma amizade.

Agradeço também ao LAHPC pela infra-estrutura disponível e aos meus companheiros de laboratório, em especial: Nilton César, Fernando Kakugawa, Mathias, Charles, Darlon, Artur, Denis e Jean.

Agradeço também ao Francisco Ribacionka pelo apoio técnico e ao LCCA/USP – Laboratório de Computação Científica Avançada – pela infra-estrutura disponibilizada.

Não posso deixar de agradecer também meu antigo orientador de mestrado, Prof. Dr. Antonio Francisco do Prado, por me abrir os caminhos da pesquisa e do trabalho científico, permitindo que eu chegasse até esse ponto.

E, por fim, agradeço a todos os quais não pude mencionar neste trabalho por falta de espaço, mas que também estiveram comigo durante essa caminhada.

Meu sincero MUITO OBRIGADO!

“Que proveito tem o homem de todo o seu trabalho, com que se afadiga debaixo do sol?... Então vi que a sabedoria é mais proveitosa do que a estultícia, quanto a luz traz mais proveito do que as trevas... Ah! Morre o sábio, e da mesma sorte, o estulto! Pelo que aborreci a vida, pois me foi penosa a obra que se faz debaixo do sol; sim, tudo é vaidade e correr atrás do vento. Também aborreci todo o meu trabalho, com que me afadiguei debaixo do sol... Porque há homem cujo trabalho é feito com sabedoria, ciência e destreza; contudo, deixará o seu ganho como porção para aquele quem por ele não trabalhou;... Porque todos os seus dias são dores, e seu trabalho, desgosto; até de noite não descansa o seu coração; também isso é vaidade... Porque Deus dá sabedoria, conhecimento e prazer ao homem que lhe agrada... De tudo o que se tem ouvido, a suma é: teme a Deus e guarda os seus mandamentos; porque isso é o dever de todo homem. Porque Deus há de trazer a juízo todas as obras, até as que estão escondidas, quer sejam boas, quer sejam más.”

Livro do Eclesiastes

RESUMO

Grades Computacionais (*computational grid*) já é uma realidade tanto no meio acadêmico quanto no meio empresarial. Seu uso se tornou popular principalmente devido à divulgação dos trabalhos nesta área e pela propaganda de produtos e *softwares* que oferecem essa idéia. Apesar disso, ambientes para o desenvolvimento de aplicações orientadas a objetos em Java para uma infra-estrutura de *grid* ainda é escasso. Algumas iniciativas oferecem bibliotecas para este desenvolvimento. Outras utilizam paradigmas diferentes, como o de passagem de mensagem, para o desenvolvimento de aplicações. Além disso, a própria infra-estrutura de *grid*, formada por diferentes domínios administrativos com diferentes políticas de segurança e uso, impede que as aplicações sejam executadas nos diversos níveis existentes no *grid*. Estes níveis, formados por computadores e *clusters* de computadores com nós de execução, possuem endereçamento privado, impossibilitando que as aplicações alocadas em cada um desses computadores/nós, em diferentes domínios e diferentes endereços, se comuniquem de forma transparente. Visando uma solução para esses problemas, esta tese apresenta um ambiente para programação orientada a objetos distribuídos e paralelos, em Java, denominado J4GE. Nesse ambiente, o modelo orientado a objetos é base para a distribuição das classes, métodos e atributos existente em uma aplicação. Além disso, o ambiente oferece transparência no acesso aos objetos espalhados pelo *grid* através de um Serviço de Mensagem, independente do nível onde o recurso, computador ou nó, se encontra. Essa transparência permite também que o programador utilize a plataforma Java sem a necessidade de aprender ou conhecer novas bibliotecas ou paradigmas, diminuindo o esforço no desenvolvimento de aplicações para *grid*. E, juntamente com os recursos da plataforma Java e do ambiente J4GE, é possível criar objetos distribuídos com comportamento paralelo e concorrente, trazendo maior eficiência para a execução da aplicação.

Palavras-chave: Grades computacionais. Sistemas distribuídos. Ambiente de programação. Java.

ABSTRACT

Computing grid is already a reality both in academic and business world. Its use has become popular mainly because of the projects in this area and the advertising of products and software that offer this idea. Nevertheless, environments for development of object-oriented applications in Java for grid infrastructure are still scarce. Some initiatives offer libraries for this development. Others use different paradigms such as the message-passing for development of applications. Moreover, the infrastructure of grid, formed by different administrative domain with different security policies, prevents the execution of applications at various levels in the grid. These levels, formed by computers and clusters of computers with execution nodes, have private addresses, make impossible the transparent communication of the applications allocated in each of these computers at different levels in different domains. Focused on these problems, this thesis presents an environment for distributed and parallel object-oriented programming in Java, called J4GE. In this environment, the object-oriented model is the basis for the distribution of classes, methods and attributes in an existing application. Moreover, the environment offers transparency in objects access around the grid through a Message Service, regardless the level where is the resource, or the computer, or the execution node. This transparency also allows the programmer to use the Java platform without knowing or learning new libraries or paradigms, reducing the effort in developing applications for grid. The resources of the Java platform and the environment J4GE together can create distributed objects with parallel and concurrent behavior, bringing greater efficiency to the application.

Keywords: Computational Grid. Distributed systems. Programming environment. Java.

LISTA DE ILUSTRAÇÕES

Figura 1 – Especificação da classe para Produtor/Consumidor, com semáforo e monitor, em Java [23]. 19
Figura 2 – Visão geral do ciclo de vida de compilação e execução de um programa Java [26]. 20
Figura 3 – Classificação das Arquiteturas de *Grids* [58]. 31
Figura 4 – Infra-estrutura integrada de Grid [1]. 32
Figura 5 – Arquitetura Javelin [66]. 34
Figura 6 – Arquitetura SUMA/G [67]. 36
Figura 7 – Comunicação em grupo no ProActive [12]. 38
Figura 8 – Arquitetura de componentes do *Globus Toolkit* [6]. 39
Figura 9 – Infra-estrutura simples de computadores em um *grid*. 44
Figura 10 – Infra-estrutura de *grid* com *cluster* de computadores. 46
Figura 11 – Sintaxe única para anotação de código-fonte, no J4GE. 48
Figura 12 – Classe anotada para execução paralela através do ambiente J4GE. 49
Figura 13 – Aplicação utilizando objetos distribuídos e paralelos pelo *grid*. 50
Figura 14 – Arquitetura J4GE em um *grid*. 51
Figura 15 – Arquitetura do Serviço de Mensagem. 52
Figura 16 – Caminho de uma mensagem por uma infra-estrutura de *grid* através do Serviço de Mensagem. 53
Figura 17 – Diagrama de seqüência para uso de um recurso do Serviço de Mensagem [90]. 55
Figura 18 – Especificação do protocolo do Serviço de Mensagem em WSDL. 56
Figura 19 – Diagramas de classes com a transformação de uma classe anotada. 60
Figura 20 – Diagrama de classes da transformação de uma referência de objeto usado em uma classe anotada. 60
Figura 21 – Código-fonte com anotação em classe para o ambiente J4GE. 61
Figura 22 – Diagramas de classes com a solução de um método anotado. 62
Figura 23 – Código-fonte com anotação em método para o ambiente J4GE. 63
Figura 24 – Diagramas de classes com a solução para anotação de atributo. 63
Figura 25 – Código-fonte para anotação em atributo para o ambiente J4GE. 64
Figura 26 – Infra-estrutura de *grid* para validação do ambiente J4GE. 67

Figura 27 – Diagrama de classes com dependência simples entre objetos. 69
Figura 28 – Diagrama de classes com dependência entre duas classes. 71
Figura 29 – Diagrama de classes com dependência entre duas classes remotas. ...71
Figura 30 – Diagrama de classes com generalização de objeto personalizado. 73
Figura 31 – Diagrama de classes com especialização da classe *Thread*. 74
Figura 32 – Diagrama de classes para associação simples. 76
Figura 33 – Diagrama de classes para associação entre classes remotas. 77
Figura 34 – Diagrama de classes para teste da estratégia de método remoto. 78
Figura 35 – Diagrama de classes para teste da estratégia de atributo remoto. 79
Figura 36 – Diagrama de classes especificando a criação de um objeto-*thread*.......81
Figura 37 – Diagrama de classes com a especificação da aplicação TSP. 84

LISTA DE TABELAS

Tabela 1 – Resumo das características e classificação das DJVMs........................29
Tabela 2 – Resumo das características dos ambientes orientados a objetos em *grid computing*...41
Tabela 3 – Descrição dos recursos disponíveis em cada Domínio administrativo para a infra-estrutura de *grid* usada na validação do J4GE.............................67
Tabela 4 – Entradas e saídas para teste da dependência simples. 70
Tabela 5 – Entradas e saídas para teste da dependência entre duas classes. 72
Tabela 6 – Entradas e saídas para teste de herança em classes personalizadas....73
Tabela 7 – Entradas e saídas para teste de especialização da classe *Thread*. 75
Tabela 8 – Entradas e saídas para teste da associação simples entre classes........76
Tabela 9 – Entradas e saídas para teste da relação de associação entre duas classes remotas...77
Tabela 10 – Entradas e saídas para teste da estratégia de método remoto. 79
Tabela 11 – Entradas e saídas para teste da estratégia de atributo remoto. 80
Tabela 12 – Entradas e saídas para teste de objeto-*thread* remoto.........................81
Tabela 13 – Tempo de execução (segundos) da procura por números primos, tanto em uma máquina local quanto no J4GE..82
Tabela 14 – Tempo de execução (segundos) de diversas ações do J4GE. 83
Tabela 15 – Tempo de execução (segundos) da aplicação TSP. 85
Tabela 16 – Casos de testes para a aplicação TSP...86
Tabela 17 – Tabela comparativa entre o J4GE e os demais trabalhos. 87

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CoG	Commodity Grid
DJVM	Distributed Java Virtual Machine
DSM	Distributed Shared Memory
HPC	High Performance Computing
J4GE	Java for Grid Environment
JaDiMa	Java Distributed Machine
JAVAPD	International Workshop on Java and Components for Parallelism, Distribution and Concurrency
JDK	Java Development Kit
JESSICA2	Java-Enabled Single-System Image Computing Architecture
JIT	Just-in-time compiler
JVM	Java Virtual Machine
LRM	Local Resource Management
MPI	Message Passing Interface
MPP	Massively Parallel Processing
NOW	Network of Workstation
OGSA	Open Grid Services Architecture
OGSI	Open Grid Services Infrastructure
OOPSLA	International Conference on Object-Oriented Programming, Systems, Languages and Applications
P2P	Peer to Peer
RMI	Remote Method Invocation
SBLP	Simpósio Brasileiro de Linguagens de Programação
SMP	Symmetric MultiProcessors
SSI	Single-System Image
TSP	Taveling Salesman Problem
WSRF	WebService Resource Framework

SUMÁRIO

INTRODUÇÃO 13
OBJETIVO 14
JUSTIFICATIVA 15
METODOLOGIA 16
ESTRUTURA DO TRABALHO 17
1 TRABALHOS RELACIONADOS 18
1.1 JAVA E A JVM (*JAVA VIRTUAL MACHINE*) 18
1.2 DJVM – *DISTRIBUTED JAVA VIRTUAL MACHINE* 22
1.2.1 cJVM 24
1.2.2 JESSICA2 25
1.2.3 Jackal 26
1.2.4 JavaSplit 27
1.2.5 Considerações sobre as DJVMs 28
1.3 *GRID COMPUTING* E A PLATAFORMA JAVA 29
1.3.1 Javelin 34
1.3.2 JaDiMa 35
1.3.3 ProActive 37
1.3.4 Globus 38
1.3.5 Considerações sobre *Grid Computing* e Java 40
2 AMBIENTE JAVA PARA GRADES COMPUTACIONAIS 42
2.1 INFRA-ESTRUTURA DE *GRID* 44
2.2 J4GE – *JAVA FOR GRID ENVIRONMENT* 46
2.2.1 A Aplicação no Ambiente J4GE 48
2.2.2 Arquitetura do J4GE em um *Grid* 50
2.2.2.1 Serviço de Mensagem 52
2.3 DESENVOLVIMENTO DO J4GE 53
2.3.1 Serviço de Mensagem 54
2.3.2 Distribuição da Aplicação 58
2.3.2.1 Estratégia para Classe/Objeto 59

2.3.2.2 Estratégia para Método 61
2.3.2.3 Estratégia para Atributo 63
2.3.3 Engenharia de *Bytecodes* **64**
2.4 CONSIDERAÇÕES SOBRE O AMBIENTE J4GE **65**
3 TESTES E ANÁLISE DE RESULTADOS **67**
3.1 VALIDAÇÃO DO MODELO ORIENTADO A OBJETOS **68**
3.1.1 Dependência entre Objetos **68**
3.1.1.1 Caso de Teste 1 – Dependência simples 69
3.1.1.2 Caso de Teste 2 – Dependência entre duas ou mais classes 70
3.1.2 Generalização de Objetos **72**
3.1.2.1 Caso de Teste 3 – Generalização de Objetos Personalizados 72
3.1.2.2 Caso de Teste 4 – Generalização de Objetos de Biblioteca e de *Threads* 74
3.1.3 Associação entre Objetos **75**
3.1.3.1 Caso de Teste 5 – Associação simples 75
3.1.3.2 Caso de Teste 6 – Associação entre Classes Remotas 77
3.1.4 Outros Casos de Teste **78**
3.1.4.1 Caso de Teste 7 – Método Remoto 78
3.1.4.2 Caso de Teste 8 – Atributo Remoto 79
3.2 VALIDAÇÃO DO MODELO DISTRIBUÍDO E PARALELO **80**
3.2.1 Caso de Teste 9 – Aplicação Distribuída Simples **80**
3.2.2 Caso de Teste 10 – Aplicação Distribuída e Paralela **83**
3.3 CONSIDERAÇÕES SOBRE OS RESULTADOS **86**
4 CONCLUSÃO **88**
4.1 CONTRIBUIÇÕES **89**
4.2 TRABALHOS FUTUROS **90**
REFERÊNCIAS **92**

INTRODUÇÃO

Nos últimos anos, o conceito de Grades Computacionais (*Computational Grid)* vem ganhando popularidade tanto na comunidade acadêmica quanto nas empresas e nos *slogan* dos produtos comerciais [1]. Esse conceito vem acompanhado de diversos padrões estabelecidos, como a OGSA/OGSI [2, 3], *webservices* [4], WSRF, dentre outros padrões que compõem esse novo conceito da computação.

Apesar da popularidade recente, as idéias e os princípios de *computational grid* foram desenvolvidos há certo tempo [5]. Porém, somente com o aumento da capacidade computacional e da confiabilidade e velocidade das redes de computadores esses conceitos começaram a ser produzidos [1].

Pode-se perceber que os conceitos de *grid* sofreram influência da computação cliente/servidor, P2P (*peer to peer*), computação distribuída, HPC (*high performance computing*), diferenciado-se no arranjo de seus equipamentos, na infra-estrutura de software necessária, na organização virtual em função das necessidades do cliente, além de outras características semânticas relevantes para esse novo paradigma [1].

Dentre os projetos de *grid*, o que mais se destaca é o *middleware* Globus [6], o qual também oferece um conjunto de ferramentas para a extensão de suas funcionalidades com o objetivo da personalização em domínios de problema específicos, como física de altas energias, bioinformática, petroquímica, dentre outros.

Dentro do contexto de multiplataforma, a linguagem Java [7] se destaca, pois também é utilizada para o desenvolvimento de diversos *grids*, como o Globus [8]. Com ela, é possível utilizar facilmente ferramentas fornecidas pelos *grids*, como os CoG (*Commodity Grid*) [9], bem como aproveitar as vantagens de novas tecnologias, como *webservices*.

Além disso, os recursos oferecidos por essa linguagem, como o paradigma orientado a objetos, a componentização e o conjunto de classes de bibliotecas com recursos para programação, permitem ao desenvolvedor utilizar conceitos de Engenharia de Software para a construção de aplicações em *grid* [10].

Porém, apesar desses avanços em produtos e serviços de *grid* e da programação concorrente e mecanismos de sincronização existentes na linguagem Java, ainda é necessário construir um ambiente especializado em aplicações *grid* para essa

linguagem, de tal forma que seja possível aproveitar todos os recursos existentes no *grid*.

Parte da definição desse ambiente é a organização dos recursos do *grid* em níveis, obedecendo às políticas de uso e de segurança que as organizações participantes do *grid* possuem. Como exemplo, pode-se citar o uso de *clusters* de computadores de acesso privado, porém pertencente ao *grid* para a realização de tarefas do tipo em lote (*batch*).

E, para que uma aplicação em um ambiente de *grid* funcione, também é necessário definir mecanismos de comunicação entre as partes dessa aplicação que foram distribuídas lógica e fisicamente em recursos diferentes desse *grid*.

Em busca de uma solução para essas situações apresentadas, essa tese propõe um ambiente para programação orientada a objetos distribuídos e paralelos em grades computacionais, denominado J4GE (*Java for Grid Environment*).

Esse ambiente permite que a orientação a objetos e todos os seus conceitos sejam utilizados no desenvolvimento de uma aplicação que será executada em um *grid*, não havendo a necessidade de conhecer ou aprender novas bibliotecas especializadas ou novos conceitos de programação paralela. Os objetos da aplicação são distribuídos pelo *grid* de acordo com *metadados* descritivos, permitindo ainda que objetos paralelos também sejam executados no *grid*.

Além disso, a referência para as instâncias das classes e a chamada de métodos em objetos são realizadas de forma transparente entre as partes da aplicação distribuída pelo *grid*, através de um mecanismo de comunicação existente no ambiente.

OBJETIVO

O objetivo deste trabalho é propor um ambiente para o desenvolvimento de aplicações orientadas a objetos distribuídos e paralelos em *grid*, denominado J4GE. Esse ambiente visa produzir um SSI (*Single-System Image*) [11], permitindo que a aplicação distribuída pelo *grid* se comporte exatamente como se estivesse em um ambiente local. Dessa forma, a aplicação pode utilizar os recursos oferecidos pela

linguagem de forma transparente, como, por exemplo, a criação de *threads* para execução paralela.

Além disso, esse ambiente oferece ainda um mecanismo de comunicação que integra as partes distribuídas da aplicação, independentemente do uso de *clusters* ou de computadores no *grid*, ultrapassando os limites do endereçamento privado de rede, mas mantendo as políticas de uso e de segurança de cada recurso.

Com uma arquitetura bem definida sobre uma infra-estrutura de *grid*, esse ambiente oferece estratégias de distribuição da aplicação pelos recursos de um *grid*. Essas estratégias, com base no modelo orientado a objetos, distribuem os objetos, métodos e atributos pelo *grid* de forma transparente para o programador, através de metadados. Essas estratégias permitem também que as bibliotecas da linguagem sejam utilizadas sem alterações. Dessa forma, é possível que uma aplicação, além de distribuir objetos pelo *grid*, tenha objetos paralelos executando concorrentemente. Além disso, esse ambiente constrói uma camada que interliga os diversos níveis de recursos existentes no *grid*, em particular os *clusters* de computadores, garantindo que a comunicação da aplicação seja realizada de forma transparente.

JUSTIFICATIVA

Soluções utilizando *computational grid* estão cada vez mais freqüentes, tanto no ambiente de pesquisa quanto no meio empresarial. Além disso, a característica multiplataforma da linguagem Java permite que ela seja utilizada como parte da solução envolvendo *grids*.

Apesar da existência de ferramentas que auxiliam o desenvolvimento de aplicações para *grids* em Java, ainda não existem ambientes que ofereçam total transparência nesse desenvolvimento, necessitando de conhecimentos de bibliotecas especializadas, como no ProActive [12], ou de bibliotecas de programação paralela, como no JaDiMa [13].

Além disso, os ambientes atuais não permitem que as partes de uma aplicação sejam distribuídas nos diversos níveis de recursos existentes, limitando o aproveitamento do *grid*. Essa limitação é decorrente do endereçamento de rede existente no *grid* como, por exemplo, em *cluster* de computadores, que possuem

endereçamento privado. Dessa forma, as partes de uma aplicação alocada em um *cluster* não podem ser acessadas pelas outras partes que foram alocadas em outros recursos do *grid*.

METODOLOGIA

A definição da arquitetura de um *grid* pode ser obtida através da literatura e dos trabalhos publicados nessa área [1, 2, 10]. Porém, a construção desse *grid*, considerando a organização de dispositivos existentes bem como a possibilidade de expansão, é uma tarefa complexa, uma vez que os manuais e tutoriais existentes, ou até mesmo os fóruns específicos das ferramentas, supõem que o organograma da equipe de montagem seja bem definido e estável. Isso pode ser verificado na estrutura com que esses documentos foram organizados, onde cada membro tem seu papel dentro do cenário de *grid* (o administrador de Sistema Operacional, administrador do *grid*, usuário do *grid*, programador, a entidade certificadora, dentre outros) [1, 14, 15].

Após o estudo desses documentos, foi possível construir uma infra-estrutura básica de *grid*, utilizando máquinas de diversos domínios que, nesse caso, serão consideradas como uma única organização virtual [1]. Cada domínio está preparado com um LRM (*Local Resource Management*) diferente para que os recursos dentro desse ambiente de teste sejam o mais heterogêneo possível.

Durante os diversos estudos sobre ambientes para *grid*, percebeu-se a necessidade de construir um ambiente multiplataforma, que não fosse dependente de uma JVM específica. Além disso, foi necessário definir uma estratégia que possibilitasse a comunicação da aplicação distribuída em diversos níveis no *grid*. Neste trabalho, uma infra-estrutura de *grid* em níveis significa que existem computadores ou *clusters* de computadores que não podem ser acessados diretamente, exigindo uma máquina de entrada, devido ao seu endereçamento privado.

Em seguida, o ambiente foi desenvolvido em Java, utilizando técnicas como *webservices*, anotações (*annotations*), *classloader* e instrumentalização.

Por fim, para validação do ambiente, foram construídas diversas aplicações exemplos que explorassem as estratégias de distribuição, paralelismo e transparência do ambiente. Dentre essas aplicações, foi construída uma particular para solução do problema do caixeiro viajante. Essa solução, além de usar diversos conceitos da orientação a objetos, como agregação e herança, utiliza também recursos da plataforma Java, como *threads* e sincronização entre essas diversas linhas de execução.

ESTRUTURA DO TRABALHO

Este trabalho de pesquisa está organizado da seguinte maneira: o primeiro Capítulo apresenta os principais trabalhos relacionados com o tema deste trabalho; o segundo Capítulo descreve o ambiente J4GE (*Java for Grid Environment*); o terceiro Capítulo apresenta os testes e os resultados obtidos com este trabalho; e, por fim, o quarto Capítulo apresenta as conclusões parciais e as propostas de continuidade do ambiente J4GE.

1 TRABALHOS RELACIONADOS

Neste Capítulo são apresentados diversos trabalhos que estão relacionados aos temas deste trabalho. Ele está dividido em três seções: a primeira descreve o conceito *multithreading* em Java para sistemas paralelos e as diversas técnicas visando alto-desempenho nas JVMs (*Java Virtual Machine*); a segunda seção apresenta os conceitos e descreve as principais soluções de DJVM (*Distributed Java Virtual Machine*); e a última seção apresenta e contextualiza *Computational Grid* com ênfase no uso de Java.

1.1 JAVA E A JVM (*JAVA VIRTUAL MACHINE*)

Desde a sua criação, a plataforma Java vem introduzindo novos conceitos tanto na comunidade acadêmica, visíveis em simpósios e congressos como o Java Grande Fórum [16], o SBLP [17, 18], o OOPSLA [19], JAVAPD [20], quanto no ambiente empresarial, através de padrões [21], ferramentas [22], plataformas [7] e *frameworks* [7, 22]. Esses avanços trouxeram novos desafios para a área de processamento de alto desempenho e sistemas distribuídos.

Aplicando os princípios de orientação a objetos, essa linguagem oferece um conjunto de componentes reusáveis que seguem os conceitos de classes, com atributos e métodos. E, nessa mesma linha, a programação multithreading é oferecida no nível da linguagem, inclusive com primitivas de sincronização, como monitores e semáforos [23]. A Figura 1 apresenta uma solução para o clássico problema Produtor/Consumidor [24] definindo uma classe com *buffer* limitado, controlada por semáforos contadores (linhas 05 e 06) e por monitores de seção crítica (linhas 18 e 30). Além dessas primitivas, outras também foram definidas pela plataforma para auxiliar na comunicação inter-*threads*, principalmente no uso de monitores [23], já que estes são nativos da linguagem: *wait()*, *notify()* e *notifyAll()*.

```
01  public class Array<E> {
02
03      protected ArrayList<E> array;
04      protected int size;
05      protected Semaphore empty;
06      protected Semaphore full;
07
08      public Array(int size) {
09          array = new ArrayList<E>(size);
10          this.size = size;
11          empty = new Semaphore(size);
12          full = new Semaphore(0);
13      }
14
15      public void put(E item) {
16          try {
17              empty.acquire();
18              synchronized (array) {
19                  array.add(item);
20              }
21              full.release();
22          } catch (Exception e) {
23          }
24      }
25
26      public E get() {
27          E tmp = null;
28          try {
29              full.acquire();
30              synchronized (array) {
31                  tmp = array.get(0);
32              }
33              empty.release();
34          } catch (Exception e) {
35          }
36          return tmp;
37      }
38  }
```

Figura 1 – Especificação da classe para Produtor/Consumidor, com semáforo e monitor, em Java [23].

Além do reuso de componentes, a linguagem Java possui outras características, como a geração de *bytecodes* intermediários, a compilação JIT (*Just-in-Time*) e mecanismos de otimização de código chamados de *hotspots*.

A primeira característica permite que o código compilado seja executado em qualquer ambiente operacional, desde que esteja presente uma máquina virtual Java que o interprete. Dessa forma, a independência de plataforma de execução pode ser alcançada com maior facilidade. Atualmente, essas máquinas virtuais (JVMs) são desenvolvidas por diversos fabricantes de *softwares* [22, 7, 25], presentes até mesmo em dispositivos móveis [21, 7]. A Figura 2 resume o ciclo de vida de compilação de um código Java, com a geração do código intermediário, denominado

bytecode (representado pelo arquivo *.class*), e sua execução na JVM de uma arquitetura qualquer [26].

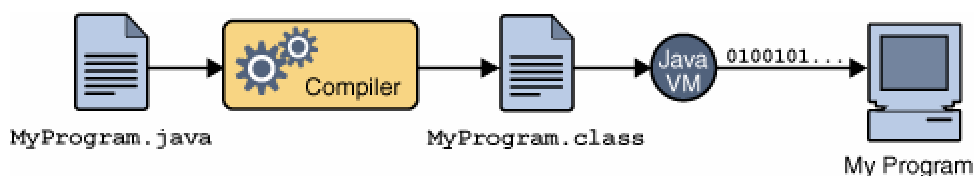


Figura 2 – Visão geral do ciclo de vida de compilação e execução de um programa Java [26].

A padronização dos *bytecodes* e a semântica que eles possuem durante a execução são definidos por uma comunidade aberta liderada pela Sun Microsystems [21, 7]. Dessa forma, as atualizações na plataforma, e o surgimento de novas idéias baseadas em Java, são amplamente discutidos e testados pelas mais variadas pessoas e empresas envolvidas com tecnologia.

Com essa padronização, diversos ambientes de compilação, interpretação e execução Java surgiram. Esses ambientes refletem a capacidade de aplicação dos diversos conceitos de Compiladores pelos seus fabricantes [22, 7, 25]. Durante a evolução da tecnologia Java, o custo da interpretação desse código intermediário foi revisto e fez-se necessário a transformação dos *bytecodes* em código nativo para a arquitetura específica. Nesse contexto, o conceito de compilação *Just-in-Time* (JIT) surgiu.

Basicamente, existem dois tipos de compilação JIT: estática e dinâmica. A primeira forma de compilação é semelhante aos compiladores tradicionais que, analisando o código fonte, geram um programa dependente da arquitetura e da plataforma alvo. Para a execução desse programa, é necessário todo o *bytecode* convertido e disponível para a plataforma específica. Entre essas soluções, pode-se citar o Caffeine [27], o Marmot [28], o Excelsior JET [29], dentre outros trabalhos e produtos que discutem e apresentam vantagens sobre o uso de compiladores estáticos para Java.

Já a segunda forma de compilação, a dinâmica, não requer que todos os *bytecodes* estejam disponíveis. Isto significa que o carregamento das classes também será dinâmico, item não observável na compilação estática. Sendo assim, a geração de código nativo e específico para uma plataforma é feito durante a execução do programa Java, sob demanda. Uma das formas de geração dinâmica é utilizar o compilador JIT sobre um código que será executado pela primeira vez (método de uma classe). Este compilador então converte os *bytecodes* para a

plataforma específica, informando essa ação para a JVM. A próxima vez que esse código for executado, a JVM já terá uma versão compilada e, conseqüentemente, a execução será mais rápida.

Um dos grandes desafios para pesquisadores dessa área é desenvolver um compilador JIT leve, que não tenha um tempo elevado de geração de código e que ocupe pouco espaço de memória. E, para isso, são estudadas estratégias de como essa compilação deve acontecer.

Entre essas estratégias estão a compilação completa do código e a compilação de trechos de códigos muito utilizados, chamados *hotspots*. Esta última estratégia é bem difundida devido à sua adoção na JVM distribuída pela Sun [30, 31]. Sua abordagem consiste em, inicialmente, executar os bytecodes por interpretação e, após uma análise do comportamento do código executado, gerar um código otimizado para a arquitetura nativa. Essa análise comportamental visa descobrir quais são os 10% de código que são executados durante os 90% do ciclo de vida de execução do programa [32].

Não só essas características, mas também outras inerentes ao processo de compilação e execução podem ser citadas para um melhor desempenho da JVM. Algumas dessas melhorias apresentam técnicas de otimização focadas, inicialmente, em arquitetura SMP (*Symmetric MultiProcessors*).

Essa arquitetura possui grande potencial para HPC (*High Performance Computing*), mapeando as diversas *threads* Java para os processadores existentes. Nesse caso, não só a interpretação dos *bytecodes* deve ser considerada, mas também o escalonamento das *threads* da JVM junto ao sistema operacional da plataforma nativa.

Além disso, existe o overhead inerente à sincronização entre *threads*. Alguns esforços para minimizar esse *overhead* têm sido adotados, como, por exemplo, a análise dos objetos bloqueados. Essa análise consiste em eliminar o bloqueio caso ele seja feito e usado apenas em uma única *thread* [33], minimizando o tempo de bloqueio do processador.

Nessa arquitetura, devido ao compartilhamento da memória, esta acaba tendo vital importância durante a execução do programa, sendo considerada um recurso valioso. Assim, um coletor de lixo eficiente, que evite longo tempo de pausas de processamento, se torna fundamental em uma JVM. Tanto a JVM da Sun [34],

quanto de outras empresas [35], já apresentam soluções para coletores de lixo paralelos, aumentando sua eficiência no gerenciamento da memória.

1.2 DJVM – *DISTRIBUTED JAVA VIRTUAL MACHINE*

Uma definição para *cluster* [36, 11] é a existência de um conjunto de computadores interconectados por uma rede de comunicação de alta velocidade. Isso significa que cada computador (também chamado de nó) possui um processador e uma memória local, conectados apenas através de um canal de comunicação. A partir desse conceito, surgem as especializações dessa infra-estrutura [11], como o MPP (*Massively Parallel Processing*), o *cluster* de computadores dedicados e o NOW (*Network Of Workstation*).

Independente da organização, essa infra-estrutura apresenta algumas vantagens em relação à arquitetura SMP [11]: o custo-benefício, o baixo-acoplamento, a escalabilidade, dentre outras que a literatura enumera [5]. Apesar de um SMP possuir capacidade de I/O mais elevada, algumas soluções em *hardware* minimizam o impacto dessas operações, como, por exemplo, o uso de uma memória global compartilhada [37].

A decisão por *cluster* dedicado se tornou muito comum principalmente pelo uso de processadores de prateleira (*of-the-shelf*) e para solucionar diversos problemas de infra-estrutura, como redundância e alta-disponibilidade [5].

Em HPC, o grande desafio é tornar esse conjunto de computadores um único ambiente de execução para os programas. Algumas estratégias permitem esta solução no nível do *hardware* [37], outras no nível do sistema operacional [5], ou ainda no nível de um *middleware*, como as DJVM (*Distributed Java Virtual Machine*) [39]. Todas estas estratégias visam produzir um sistema de imagem única (*Single-System Image*) [11].

As DJVMs aproveitam as facilidades que a linguagem Java oferece para a construção de soluções *multithreads*. Nesse caso, é possível afirmar inclusive que essas construções paralelas em Java são mais fáceis e naturais que outras soluções, como MPI (*Message Passing Interface*), pois não precisa de um particionamento explícito dos dados entre os processadores.

Uma DJVM permite que *threads* sejam criadas e executadas em um ambiente distribuído de *cluster* como se fosse um ambiente local. Dessa forma, uma DJVM cria uma SSI (*Single-System Image*) para a plataforma *multithread* Java. Além da criação de *threads*, uma DJVM deve oferecer os mesmos recursos das JVM convencionais.

Para a existência desse sistema único, as DJVMs precisam de um escalonamento entre os nós do *cluster*, transparência de acesso entre os objetos distribuídos e operações de I/O.

Diferentemente das JVM, que focam os principais estudos em compiladores JIT, os principais focos de estudos de uma DJVM envolvem melhores formas de acesso e compartilhamento de objetos em um *cluster*, bem como diferentes algoritmos de escalonamento entre seus nós. Além disso, a necessidade de um coletor de lixo eficiente e distribuído também se faz necessário.

Existem alguns pontos de vista a serem analisados em uma DJVM. Dentre esses pontos, pode-se destacar a visão do programador Java e a construção de uma DJVM. Sob o primeiro ponto de vista, a melhor DJVM é aquele que oferece um ambiente de execução sem alterações no código fonte. Nesse caso, o programador desenvolve as aplicações de maneira convencional. Na outra extremidade, a DJVM oferece um conjunto de sintaxe ou biblioteca particular para a distribuição, o compartilhamento e a execução de *threads* distribuídas no *cluster*. Nesse caso, o programador é obrigado a modificar seu código fonte, atendendo as exigências da DJVM. Além disso, a DJVM pode optar pela compilação do código fonte para um binário dependente de arquitetura, facilitando o controle das *threads* distribuídas em detrimento das características dinâmicas que Java oferece.

O outro ponto de vista, envolvendo a construção de uma DJVM, analisa três principais características de um DJVM: escalonamento distribuído de *threads*, compartilhamento de memória distribuída e *engine* de execução. A primeira característica se preocupa em escalonar o contexto de cada *thread* Java eficientemente dentro do cluster, de tal forma que se alcance um aumento de desempenho da aplicação. A segunda característica deve oferecer o espaço de memória Java (chamado de *heap*) de forma compartilhada para que as *threads* distribuídas consigam eficientemente trabalhar com seus dados compartilhados. Esta característica merece atenção especial devido à sobrecarga de comunicação que pode existir, bem como devido à necessidade de um coletor de lixo capaz de

lidar com memória compartilhada. E, por fim, a *engine* de execução deverá ser especialmente projetada para entender esse ambiente distribuído e oferecer as ações apropriadas para cada objeto, sem prejudicar o desempenho de sua execução.

Existem diversos trabalhos que apresentam um protótipo de DJVM. Alguns deles não são transparentes para o programador ou não apresentam mecanismos de máquina virtual. Alguns desses trabalhos são apresentados nas seções a seguir.

1.2.1 cJVM

cJVM [40, 41] é uma DJVM que oferece uma visão SSI para as diversas JVM de um *cluster*. Ele é baseado na versão 1.2 do padrão Java da Sun. Atualmente, o projeto não está mais em andamento, mas foi um dos pioneiros em oferecer um SSI para essa plataforma.

Aplicações convencionais *multithread* podem ser executadas diretamente na cJVM, sem necessitar de codificação explícita ou pré-processamento. Para isso, cada nó do cluster possui um processo JVM para garantir a execução das *threads* distribuídas. Quando uma *thread* é criada, um algoritmo de balanceamento de carga escolhe a localização de sua criação. Uma vez escolhido o nó, a *thread* é criada e seu ciclo de vida fica restrito àquela máquina.

O compartilhamento de memória é feito através de *proxies* [41]. Quando um objeto é criado em um nó, este pertence apenas a este nó e poderá ser acessado pelos outros nós através de *proxies*. Esse *proxy* é utilizado como uma referência remota ao objeto, porém transparente para a aplicação Java. Essa referência remota só é visível no nível do *kernel* do cJVM. Dessa forma, todos os *heaps* locais das cJVMs formarão uma única memória para as *threads*.

Alguns recursos para reduzir a sobrecarga de comunicação entre objetos são aplicados aos *proxies* do cJVM. Dentre estes recursos, uma análise durante o carregamento da classe verifica se um método é *stateless*, ou seja, não acessa qualquer atributo da classe, executando essa ação localmente.

1.2.2 JESSICA2

JESSICA2 (*Java-Enabled Single-System Image Computing Architecture*) [42, 43] é um *middleware* para execução paralela de *threads* Java em *clusters* de computadores. É uma DJVM que também oferece uma visão SSI para JVMs. Baseada na versão 1.2 de referência da Sun, seu núcleo utiliza a versão 1.0.6 do compilador Kaffe [44].

Os programas Java podem ser executados diretamente no JESSICA sem pré-processamento algum. Durante sua execução, as *threads* podem ser redistribuídas visando alcançar um real paralelismo.

Dessa forma, é possível ter *threads* preemptivas com migração durante a execução da aplicação. Essa migração acontece segundo a segmentação do código de acordo com o contexto de execução. Cada segmento é classificado como independente ou dependente. Os segmentos independentes são considerados possíveis de migração e, quando executados remotamente, o controle da execução é repassado novamente ao segmento local dependente. Já os segmentos dependentes não migram, executando localmente seus códigos e permitindo inclusive o uso de métodos nativos.

Além da migração do *bytecode*, é necessário analisar quais segmentos identificados sofrerão influência do compilador JIT. No processo de migração de um segmento, as informações sobre essa *thread*, e que foi compilada para a plataforma local, precisa ser extraída. Isso significa que os códigos nativos compilados pelo JIT precisarão sofrer uma transformação de volta ao *bytecode*. Além disso, após a migração, novamente será necessário a utilização do compilador JIT para transformar o *bytecode* em código nativo.

O compartilhamento de memória, chamado *Global Object Space* (GOS), permite a referência de objetos e a chamada de métodos remotamente de forma transparente para a *threads*. A arquitetura do GOS define uma estrutura de *heap* para os objetos locais e objetos remotos que, quando acessados, são identificados pelo núcleo da JVM e associado por seus respectivos objetos locais ou remotos.

1.2.3 Jackal

Jackal [45, 46] é uma implementação de um *Distributed Shared Memory* (DSM) para a linguagem Java. Sua abordagem utiliza a compilação estática. Apesar disso, a construção das *threads* pelo programador não é alterada, apesar de necessitar da aplicação inteira (todas as classes) antes de ser executada.

Jackal aceita como entrada um código *bytecode* convencional ou um código fonte Java. Um pré-compilador analisa essa entrada e gera uma linguagem intermediária denominada LASM (*Liberally Assemby Language*). A partir dessa linguagem, um segundo compilador é responsável pelas otimizações seqüenciais e paralelas existentes. Por fim, esse código é compilado para um código paralelo nativo.

Durante a execução, quando uma *thread* é criada, Jackal intercepta esse evento e escolhe um nó ocioso do cluster aonde será criada essa nova *thread*. Ela fica alocada nesse nó até o final de seu ciclo de vida de execução.

Devido à análise feita durante a primeira fase de compilação, Jackal consegue identificar dois tipos de refinamentos em sua estrutura DSM. O primeiro tipo é definido baseado na relação entre os objetos de acordo com o uso de métodos de uma classe. Estes serão alocados com sendo uma única unidade de acesso, diminuindo o custo de sua identificação. O segundo tipo está relacionado com métodos que necessitam de sincronização, ou seja, controle de seção crítica. Nesse caso, Jackal cria um código especial de acesso a essa seção crítica de tal forma que o objeto que possui acesso a essa sessão executa o código no seu próprio nó, evitando diversas trocas de mensagens de coordenação. Se houver solicitações de acessos à mesma seção crítica, esta é enviada para o nó que primeiro instanciou o objeto. Este nó controla o acesso concorrente e permite que cópias do objeto sejam enviadas aos nós para execução local.

Jackal utiliza código nativo para executar a aplicação, descartando o uso de uma máquina virtual. Dessa forma, a aplicação fica dependente da plataforma e suas otimizações são feitas em tempo de compilação.

1.2.4 JavaSplit

JavaSplit [47, 48] é um compilador estático, semelhante ao Jackal. Porém, sua principal diferença é a geração de *bytecodes* ao final da compilação de um programa *multithread*. Isso permite a execução do programa em máquinas virtuais convencionais, ou seja, obtém um alto grau de portabilidade e uso dos recursos da JVM (como, por exemplo, do JIT). Essa re-compilação transforma um programa *multithread* convencional em um programa paralelo.

Nesse processo de re-compilação de um programa, JavaSplit incorpora um conjunto de bibliotecas e ferramentas no novo programa gerado. Em seguida, esse novo programa é enviado aos nós de trabalhos para execução através do método *main* padrão.

Para cada nova *thread* criada, uma alocação simples é feita: cada *thread* é alocada em um nó do *cluster* a partir de um algoritmo de balanceamento de carga. Uma vez alocada a *thread*, esta terá todo seu clico de vida de execução no mesmo nó. Além disso, ele permite que diversos algoritmos de balanceamento sejam incorporados no escalonamento do nó.

JavaSplit oferece uma camada transparente para o compartilhamento de memória (DSM). Ele inicialmente classifica os objetos criados como *local* e *compartilhado*, sendo necessário o gerenciamento apenas deste último. A identificação de objetos do tipo compartilhado acontece de forma dinâmica, uma vez que objetos locais podem ser gerenciados pela própria JVM e seu coletor de lixo. E, os poucos objetos que se comportam de forma compartilhada entre *threads* [47] permitem um ganho significativo no seu gerenciamento dinâmico.

Uma vez identificado um objeto compartilhado, o nó que o criou mantém a cópia principal, a qual deve ser gerenciada como a versão mais atualizada. As *threads* que utilizam esse objeto mantêm uma cópia local (cacheada), baseada na cópia principal. Em certos momentos de sincronização, as informações das cópias locais são enviadas para a cópia principal.

O processo de compilação é baseado na instrumentalização estática dos *bytecodes* utilizando a biblioteca BCEL (*Byte Code Engineering Library*) [49, 50], reescrevendo linearmente as classes para outras classes. Este processo é realizado em cinco etapas: (i) altera os códigos de criação local de *threads* pela criação

remota de *threads*; (ii) modifica os trechos de código com sincronização para as corretas sincronizações distribuídas necessárias; (iii) altera o acesso aos objetos para novo tipo de acesso que verifica a consistência de memória entre objetos distribuídos; (iv) acrescenta novos campos que indicam o estado do objeto; e por fim, (v) acrescenta métodos específicos de DSM na nova classe.

1.2.5 Considerações sobre as DJVMs

Apesar do estudo apresentar apenas algumas DJVMs, existem outras iniciativas que pesquisam a construção de um SSI para o ambiente Java. Dentre esses outros trabalhos, se destacam ainda o Java/DSM [51], o Hyperion [52] e o JavaParty [53].

Java/DSM [51] é baseado na versão de referência JDK-Sun 1.0.2 e o compartilhamento de memória é feito através da ferramenta Treadmarks [54], uma versão em C de DSM. Durante a programação, na criação de *threads*, o programador precisa explicitamente indicar em que computadores serão criadas. Além disso, não há migração de *threads* entre as máquinas do *cluster*, e sua JVM não dispõe de um compilador JIT.

Hyperion [52] é um compilador estático que converte *bytecodes* Java para código C. Durante essa transformação, diretrizes DSM são adicionadas ao código fonte, bem como otimizações para melhorar o desempenho da memória compartilhada em determinado trecho de código. Em seguida, o programa é compilado para a plataforma nativa.

JavaParty [53] utiliza uma JVM nativa para a execução de suas aplicações. Ele oferece um pré-processador e um ambiente de execução para *cluster* de computadores. Para isso, ele utiliza os conceitos de RMI (*Remote Method Invocation*) presentes na plataforma Java nativa. Para que o pré-processador identifique corretamente os blocos remotos, foi adicionada a palavra reservada "*remote*" à programação que, quando encontrada, é convertida para arquitetura RMI.

Diante do estudo e das descrições dessas diversas DJVMs, é possível apresentar um quadro comparativo com o resumo de suas principais características. Essas

características são apresentadas na Tabela 1 e enumeradas conforme o critério presente nos seguintes trabalhos [42, 48], a saber:

- Transparência: um sistema é considerado transparente se ambos, o programador e o responsável pela execução, não precisam ter conhecimento do ambiente distribuído;
- Portabilidade: um sistema é portável nesse contexto se possuir as mesmas características definidas para Java (*write once, run anywhere*);
- Compilação: processo de compilação utilizado até a geração final do programa;
- Compilador JIT: existência de um compilador JIT no ambiente de execução;
- Ambiente de execução: define o ambiente de execução entre uma JVM genérica, JVM proprietária ou um Sistema Operacional (SO);
- Entrada: itens necessários para a compilação do programa;
- Saída: resultado final, após a compilação.

Além dessas características determinantes para a aproximação de uma DJVM com um SSI, existem outros critérios de comparação de suas arquiteturas, como modelo de memória, coletor de lixo, memória compartilhada, migração de objetos, dentre outros que podem ser observados nos trabalhos [42, 48].

Tabela 1 – Resumo das características e classificação das DJVMs

	Java/DSM	cJVM	JESSICA2	Hyperion	Jackal	JavaParty	JavaSplit
Transparência	parcial	completa	completa	completa	completa	completa	completa
Portabilidade	parcial	parcial	parcial	não	não	completa	completa
Compilação	dinâmica	dinâmica	dinâmica	estática	estática	estática	estática
JIT	inexistente	inexistente	próprio	não se aplica	não se aplica	nativo	nativo
Ambiente	JVM própria	JVM própria	JVM própria	SO	SO	JVM genérica	JVM genérica
Entrada	*bytecode*	*bytecode*	*bytecode*	*bytecode*	código fonte	código fonte	*bytecode*
Saída	*bytecode* próprio	*bytecode* próprio	*bytecode* próprio	binário	binário	*bytecode* genérico	*bytecode* genérico

1.3 *Computational Grid* e a Plataforma Java

Computational Grid surgiu em conseqüência da rápida evolução das redes de computadores e dos diversos dispositivos computacionais conectados entre si. Essa

evolução foi impulsionada pela expansão e uso da *Internet*, modificando a computação e a sociedade. Além disso, o alto poder computacional distribuído e accessível através de redes de alta velocidade abriu novos horizontes para a solução de uma larga escala de problemas, tanto na ciência, quanto na engenharia, como também na indústria de negócios [55].

Até então, a computação não tinha condições suficientes para lidar com a complexidade e o tamanho desses problemas, uma vez que estes eram grandes consumidores de recursos computacionais e de espaço de memória e disco [55].

O termo *Grid* foi escolhido devido a sua semelhança com a rede elétrica (*eletric Grid*) [1]. Esta rede fornece energia sob demanda, deixando transparente sua origem e a complexidade da malha existente na transmissão e comunicação. Já a rede existente em *Grid* forneceria serviços e recursos computacionais, sob demanda [56], transparentemente.

Dessa forma, um *grid* possibilitaria o compartilhamento, escolha e agregação de diversos recursos, como supercomputadores, *storages*, recursos de dados, dispositivos especiais, dentre outros, que estão geograficamente espalhados e que pertencem a diversas empresas e organizações, visando a solução dos problemas na ciência, engenharia e na indústria de negócios.

Esses recursos, de forma mais geral, são chamados de serviços de um *grid*. E, a partir desse momento, é notável a participação tanto acadêmica quanto da indústria na viabilização de serviços sob demanda, caminhando naturalmente para uma convergência no uso dessa tecnologia [56].

É importante ressaltar algumas características que evidenciam o caráter distribuído, diverso e complexo de uma arquitetura *grid* [55, 56]:

- *Heterogeneidade*: existem os mais variados tipos de recursos, versões, gerações, instrumentos e até mesmo serviços, sendo necessário lidar com todos eles;
- *Distribuição Geográfica*: a existência de locais participantes de um *grid* espalhados pelo globo define muito bem essa característica;
- *Compartilhamento*: os recursos e serviços são compartilhados entre todas as aplicações que devem saber lidar com esse cenário, diferentemente de aplicações em lote com tempo de recurso dedicado;

- *Diversos Domínios*: como existem diversas instituições envolvidas no compartilhamento dos recursos e serviços, estes sofrem influência da política de acesso externo e interno aos itens compartilhados;
- *Controle Distribuído*: como conseqüência da distribuição e dos diversos domínios participantes, não há uma entidade centralizadora e controladora dos recursos e serviços.

Apesar dessas características, ainda não existe uma definição literal sobre o termo *Grid*. Porém, é possível perceber quando o termo é utilizado apenas para o *marketing* de um determinado produto [57].

E, analisando as necessidades gerais de uma aplicação em *grid*, é possível ainda classificar as arquiteturas *grid* considerando o fato de que estes são gerenciadores de recursos e serviços [58]. Essa classificação pode ser vista na Figura 3.

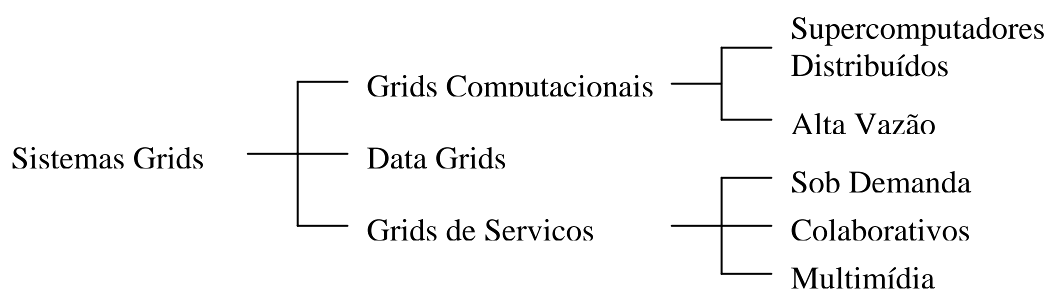


Figura 3 – Classificação das Arquiteturas de *Grids* [58].

A categoria *Grids Computacionais* representa o sistema que agrega alto poder de computação para uma única aplicação, e esta agregação é maior que qualquer computador individual. Dependendo de como esse sistema é usado, ele pode ser subdividido em outras duas categorias: *Supercomputadores distribuídos*, que executa aplicações de forma paralela em diversas máquinas para reduzir o tempo total de sua execução; e *Alta Vazão*, que aumenta a taxa de execução de uma seqüência de *jobs* e está bem adaptado para aplicações probabilísticas e estatísticas.

A categoria *Data Grids* é a classificação para sistemas que oferecem uma infra-estrutura para manipulação de dados, desde seu armazenamento até *datawarehouses*, em um ambiente distribuído. *Grids Computacionais* devem oferecer também mecanismos de armazenamento, mas se diferem dos *Data Grids*

devido à especialização que este último deve ter sobre o gerenciamento e armazenamento dos dados.

Por fim, a categoria de *Grids de Serviços* é para sistemas que oferecem serviços que não podem ser realizados em um único computador. Pode ser subdividido em três categorias: *Sob Demanda*, que re-organiza os recursos disponíveis no *grid* dinamicamente para oferecer novos e melhores serviços; *Colaborativos*, que interconecta usuários e aplicações dentro de ambientes virtuais, colaborativos e de tempo-real; e *Multimídia*, que oferece uma infra-estrutura para aplicações multimídia de tempo-real, com QoS (*Quality of Service*) superior à oferecida por uma aplicação em uma única máquina dedicada.

Apesar dessa classificação, a denominada infra-estrutura integrada de *grid* [1] pode ser composta por quatro componentes básicos e gerais, conforme apresentado na Figura 4.

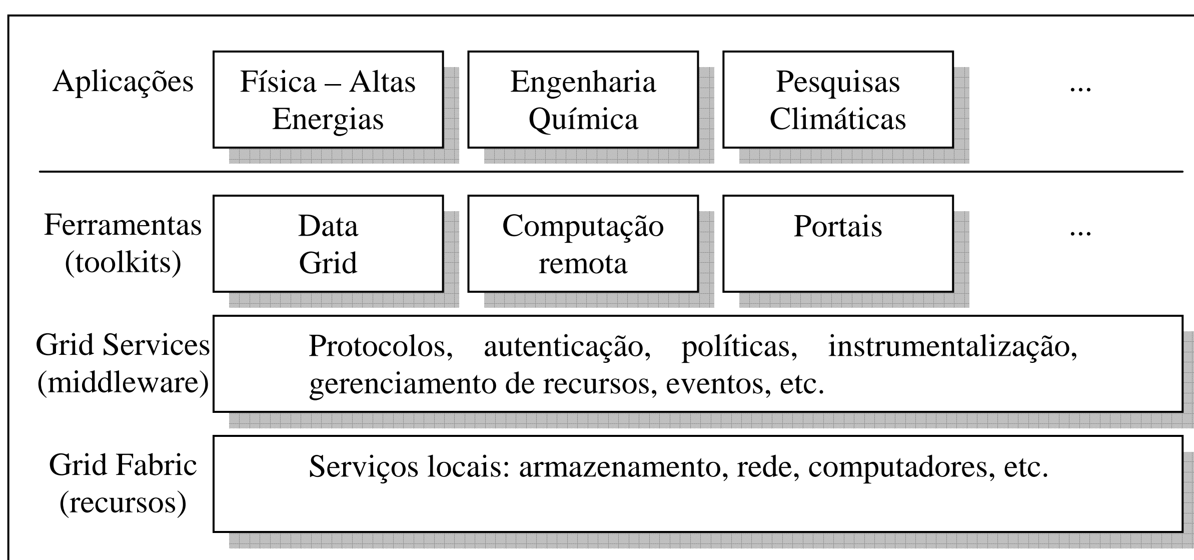


Figura 4 – Infra-estrutura integrada de Grid [1].

O *Grid Fabric* oferece recursos básicos e específicos para a operação de um *grid*, como, por exemplo, gerenciamento de recursos em um supercomputador ou QoS de um roteador de rede. A partir dessa infra-estrutura, é possível construir serviços de *grid* (*Grid Services*) independentes de plataforma ou aplicação. Como exemplo, pode-se citar um serviço de informação, que provê acesso uniforme à estrutura e ao estado dos recursos do *grid*. Podem ser citados ainda os mecanismos de autenticação e autorização, estabelecendo credenciais e estrutura de identificação.

A partir desses dois componentes básicos é possível desenvolver serviços mais específicos, dando origem às *Ferramentas* (*toolkits*): serviços de gerenciamento e realocação de dados distribuídos para aplicações *data-intensive*; gerenciamento de fluxos para ambientes colaborativos, etc. Esses serviços e ferramentas são então usados para a construção de *Aplicações* em seus domínios específicos.

Essa composição é o resultado dos passos dados pela comunidade científica de alto desempenho na construção de uma infra-estrutura que use recursos de alto poder computacional de forma distribuída e controlada [1].

Em paralelo, a quantidade de tecnologias de computação distribuída, que são vendidas como *commodities*, aumentou com a rápida difusão da Internet. Essa explosão trouxe diversos aspectos revolucionários no acesso a dados e processamento. Dentre essas tecnologias, surge o *Commodity Grid* (CoG), possibilitando o uso de tecnologias emergentes na construção de aplicações para *grids* e exportando toda a infra-estrutura e tecnologia de *grid* para computadores comerciais [9].

Os primeiros resultados dentro dos projetos CoG foram a construção de *frameworks* e ambientes de propósito geral que mapeavam um *grid*, denominados CoG Kit (*Commodity Grid Toolkit*). Através desses CoG Kits é possível construir aplicações que exploram as vantagens dos *grids*, como seus serviços avançados, juntamente com o alto nível de desenvolvimentos oferecido pelas tecnologias.

Dentre essas tecnologias de mercado (oferecidas como *commodities*), a que tem chamado atenção é a tecnologia Java. Existem diversos fatores para que essa tecnologia seja adotada: a orientação a objetos e a componentização dentro do conceito de alto nível da engenharia de software; sua interoperabilidade e portabilidade oferecida pelo seu código intermediário e a facilidade de se encontrar interpretadores (JVM) para diversas plataformas sem a necessidade de recompilação; a existência de diversas bibliotecas que oferecem protocolos e comunicação para redes de computadores e Internet; mecanismos de segurança como tipos de dados fortemente tipificados e área de segurança de execução; dentre outros fatores que podem ser encontrados na literatura [59, 60, 10].

Além disso, a tendência natural para o uso dessa tecnologia pode ser vista na popularidade que essa plataforma tem em comparação a outras existentes [61], bem como na força tarefa de órgãos internacionais, como ACM [62] e IEEE [63], na tentativa de padronização e uso nos cursos de graduação nas Universidades.

Seguindo essa tendência, diversos trabalhos tem sido desenvolvidos usando a tecnologia Java, utilizando também os resultados do projeto Java CoG Kit [59, 60], para a criação de novos ambientes de uso e desenvolvimento de aplicações para *grids.* Alguns desses trabalhos são apresentados nas seções a seguir.

1.3.1 Javelin

Javelin [64, 65, 66] é uma infra-estrutura Java para computação paralela sobre a Internet. Sua arquitetura é formada por três principais itens, apresentados na Figura 5: a aplicação cliente, os nós de execução e os *brokers.* A aplicação cliente é um processo a procura de recursos computacionais; os nós de execução oferecem esses recursos computacionais; e o *broker* é um processo responsável pela coordenação e alocação dos recursos para os clientes.

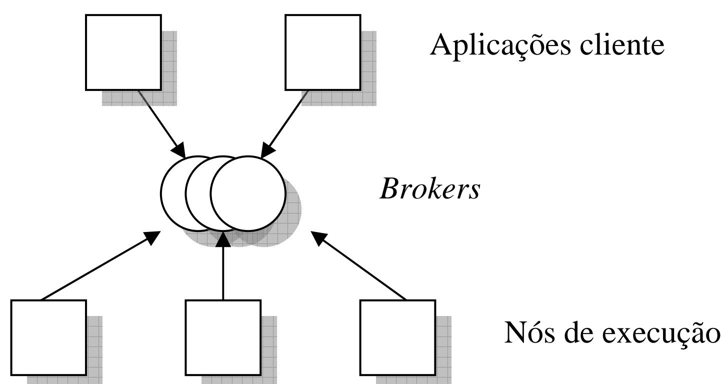


Figura 5 – Arquitetura Javelin [66].

Um dos modelos de computação paralela suportado por Javelin é a decomposição independente das tarefas, sendo estas autônomas inclusive de comunicação e/ou do escalonamento. Esse modelo é ideal para aplicações estatísticas e probabilísticas bem como para aplicações mestre-escravo. Além desse modelo, a última versão da infra-estrutura permite computação paralela aplicada em algoritmos *branch-and-bound*.

Javelin integra mecanismos de escalabilidade através de um escalonador determinístico juntamente com técnicas de tolerância a defeitos, substituindo nós que venham a falhar ou que sejam rejeitados.

Com essas características, Javelin pode ser classificado como um *grid* de alta vazão. Seus elementos são organizados hierarquicamente, onde os *brokers* são os controladores de árvores de nós. Esses nós oferecem os recursos computacionais e são nomeados de acordo com a hierarquia onde estão vinculados.

Qualquer nó provedor de serviços que deseja fazer parte da hierarquia Javelin se comunica com o JavelinBNS. Este é um sistema que gerencia a lista de *brokers* existentes na infra-estrutura. Em seguida, o nó se comunica com os *brokers*, decide pelo melhor deles e passa a fazer parte da lista de recursos gerenciados por este *broker* escolhido.

Tanto os nós quanto os *brokers* trocam informações durante seu ciclo de vida para manter a tabela de escalonamento atualizada. Este escalonamento é realizado de forma descentralizada, seguindo políticas pré-determinadas para cada aplicação. Para assegurar o balanceamento de carga no sistema, são os nós que requisitam tarefas a serem realizadas.

O mecanismo de busca por recursos de computação também é descentralizado, uma vez que as informações sobre cada nó de execução estão registradas nos diversos *brokers* existentes.

1.3.2 JaDiMa

JaDiMa[1] (*Java Distributed Machine*) [13] é um ambiente de execução de programas Java para plataformas *grid*. Ele automaticamente gerencia bibliotecas remotas usadas em aplicações Java aproveitando-se de vantagens como a orientação a objetos, a modularidade e a portabilidade existentes em Java.

JaDiMa está integrado a uma plataforma baseada em Globus denominada SUMA/G (*Scientific Ubiquitous Metacomputing Architecture/Globus*) [67]. Esta plataforma executa transparentemente *bytecodes* Java em máquinas remotas, estendendo o mecanismo de execução Java, em particular, o carregamento das classes e dos dados. A arquitetura do SUMA/G pode ser visto na Figura 6.

---

[1] O projeto não tem versão estável para *download*, mas pode ser acessado através do portal do projeto, em http://www.sourceforge.net/projects/jadima .

SUMA/G foi construído utilizando os recursos existentes no Java CoG Kit com comunicação através de RMI e CORBA. Para a execução de uma aplicação Java, o usuário utiliza as ferramentas dessa plataforma para indicar a execução de sua aplicação. Nesse processo, o usuário é autenticado pelo GSI/Globus. Em seguida, com a ajuda do MDS/Globus, um nó destino é escolhido e notificado da necessidade de execução de uma aplicação Java. Nesse nó remoto existe parte da plataforma SUMA/G que, ao receber essa notificação, busca as classes da aplicação, e a executa.

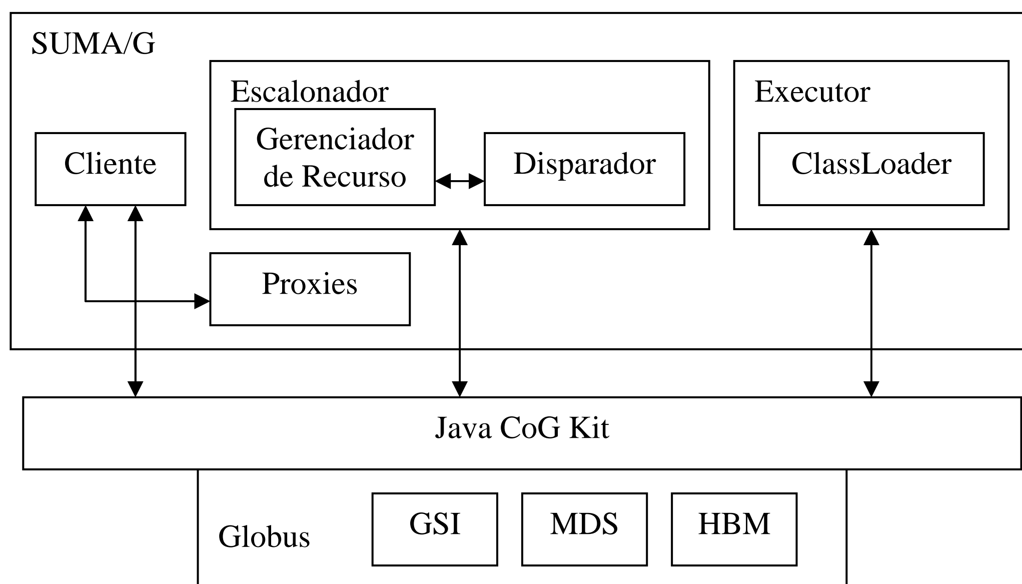


Figura 6 – Arquitetura SUMA/G [67].

A execução de uma aplicação no SUMA/G pode ser classificada como *interativa* e *em lote*. Na execução interativa, as entradas e saídas são transparentemente redirecionadas para o usuário de tal forma que sua percepção é de execução local e as classes são carregadas sob demanda. Na execução em lote, o usuário empacota todos os arquivos de entrada e as classes de execução, envia para a plataforma e, ao final da execução, solicita os arquivos de saída.

Além disso, SUMA/G oferece mecanismos tolerantes a falha (*checkpoint* e *recover*) [68]. Esse mecanismo faz o *checkpoints* da aplicação Java de forma assíncrona e, caso essa aplicação falhe, o mecanismo de *recover* é acionado, restaurando a aplicação a partir do último *checkpoint*.

A integração ao JaDiMa ocorre devido ao mecanismo de gerenciamento de bibliotecas remotas. Quando essas bibliotecas são disponibilizadas, repositórios remotos são escolhidos para o armazenamento das classes e das API (*Application*

Programming Interface) da biblioteca. Automaticamente, o sistema gera *stubs* de todas as classes da biblioteca, resolvendo as dependências de outras bibliotecas remotas. Ao ser executada, a aplicação, dinamicamente, encontrará o repositório da biblioteca utilizada. Em função dessa procura, JaDiMa também possui mecanismos de *cache* e *pre-fetching* de classes durante a execução da aplicação.

1.3.3 ProActive

ProActive[2] [12, 69] é um *middleware* para aplicações paralelas, distribuídas e *multi-threads*. Ele foi desenvolvido inteiramente em Java e oferece ainda mecanismos de segurança e de migração. Por padrão, RMI é utilizado como camada de comunicação dessa infra-estrutura.

Por ter sido desenvolvido sobre a API padrão do Java, não é necessário modificar o ambiente de execução da aplicação, nem mesmo utilizar um compilador especial, um pré-processador ou até mesmo uma JVM modificada.

No ProActive, as aplicações são compostas por objetos de média granularidade denominados *active objects*. Cada *actrive object* possui sua própria *thread* de controle permitindo escolher a qual requisição responder. As requisições são, na verdade, chamada aos métodos desse *active object* e são armazenados em uma fila de requisições pendentes.

Quando um método de um *active object* é invocado, uma referência à resposta desse método é criada. Essa referência é denominada *future object* e será usada para acessar a resposta. Do ponto de vista do programa cliente, essa chamada é assíncrona e transparente, como se fosse um objeto local. A requisição, por sua vez, fará parte da fila de espera desse *active object*. Isso permite que o método chamado e a aplicação cliente sejam executados paralelamente. Caso um método do *future object* seja chamado, mecanismos de sincronização bloquearão o acesso até que a resposta esteja disponível. Esse mecanismo é denominado de *wait-by-necessity* [69].

Os parâmetros existentes na chamada de um método de um *active object* são passados como cópia profunda (*deep-copy*). Nessa cópia, todas as referências

[2] O projeto pode ser acessado no site http://proactive.inria.fr/ .

existentes são clonadas hierarquicamente, até completar a árvore de dependência. Enquanto essa cópia não é feita, o programa cliente ficará em espera.

Active objects também possuem a habilidade de migrar. A migração é uma característica importante para aplicações que precisam de balanceamento de carga. Através dessas características de migração, é possível estabelecer mecanismos de tolerância a defeitos, com *checkpoint* e *recover*.

Além da comunicação ponto-a-ponto, os *active objects* podem fazer parte de grupos, permitindo a comunicação em grupo. Nessas chamadas, os parâmetros dos métodos são enviados para cada um dos métodos dos objetos pertencente ao grupo. O *future object* também pode ser um grupo de objetos, com as mesmas restrições do mecanismo *wait-by-necessity*. Um exemplo dessa comunicação em grupo pode ser visto na Figura 7.

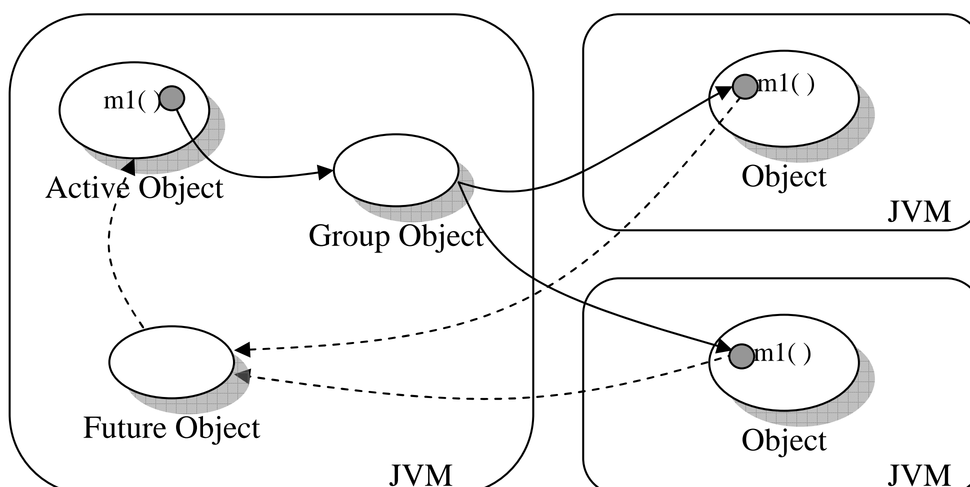


Figura 7 – Comunicação em grupo no ProActive [12].

1.3.4 Globus

Globus[3] [6] é uma infra-estrutura de software que permite as aplicações enxergar recursos computacionais heterogêneos e distribuídos como uma única máquina. Seu elemento central é o Globus Toolkit (GT), o qual define serviços necessários para a construção de um *grid*. Esse *toolkit* é um conjunto de componentes organizados em

[3] O projeto pode ser acessado no site http://www.globus.org/ .

camadas e orientados a serviços, oferecendo serviços básicos como descoberta e gerenciamento de recursos, movimentação de dados, segurança, dentre outros [70]. Sua arquitetura é organizada em camadas de tal forma que os serviços das camadas de alto nível sejam definidos a partir dos serviços oferecidos pelas camadas de mais baixo nível. Isso permite que as aplicações ou ferramentas que utilizam o *grid* também sejam construídas a partir dos diversos níveis de serviços oferecidos.

Entre os componentes existentes nesse *toolkit*, destacam-se o GRAM (*Grid Resource Allocation Management*), RFT (*Reliable File Transfer*), MDS (*Monitoring and Discovery System*), GridFTP. A Figura 8 apresenta a arquitetura e os componentes desse *toolkit*.

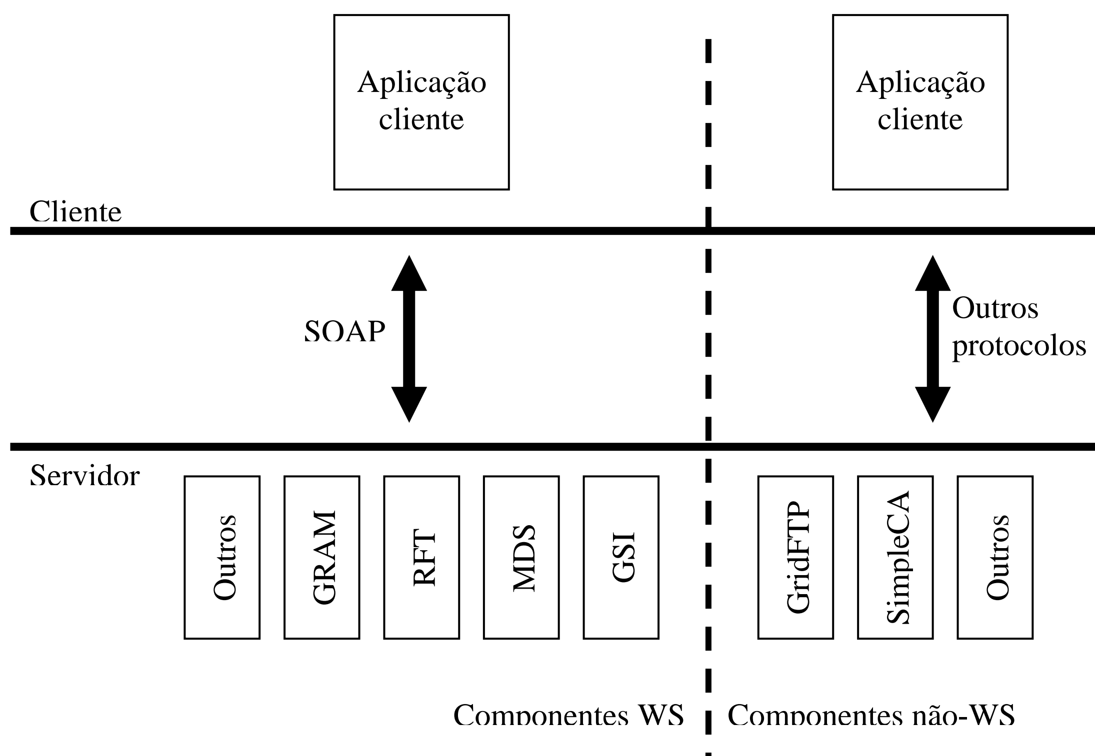


Figura 8 – Arquitetura de componentes do *Globus Toolkit* [6].

Na última versão (GT 4.0), os serviços são oferecidos através dos conceitos de *WebServices* (WS) e do padrão WSRF (*WebService Resource Framework*). Apesar disso, ainda existem serviços oferecidos fora desse padrão, como o próprio GridFTP, devido à sua natural evolução.

Esse sistema oferece mecanismos de segurança através de certificados digitais, controlados pelo GSI (*Grid Security Infrastructure*). Portanto, para que uma aplicação

cliente utilize esse ambiente, é necessário que esteja devidamente autenticado. Estes certificados seguem o padrão X.509.

Outro componente importante é o GRAM. Ele gerencia a execução das tarefas alocadas para o *grid*. Esse gerenciamento consiste em iniciação, monitoramento, gerenciamento, escalonamento, e coordenação dessas tarefas.

1.3.5 Considerações sobre *Computational Grid* e Java

Um dos grandes trabalhos sobre *Grid* foi feito por Ian Foster e Carl Kesselman e apresentado no livro sobre esse assunto, impulsionando a pesquisa nessa área [1]. Este trabalho também consolidou seu *middleware* Globus [6]. A partir de então, não só o meio acadêmico, mas também empresas começaram a utilizar esse *middleware* para dar suporte ao ambiente de *grid* que cada uma delas necessitava [70].

Atualmente, a lista de trabalhos envolvendo os conceitos de *grid* é realmente grande. Parte dessa lista pode ser vista no portal de *grid* mantido por Rajkumar Buyya, outro pesquisador sobre *grid* [71, 55]. Diversos desses trabalhos são desenvolvidos ou possui parte de seu desenvolvimento em Java: *middlewares* (Globus, Gribus), *data grids* (Vlab), sistemas de *grids* (Javelin, JaDiMa), escalonadores (Grid SBroker), portais (JClarens), ambiente de desenvolvimento (ProActive, JaDiMa, GAF4J, Java CoG, DPPEJ, G4JEE), dentre outros trabalhos e categorias que podem ser vistos nesse portal e em publicações [1, 6, 13, 70, 71, 72].

Boa parte desses trabalhos sofre a influência das principais definições do Globus, como, por exemplo, no uso de *webservices*. Essa influência é resultado da ampla divulgação e, conseqüentemente, do seu uso nos projetos de *grids* existentes [70]. De certa forma, essa disseminação facilita o estudo de assuntos específicos para *Grid*, além de também disseminar o uso dos recursos e serviços de *grid* em outras áreas de pesquisa [1, 54].

Em particular, considerando soluções orientadas a objetos, poucos trabalhos apresentam um ambiente de *grid* para esse paradigma. E esses ambientes ainda necessitam que o projetista ou desenvolvedor intervenham diretamente na solução. As formas de intervenções necessárias dos principais ambientes são apresentadas na Tabela 2.

Além dessas características, é importante notar que para a comunicação efetivamente acontecer é necessário que todos os nós do *grid* possuam endereçamento público, ou seja, sejam válidos na *Internet*.

Tabela 2 – Resumo das características dos ambientes orientados a objetos em *computational grid***.**

	Javelin [65]	**JaDiMa [68]**	**ProActive [69]**
Comunicação	*socket* (http)	passagem de mensagem (MPI)	RMI
API	própria (ex. classe *JavelinClient*)	projeto *mpiJava*	própria (ex. classe *ProActive*)
Migração	inexistente	inexistente	através da classe *ProActive*
Tolerância a defeitos	inexistente	*checkpoint/recover*	*checkpoint/recover*
Infra-estrutura	própria	SUMA/G	Globus, LSF, PBS, dentre outros
Hierarquia	em níveis	inexistente	inexistente

2 AMBIENTE JAVA PARA GRADES COMPUTACIONAIS

A partir dos estudos realizados nas plataformas e ambientes de *grids* voltados para a Orientação a Objetos (OO) e Java [64, 13, 12], apresentados no capítulo anterior, é possível detectar alguns pontos que estes ambientes e ferramentas não discutem com profundidade:

- Os nós participantes de uma solução *grid* devem possuir endereçamento público. Isso significa que *clusters* de computadores já existentes, com nós de endereçamento privado, não conseguem fazer parte, diretamente, de um *grid* devido às restrições administrativas e de segurança, além do próprio limite existente no endereçamento do protocolo IPv4, contrariando a idéia inicial de alta conectividade entre dispositivos computacionais [1];
- A re-implementação de uma solução em *grid* necessita de alterações no código-fonte e/ou até mudança de paradigma. A manutenção do código-fonte de um sistema em funcionamento traz novos desafios para a validação do novo sistema gerado [73]. Além disso, o custo pode ser ainda maior se a manutenção desse sistema sofre a influência de outros paradigmas, como por exemplo, o de passagem de mensagem;
- O uso de bibliotecas e arquivos de configurações é necessário para o correto comportamento do sistema. Um ambiente integrado de *grid* deve permitir a escolha do recurso para execução da aplicação (*stand-alone*, *cluster* local, hierárquico, etc) de forma transparente para o programador e para a própria aplicação, diferentemente de como são oferecidos pelos ambientes e plataformas atuais. Nestes, os arquivos descritivos são estáticos e apresentam estruturas complexas para representação dos recursos;
- A menor granularidade de um objeto deve ser o próprio objeto. Apesar da maioria dos ambientes OO definir um objeto como sendo sua menor granularidade, é possível ainda subdividir um objeto em atributos e operações [74] e trabalhar de forma eficiente com cada uma dessas partes.

Além desses pontos, a construção de uma DJVM específica para um ambiente *grid* se torna inviável devida a alguns fatores: (i) a manutenção da compatibilidade com os padrões definidos pela JCP (*Java Community Process*) [21] para a

linguagem Java dificulta a construção de versões estáveis de DJVMs; (ii) é necessário construir ambientes de execução para cada recurso do *grid*, uma vez que cada recurso pode possuir diferentes sistema operacionais e arquiteturas de *hardware*; (iii) a criação de compiladores JIT deve ser específico para cada recurso existente no *grid*, com estratégias particulares de compilação para cada plataforma de *hardware*; (iv) a integração com um *middleware* de *grid* é complexa, sabendo que a DJVM deverá oferecer suporte de execução e comunicação paralela e distribuída; (v) retirar as decisões sobre o comportamento da aplicação no *grid* das mãos do programador pode prejudicar o desempenho da aplicação; dentre outros fatores que podem ser encontrados na literatura [48].

Dessa forma, este trabalho apresenta uma solução para os pontos discutidos anteriormente sobre *grid*. Esta solução também não é uma DJVM, pois não apresenta as restrições existentes em uma JVM distribuída, mas utiliza técnicas de engenharia de *bytecodes* em conjunto com as JVMs [75].

A solução proposta oferece distribuição da aplicação pelo *grid* usando os conceitos da orientação a objetos: (i) criação e acesso a objetos distribuídos; (ii) execução de métodos em diferentes nós; e (iii) armazenamento de atributos, apenas no caso em que são objetos, em diferentes recursos no *grid*.

Além disso, devido à construção de *threads* através de herança de objetos em Java, esta solução oferece ainda mecanismos de execução paralela através da criação de *threads* remotas nos recursos computacionais do *grid*.

E, nessa solução, a referência para os objetos, a chamada de métodos, o acesso aos atributos, e a criação de objetos e *threads* remotos são feitos de forma transparente pelo programador durante a construção de uma aplicação.

Por isso, para a construção da solução proposta, foi desenvolvido um ambiente no qual foi necessário:

- especificar e desenvolver uma camada de comunicação entre as diversas partes de uma aplicação distribuída pelo *grid*;
- modelar as novas construções das classes e de seus relacionamentos para comportar referências de objetos remotos e chamadas de métodos remotos;
- definir a sintaxe dos metadados que serão utilizados durante o desenvolvimento de um programa orientado a objetos, a partir das técnicas de anotação do Java (*annotation*);

- construir agentes de instrumentalização que atuarão nos *bytecodes* da aplicação, juntamente com a JVM (*Java Virtual Machine*).

Neste capítulo é apresentado esse ambiente para programação orientada a objetos distribuídos e paralelos em grades computacionais (*computational grid*), denominado J4GE (*Java for Grid Environment*). O capítulo é dividido da seguinte forma: a primeira seção apresenta a infra-estrutura de *grid* necessária para a execução de uma aplicação; a segunda seção descreve as principais características do ambiente bem como sua arquitetura no *grid*; a terceira seção apresenta o desenvolvimento do ambiente; e, por fim, a quarta seção apresenta algumas considerações sobre o ambiente J4GE.

2.1 INFRA-ESTRUTURA DE *GRID*

Uma infra-estrutura simples de *grid* pode ser composta por computadores interconectados por uma rede de alta velocidade que, eventualmente, também utiliza a *Internet*. A Figura 9 mostra um exemplo dessa infra-estrutura, na qual se pode perceber o uso de um *middleware* de *grid*. Neste trabalho, a denominação *middleware* de *grid* representa um conjunto de *softwares* básicos e ferramentas que garantam o funcionamento e uso de um *grid* [15, 76, 77].

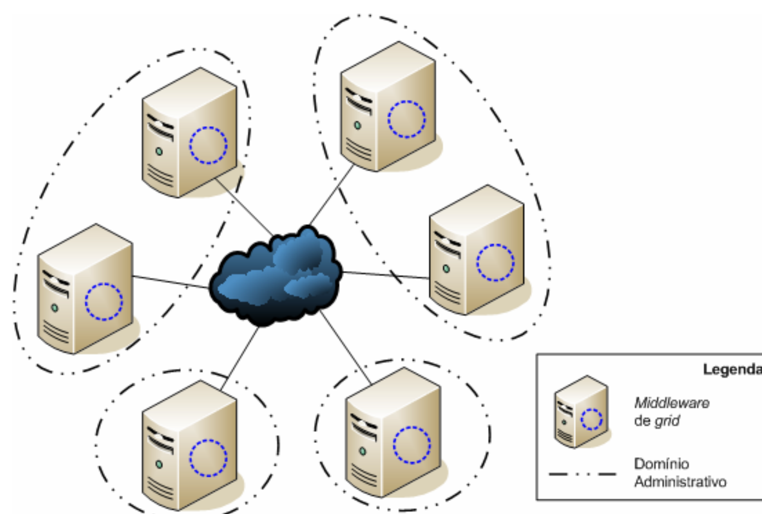


Figura 9 – Infra-estrutura simples de computadores em um *grid*.

Ainda nesse exemplo, todos os computadores possuem acesso e podem ser acessados diretamente através da *Internet*. Essa é uma característica fundamental para que recursos sejam oferecidos em um *grid*.

Esses recursos podem pertencer a diversas empresas, departamentos, setores, ou quaisquer outras subdivisões organizacionais que, por motivos administrativos, possuem sua própria infra-estrutura física de redes de computadores e políticas administrativas. Neste trabalho, essas subdivisões serão chamadas de domínio administrativo, ou simplesmente, Domínio. Na Figura 9, os Domínios são delimitados pelas linhas tracejadas.

Esses diversos Domínios participam do *grid* de forma cooperativa, visando o compartilhamento de seus recursos entre si, formando uma Organização Virtual (VO) [1]. Uma Organização Virtual possui novas políticas administrativas de uso e segurança, porém não se sobrepõem às políticas locais de cada Domínio. As discussões e problemas apresentados neste trabalho se limitam ao arranjo dos recursos compartilhados dentro de uma VO.

Um Domínio pode oferecer não apenas computadores, mas também *clusters* de computadores como recursos para um *grid*. Um *cluster* é gerenciado localmente por ferramentas especializadas como PBS [78], Condor [79], Loadleveler [80], dentre outros LRM (*Local Resource Management*), e possuem sua própria política de controle, segurança e uso. Dessa forma, a infra-estrutura do *grid* pode ser organizada em níveis de acesso, como mostra a Figura 10.

Nesse exemplo, é possível perceber os dois níveis existentes no *grid*. O primeiro nível é composto pelos computadores que estão diretamente conectados à rede *Internet*. Já o segundo nível é formado pelos *clusters* de computadores, ou até mesmo computadores, que não podem ser acessados diretamente através da *Internet*. Esses recursos de segundo nível possuem endereçamento privado e somente podem ser acessados através de uma máquina de entrada, por onde as requisições e tarefas são submetidas. Normalmente, essas máquinas de entrada são acessadas através de portais e são responsáveis pela interface entre dois níveis adjacentes.

Dessa forma, uma infra-estrutura de *grid* mais complexa pode ser organizada em diversos níveis, contendo não só computadores, mas também *clusters* de computadores. Apesar desse arranjo, tarefas construídas para execução paralelas em *grid* somente podem ser executadas em computadores do primeiro nível ou

somente em um único recurso interno, como um *cluster*. Isso pode ser verificado nos ambientes atuais, orientados a objetos ou não, como o Condor/G [81], MPICHG2 [82], JaDiMa [68], ProActive [69], dentre outros, que utilizam apenas alguns serviços do *middleware* de *grid*, como autenticação, comunicação, segurança, e não priorizam o uso de todos os recursos existentes e disponíveis em um *grid*.

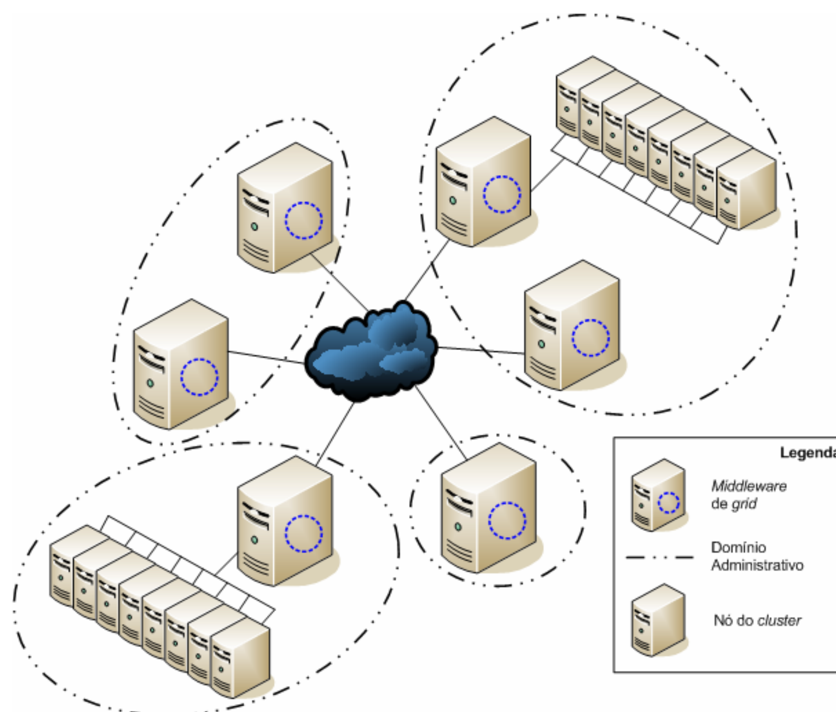


Figura 10 – Infra-estrutura de *grid* com *cluster* de computadores em níveis de acesso.

Para permitir que aplicações paralelas em *grid* utilizem todos seus recursos, em qualquer nível, e ainda assim consigam se comunicar, foi especificado um componente que permite a interação entre as partes da aplicação. Esse componente faz parte do ambiente J4GE, como mostra a seção seguinte.

2.2 J4GE – *JAVA FOR GRID ENVIRONMENT*

J4GE é um ambiente Java para programação e execução de aplicações orientadas a objetos distribuídos e paralelos. Através deste ambiente é possível ter um SSI de tal forma que a aplicação não precisa ter conhecimento sobre os detalhes do ambiente de execução. Portanto, para o desenvolvimento da aplicação, o

programador utiliza os recursos convencionais da linguagem Java como, por exemplo, *threads* para múltiplas linhas de execução.

Esse ambiente utiliza a orientação a objetos devido às características desse paradigma como, por exemplo, encapsulamento, herança, polimorfismo, dentre outras [74, 83]. Essas características são fundamentais para a correta construção de uma aplicação dentro desse paradigma e podem ser utilizados também em um ambiente de *grid* [10].

Sabe-se também que essas características são importantes para a definição de outros paradigmas [84], como, por exemplo, a orientação a aspecto [85], a orientação a serviços [84] e o desenvolvimento baseado em componentes [83, 86]. Dessa forma, é possível estender as construções existentes no J4GE para contemplar as características desses e de outros paradigmas.

Portanto, a aplicação deve ser desenvolvida usando os conceitos da orientação a objeto existentes na linguagem Java. E, durante sua construção, não há a necessidade de se conhecer ou aprender uma nova biblioteca especializada em *grid* ou programação paralela, como existe no *middleware* ProActive [12], no projeto JaDiMa [13] ou qualquer outro ambiente para *grid* baseado em bibliotecas. A construção segue os conceitos da própria linguagem Java adicionando apenas metadados, também conhecido como anotações (*annotations*) [87]. Essas anotações informam como as classes, os métodos e os atributos se comportarão em um ambiente de *grid*, não alterando o algoritmo de sua execução.

O ambiente, durante a execução das aplicações, identifica as anotações realizadas no código e adapta seu comportamento para o *grid*, utilizando também o Serviço de Mensagem. Esse processo de identificação é realizado juntamente com a JVM nativa. Apesar dessa ligação com a máquina virtual, o ambiente possui um baixo grau de acoplamento, garantindo sua independência da JVM, porém mantendo sua coesão com a especificação definida para esse processo de identificação [88].

Nas seções seguintes são apresentados o comportamento da aplicação e a arquitetura do J4GE.

2.2.1 A Aplicação no Ambiente J4GE

O desenvolvimento de aplicações orientadas a objetos no ambiente J4GE se diferencia dos demais trabalhos devido à sua facilidade em demonstrar quais partes da aplicação devem ser distribuídas pelo *grid*. Além disso, da mesma forma que a construção de *threads* é de responsabilidade do desenvolvedor, a identificação dessas partes na aplicação para o ambiente J4GE também é de responsabilidade do programador. Por isso, o ambiente oferece mecanismos simples para a identificação dessas partes. Além disso, o ambiente fica responsável pelo escalonamento, pela distribuição, pela manutenção na transparência durante o acesso aos objetos remotos, e por diversas outras características que esse ambiente possui.

Os mecanismos que identificam as partes da aplicação são descritivos, não interferindo na lógica ou algoritmo utilizado na aplicação. Essa técnica é denominada de metadados que, no Java, também é conhecida como anotações (*annotation*) [87]. Dessa forma, não há a necessidade de conhecimentos específicos de bibliotecas para *grid* ou de paradigmas para sistemas distribuídos.

Nesse ambiente, são três os mecanismos de anotação utilizados para distribuir uma aplicação: classe/objeto, método e atributo. A Figura 11 mostra esses três mecanismos em uma única classe.

```
01 import org.j4ge.Remote;
02
03 @Remote
04 public class Classe {
05
06         public Object atributo1;
07
08         @Remote
09         public Object atributo2;
10
11         public void metodo1() {
12                 // ...
13         }
14
15         @Remote
16         public void metodo2() {
17                 // ...
18         }
19 }
```

Figura 11 – Sintaxe única para anotação de código-fonte, no J4GE.

O mecanismo de anotação em classe/objeto indica que as instâncias de objetos devem ser criadas de forma distribuída. Cada instância é alocada em um nó do *grid*,

e sua referência é mantida de forma transparente pelo ambiente J4GE. Esse mecanismo é ideal para utilizar paralelismo na aplicação, instanciando objetos que tenham o comportamento de *thread*. Dessa forma, cada *thread* será criada em um nó diferente do *grid*.

O mecanismo de anotação em método encapsula sua execução remota. Isso permite que, dado um objeto, apenas aquele método anotado seja executado em um nó remoto do *grid*. Esse mecanismo é ideal para execução de métodos que necessitam de alto poder de processamento.

Por fim, o mecanismo de anotação em atributo indica que aquele atributo deve ser alocado em um nó remoto do *grid*. Isso permite que atributos que necessitam de muito recurso, como espaço de memória, sejam alocados em nós que satisfazem suas necessidades, sendo mantida a transparência no acesso às suas operações pelo ambiente J4GE.

Utilizando os mecanismos do ambiente J4GE e as bibliotecas disponíveis na plataforma Java [23], o programador consegue também que sua aplicação possua, além de objetos distribuídos, *threads* distribuídas, conforme mostra a Figura 12.

```
01  import org.j4ge.Remote;
02
03  @Remote
04  public class MinhaThread extends Thread {
05
06          public Object atributo;
07
08          @Override
09          public void run() {
10                  // ...
11          }
12
13          public synchronized void metodo() {
14                  // ...
15          }
16  }
```

Figura 12 – Classe anotada para execução paralela através do ambiente J4GE.

Essa figura mostra uma *thread* sendo definida a partir de uma classe de biblioteca Java, juntamente com um mecanismo de anotação do J4GE, com o comportamento descrito no método de execução (*run()*), conforme especificado na plataforma Java [21].

É importante ressaltar que mecanismos de sincronização (*synchronized*) são de responsabilidade do programador e, se estiverem presentes no código, o J4GE mantém seu funcionamento e coerência.

Para o uso dessa classe, a aplicação utiliza os conceitos da própria linguagem Java, como instanciação, referência a objetos e chamada de métodos. Essa aplicação pode ser vista na Figura 13.

```
01  public class Principal {
02
03        public static void main(String[] args) {
04              MinhaThread[] threads = new MinhaThread[3];
05
06              for (int i = 0; i < threads.length; i++) {
07                    threads[i] = new MinhaThread();
08                    threads[i].start();
09              }
10
11              for (int i = 0; i < threads.length; i++)
12                    threads[i].metodo();
13        }
14  }
```

Figura 13 – Aplicação utilizando objetos distribuídos e paralelos pelo *grid*.

Nesse caso, a aplicação solicita a criação de três *threads* (linhas 04 e 07), que são alocadas pelo J4GE em nós diferentes do *grid* em relação ao nó da aplicação. Em seguida, para cada *thread* remota criada, é dado o sinal de início de sua execução através do método *start()* (linha 08). Ao final, também em cada uma das *threads* remota, é utilizado o método sincronizado *metodo()* (linha 12) para a realização de alguma operação, e a aplicação termina sua execução.

Uma aplicação completa, com uma solução para o problema do Caixeiro Viajante, será discutida no próximo Capítulo.

2.2.2 Arquitetura do J4GE em um *Grid*

Uma vez definida a infra-estrutura de *grid* para a execução de aplicações Java que utilizam o J4GE, a arquitetura completa do ambiente é definida conforme mostra a Figura 14.

Na camada *LRM* (*Local Resource Management*) são instalados os gerenciadores locais dos computadores ou *clusters* de computadores. São nesses gerenciadores que as políticas e os mecanismos de seguranças de cada Domínio são definidos. Eles possuem escalonadores e disparadores próprios, os quais coordenam as tarefas submetidas para cada nó local.

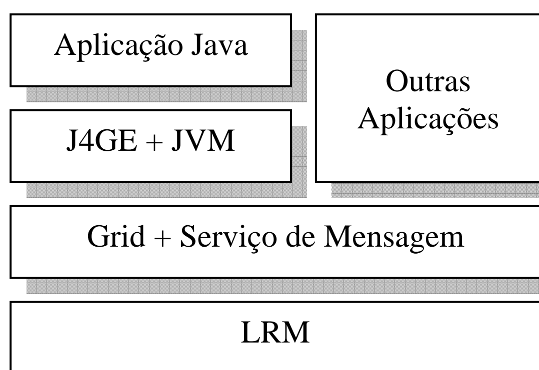


Figura 14 – Arquitetura J4GE em um *grid*.

A camada *Grid + Serviço de Mensagem* representa o conjunto de ferramentas que um *middleware* de *grid* oferece juntamente com o Serviço de Mensagem construído para o J4GE, formando a infra-estrutura básica do *grid*. Cada Domínio necessita dessa infra-estrutura para que uma aplicação orientada a objetos distribuídos e paralelos funcione corretamente no J4GE. É importante comentar que o escalonamento de *grid* utilizado atualmente é manual, ou seja, o próprio usuário escolhe quais recursos serão utilizados pela aplicação. Versões mais recentes do *middleware* para *grids* oferecem escalonadores ou meta-escalonadores, como o Gridway [89], porém estes não satisfazem as necessidades do J4GE para um escalonamento em níveis. Um outro mecanismo de escalonamento que atende essas necessidades foi desenvolvido no laboratório sob orientação da Profa. Liria Matsumoto Sato, no LAHPC[4].

O Serviço de Mensagem é o componente responsável pela transparência na comunicação da aplicação distribuída pelos diversos níveis do *grid*. Este serviço é detalhado na subseção seguinte.

A camada *J4GE + JVM* representa o núcleo do ambiente descrito nesse trabalho, atuando juntamente com a JVM para a execução de aplicações orientada a objetos em *grid*.

A camada *Aplicação Java* representa qualquer aplicação que foi desenvolvida observando as estratégias e anotações definidas pelo ambiente J4GE.

Apesar dessa arquitetura, *Outras Aplicações* que necessitam de uma infra-estrutura de *grid* convencional podem ser executadas nessa arquitetura, pois ela oferece mecanismos básicos para controle e execução de aplicações [2].

[4] *Laboratory of Architecture and High Performance Computing*, POLI/USP: http://regulus.pcs.usp.br/~lahpc

2.2.2.1 Serviço de Mensagem

O Serviço de Mensagem, no J4GE, garante a transparência na comunicação da aplicação distribuída nos diferentes níveis do *grid*. É através deste serviço que as mensagens entre os objetos da aplicação são trocadas. Para isso, este serviço oferece mecanismos de envio e recebimento de mensagem.

O Serviço de Mensagem deve fazer parte do *middleware* de *grid* e, preferencialmente, configurado na máquina entre os níveis da infra-estrutura de *grid*. É nessa interface que esse serviço controla as mensagens que chegam e que saem das aplicações do nível mais interno. Como parte de um *middleware* de *grid*, o Serviço de Mensagem está sujeito às políticas já existentes no *grid*, adicionando apenas a transparência na comunicação de aplicações distribuídas e paralelas.

A arquitetura do Serviço de Mensagem pode ser vista na Figura 15.

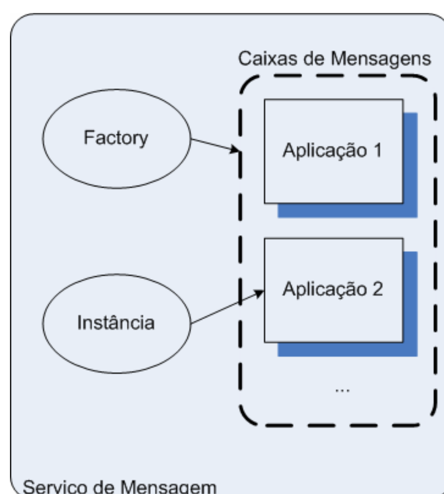


Figura 15 – Arquitetura do Serviço de Mensagem.

O controle dessas mensagens é feito através de Caixas de Mensagens [5], que são criadas para cada aplicação (*Factory*). Como as partes de uma aplicação estão distribuídas pelo *grid*, haverá diversas Caixas de Mensagens, onde cada caixa é responsável por coordenar as mensagens de envio e recebimento entre o nível interno e o nível externo.

O acesso à Caixa de Mensagem é exclusivo de cada aplicação, mantendo o contexto e o histórico de troca de mensagens (*Instância*). É também através desse acesso que a aplicação envia suas mensagens e é notificada de que há novas mensagens em sua Caixa de Mensagem.

Dessa forma, é possível visualizar o Serviço de Mensagem na infra-estrutura de *grid*, conforme mostra a Figura 16.

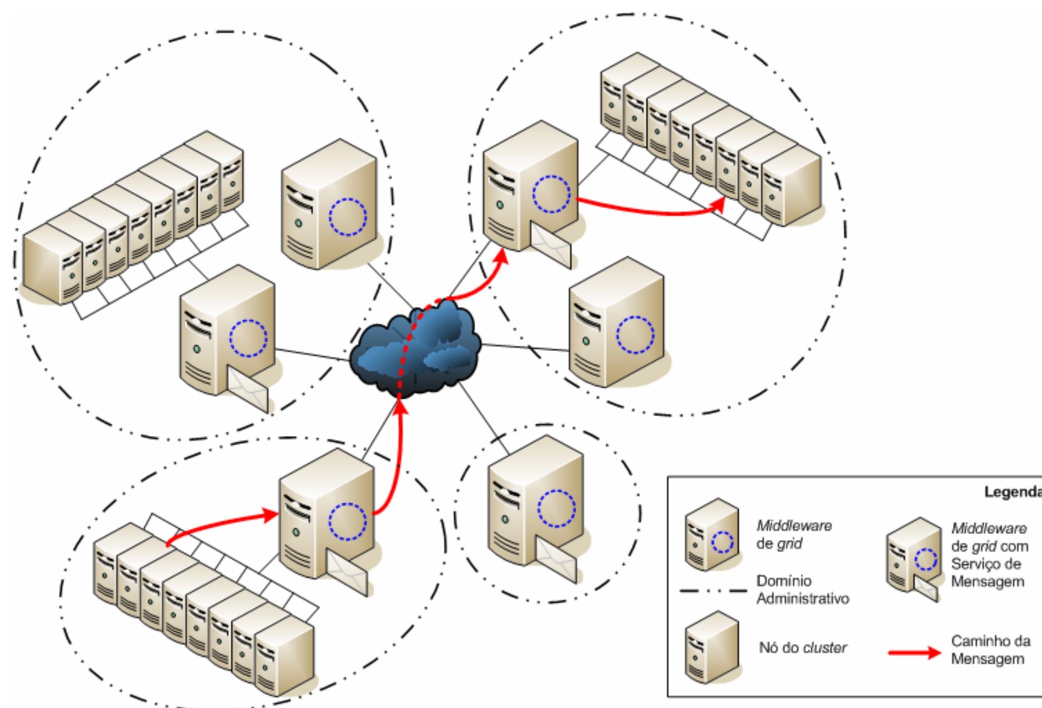


Figura 16 – Caminho de uma mensagem por uma infra-estrutura de *grid* através do Serviço de Mensagem.

Fica evidente que a comunicação entre duas partes da aplicação que estão em nós diferentes necessita da intervenção do Serviço de Mensagem, tanto no envio quanto no recebimento. Isso se deve aos níveis existentes no *grid* em função do endereçamento de redes normalmente utilizada para a construção da infra-estrutura com uma máquina de entrada. Essa máquina possui dois endereços: um interno, privado, e utilizado para acesso pelo *cluster*; e outro externo, público, utilizado para acesso de serviços remotos. Para que o Serviço de Mensagem funcione corretamente, esses endereços devem ser conhecidos.

2.3 DESENVOLVIMENTO DO J4GE

Para o desenvolvimento do ambiente, foi necessário escolher um *middleware* de *grid* que oferecesse recursos básicos para a construção de uma infra-estrutura, de acordo com a arquitetura do J4GE. E, devido à sua ampla divulgação, através de trabalhos e produtos [1], e a sua compatibilidade com o desenvolvimento baseado

em serviços web, o Globus (versão 4.0) foi escolhido para a construção da infra-estrutura desse ambiente [90].

Ainda seguindo a arquitetura do J4GE, foi construído o Serviço de Mensagem e adicionado ao Globus, para garantir a transparência na comunicação entre as partes de uma aplicação distribuída.

Também foi necessário definir as estratégias de distribuição para cada mecanismo de anotação utilizado em um código-fonte. Essas estratégias compõem o núcleo do ambiente, finalizando a infra-estrutura de *grid* necessária para a execução de aplicações utilizando o ambiente J4GE.

As seções seguintes descrevem os itens construídos do ambiente J4GE: o Serviço de Mensagem, as estratégias de distribuição e a engenharia de *bytecodes*.

2.3.1 Serviço de Mensagem

O Globus é um *middleware* de *grid* baseado na tecnologia de *webservices* [4]. Utilizando essa tecnologia, o Serviço de Mensagem foi desenvolvido como um serviço do Globus [90], seguindo o WSRF (*WebService Resources Framework*) [4]. Utilizando o padrão de projeto *factory/instance*, a interface do serviço foi definida. Esse padrão foi utilizado prevendo o intenso uso desse serviço pelas diversas aplicações que estarão sendo executadas em um *grid*. Dessa forma, para cada aplicação, um recurso (*resource*) é criado e gerenciado. Conforme já mencionado anteriormente, esses recursos são as Caixas de Mensagens de cada aplicação.

Ainda através desse padrão de projeto, esses recursos podem ser acessados de forma concorrente por diversas aplicações diferentes, garantindo a integridade do conteúdo de cada caixa de mensagem [4].

A Figura 17 apresenta um diagrama de seqüência que facilita o entendimento do ciclo de vida de um recurso do Serviço de Mensagem para uma aplicação [70, 90]:

1. quando a aplicação for iniciada, ela entra em contato com a classe do padrão *factory* para a criação de um recurso de Caixa de Mensagem específico para a aplicação;
2. para que a criação seja feita, é necessário encontrar o gerenciador de recursos através da classe das ferramentas do Globus: ResourceContext;

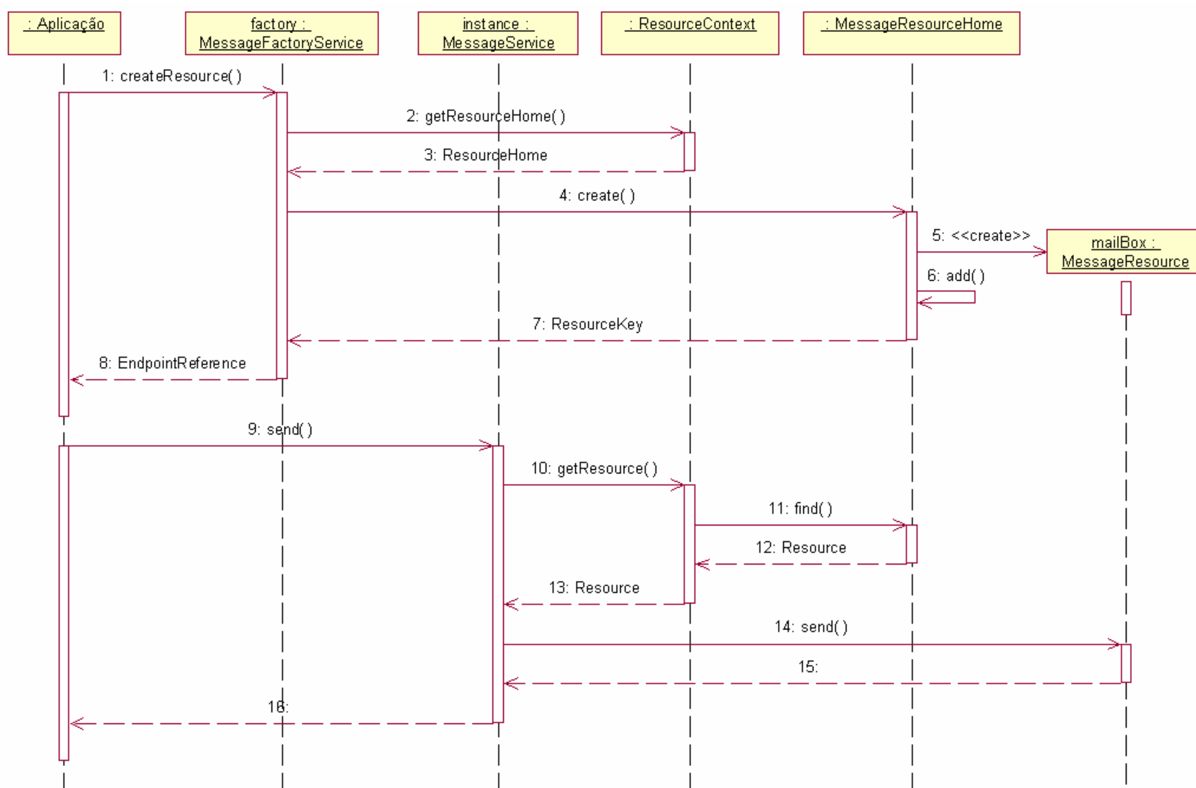


Figura 17 – Diagrama de seqüência para uso de um recurso do Serviço de Mensagem [90].

3. após a pesquisa, o ResourceContext retorna a classe Home que gerencia recursos;
4. através do objeto Home, é possível criar o recurso;
5. na criação, é gerada uma nova instância do objeto de recurso (Caixa de Mensagem);
6. e este objeto é adicionado à lista interna de recursos existentes, para futura consulta;
7. como resultado, uma chave desse recurso é retornado, a qual será utilizada para gerar a URI (*Uniform Resource Identifiers*) para o cliente;
8. por fim, a criação do recurso retorna a URI para o serviço, juntamente com sua chave identificadora.

A Figura 17 também apresenta a seqüência de chamadas para o uso do Serviço de Mensagem e de seu respectivo recurso (Caixa de Mensagem):

9. após a criação, a mesma aplicação invoca um método do serviço, nesse caso, enviando uma mensagem para uma Caixa de Mensagem;

10. para que essa operação seja executada, é necessário buscar pelo contexto correto da aplicação, através da classe ResourceContext;
11. este, por sua vez, entra em contato com o gerenciador de recursos, procurando por uma Caixa de Mensagem (MessageResrouceHome);
12. após encontrado o recurso, este é retornado para o ResourceContext;
13. que também o envia para o Serviço de Mensagem;
14. só então o método é efetivamente invocado, com seus parâmetros;
15. retornando, por fim, o resultado da operação;
16. o qual também é enviado para a aplicação.

Como o Serviço de Mensagem foi construído para a troca de mensagens entre classes e objetos, foi necessário especificar o protocolo de comunicação entre os pontos envolvidos. Parte desse protocolo, descrito através da WSDL (*WebService Definition Language*), pode ser visto na Figura 18.

```
01  <xsd:complexType name="ObjectID">
02        <xsd:sequence>
03              <xsd:element name="host" type="xsd:string" />
04              <xsd:element name="type" type="xsd:string" />
05              <xsd:element name="oid" type="xsd:int" />
06              <xsd:element name="sequence" type="xsd:int" />
07        </xsd:sequence>
08  </xsd:complexType>
09  <xsd:complexType name="Call">
10        <xsd:sequence>
11              <xsd:element name="method" type="xsd:string" />
12              <xsd:element name="static" type="xsd:boolean" />
13              <xsd:element name="types" type="xsd:hexBinary" />
14              <xsd:element name="params" type="xsd:hexBinary" />
15        </xsd:sequence>
16  </xsd:complexType>
17  <xsd:complexType name="Response">
18        <xsd:sequence>
19              <xsd:element name="type" type="xsd:hexBinary" />
20              <xsd:element name="value" type="xsd:hexBinary" />
21              <xsd:element name="ready" type="xsd:boolean" />
22              <xsd:element name="exception" type="xsd:hexBinary" />
23        </xsd:sequence>
24  </xsd:complexType>
25  <xsd:complexType name="MessageCallType">
26        <xsd:sequence>
27              <xsd:element name="application" type="xsd:int" />
28              <xsd:element name="from" type="xsd:string" />
29              <xsd:element name="to" type="xsd:string" />
30              <xsd:element name="oid" type="tns:ObjectID" />
31              <xsd:element name="message" type="tns:Call" />
32              <xsd:element name="response" type="tns:Response" />
33        </xsd:sequence>
34  </xsd:complexType>
```

Figura 18 – Especificação do protocolo do Serviço de Mensagem em WSDL.

O principal elemento presente no protocolo é o *MessageCallType*. É nele que a aplicação pode ser reconhecida (*application*) e, portanto, garantir não só a integridade da caixa de mensagem, mas também que a mensagem será entregue, a partir do nó de origem (*from*), para um nó de destino (*to*), participantes da execução da aplicação. Além disso, a instância de objeto (*oid*), a mensagem a ser enviada (*message*) e a sua resposta (*response*) também são definidas.

É importante ressaltar que a identificação das instâncias de objetos, o elemento *ObjectID*, é fundamental para o correto funcionamento do Serviço de Mensagem desenvolvido, uma vez que o mecanismo de referência de objetos nativo do Java foi substituído. Isso significa que, a cada instância de um novo objeto, um identificador único lhe é atribuído (*oid*), garantindo sua univocidade naquele nó (*host*). Além disso, é importante garantir a seqüência de entrega de mensagens ao objeto (*sequence*) para que não haja inconsistência das mensagens nos nós destinos [5]. Essa inconsistência poderia acontecer se dois nós de origem utilizassem concorrentemente um mesmo objeto remoto, porém, por algum motivo não-determinístico (e.g. tempo de processamento do Serviço de Mensagem, latência de rede), a ordem das mensagens fosse invertida.

A mensagem, um elemento *Call*, é composta do nome do método do objeto (*method*), dos parâmetros (*params*) e dos seus respectivos tipos de dados (*types*). A resposta, um elemento *Response*, é composta pelo resultado da execução do método (*value*) e do seu tipo de dado (*type*). Para garantir mecanismos de sincronização da resposta na aplicação, foi definido um elemento de controle (*ready*). Com esse controle é possível construir uma aplicação com chamadas de métodos síncronos e assíncronos [5]. Por fim, se durante a execução do método acontecer alguma exceção, esta também é transmitida de volta ao nó de origem (*exception*).

Uma mensagem somente é entregue ao nó destino participante de uma aplicação mediante a inscrição e registro de interesse deste nó no Serviço de Mensagem. Esse mecanismo de notificação faz parte da especificação do WSRF, e está baseado no conceito produtor/consumidor [90]. A partir do momento em que se inscreveu, a aplicação do nó destino começa a receber as mensagens da Caixa de Mensagem, desde que elas sejam de seu interesse.

Com esse protocolo, o Serviço de Mensagem oferece a transparência na comunicação entre a aplicação distribuída pelos diversos níveis do *grid*, em diferentes Domínios.

2.3.2 Distribuição da Aplicação

Anotações (*annotations*), ou metadados, são informações adicionais sobre o programa que não afetam o comportamento de sua execução. Porém, ferramentas, bibliotecas e ambientes podem utilizar essas marcações para decidir sobre a execução do código anotado [91].

As anotações são usadas em diversas situações: (i) pelo compilador, para detecção de erros ou suprimindo alertas; (ii) em tempo de compilação ou de implantação, quando as ferramentas analisam essas informações para geração de novos códigos, arquivos de descrição XML, instalação dos códigos, dentre outros; e (iii) em tempo de execução, para controle e gerenciamento da aplicação durante sua execução [87].

O uso de anotações pode ser observado na própria plataforma para ambientes corporativos Java [92], bem como em extensões envolvendo mecanismos de persistência [93] e outros paradigmas [85], e até mesmo em trabalhos da área de *grid* [72].

No J4GE, as anotações são utilizadas para identificar no código como a aplicação deve ser distribuída em um *grid*. Juntamente com o mecanismo de *classloader* padrão do Java [88], é possível identificar quais são as classes que estão sendo carregadas e verificar se elas foram anotadas, de acordo com as definições do J4GE.

Uma vez identificadas essas anotações no código, dá-se início a instrumentalização do código (engenharia de *bytecodes*) [94]. Na verdade, um código anotado será adaptado em tempo de execução para um ambiente de *grid*, reescrevendo parte de seu *bytecode*. Essa adaptação reorganiza o relacionamento entre classes da aplicação criando novas classes dinamicamente e inserindo instruções de acesso às funcionalidades do ambiente J4GE que, consequentemente, usa o Serviço de Mensagem da infra-estrutura de *grid*.

É importante reforçar que as anotações definidas no J4GE não interferem na execução do código, mas oferecem mecanismos para que o programador indique quais são as partes da aplicação que precisam de recursos especiais durante a sua execução. Nessa mesma linha de raciocínio, um aplicativo já construído não sofrerá mudanças na sua estrutura de código, havendo a necessidade apenas de introduzir as anotações indicativas de distribuição pelo *grid*.

Apesar das anotações presentes no código, a JVM, e conseqüentemente o J4GE, podem ignorá-las de acordo com parâmetros de execução, permitindo que a aplicação seja executada normalmente em um ambiente local [88].

No J4GE, há três mecanismos de anotação utilizados para a distribuição da aplicação no *grid*: classe/objeto, método e atributo. Cada um destes mecanismos foi elaborado utilizando estratégias para que a transparência no acesso a objetos e métodos fosse possível durante a execução da aplicação.

Através desses mecanismos, juntamente com as classes da plataforma Java, é possível ter uma aplicação com execução paralela, além de distribuída.

Esses mecanismos e estratégias são descritos nas seções seguintes.

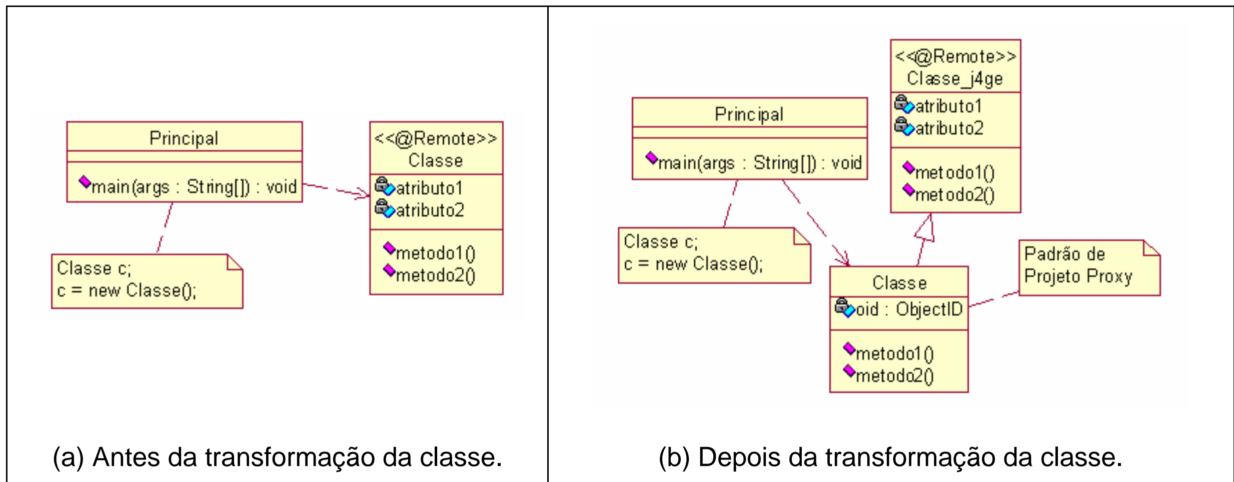
2.3.2.1 Estratégia para Classe/Objeto

Uma classe anotada no J4GE indica que cada instância criada dessa classe deve ser alocada em um nó do *grid*, diferente do nó atual.

Para que esse comportamento seja possível, uma nova classe é escrita, substituindo a primeira e utilizando especialização/generalização (herança) da orientação a objetos. Isso significa que, quando o código original instanciar um objeto da classe anotada, após essa transformação, a instância criada é de um objeto especializado, responsável pelo correto comportamento no *grid* da classe generalista. Essa transformação pode ser vista nos diagramas de classes da Figura 19.

Após a transformação, a classe original recebe um novo nome (acréscimo de *_j4ge*) e a nova classe criada é nomeada como a classe original. Essa nova classe especialista reescreve o comportamento de todos os métodos definidos na classe original, seguindo o padrão Proxy de projeto [95]. Se a classe original também

possuir métodos reescritos devido a uma herança, esses métodos também serão reescritos na classe especialista. Nessa nova classe também existe um atributo (*oid*) que referencia sua instância de objeto remoto. Essa referência é feita de acordo com o protocolo de comunicação do Serviço de Mensagem (Figura 18).



(a) Antes da transformação da classe. (b) Depois da transformação da classe.

Figura 19 – Diagramas de classes com a transformação de uma classe anotada.

Após essa transformação, o ambiente J4GE fica responsável pela alocação de algum nó remoto do *grid* e a criação de uma instância de objeto da classe original neste nó. A referência para esse objeto é definida e armazenada com a instância especialista local (*oid*).

Objetos que são passados e/ou recuperados da classe anotada também devem ser modificados para que o mecanismo de referência em Java continue transparente. A solução para esses objetos é apresentada na Figura 20, com a criação de uma especialização. Nessa especialização, o nome da classe é composto pelo nome da classe original acrescentada de *J4GEAttribute*, e a classe é identificada pela realização da interface *J4GEAttribute*.

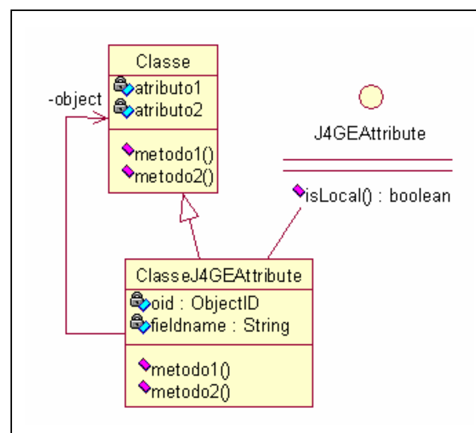


Figura 20 – Diagrama de classes da transformação de uma referência de objeto usado em uma classe anotada.

Esse novo relacionamento entre classes permite que uma referência a um atributo interno de um objeto remoto seja possível. Além disso, é necessário manter essa estrutura de classes até os níveis mais elevados de uma generalização (herança) visando à transparência no acesso a objetos internos (associação, agregação, composição, dentre outros). A identificação dessa hierarquia de classes/objetos é feita através dos atributos *oid* e *fieldname*, trabalhados internamente através de reflexão (*reflection*) [96, 23]. Nos nós envolvidos, a diferenciação da execução dos métodos é feita pelo relacionamento *object* e pelo método *isLocal()*.

O exemplo de código relativo à Figura 19(a) é apresentado na Figura 21.

```
01  import org.j4ge.Remote;
02
03  @Remote
04  public class Classe {
05
06        public Object atributo1;
07
08        public Object atributo2;
09
10        public void metodo1() {
11              // ...
12        }
13
14        public void metodo2() {
15              // ...
16        }
17  }
```

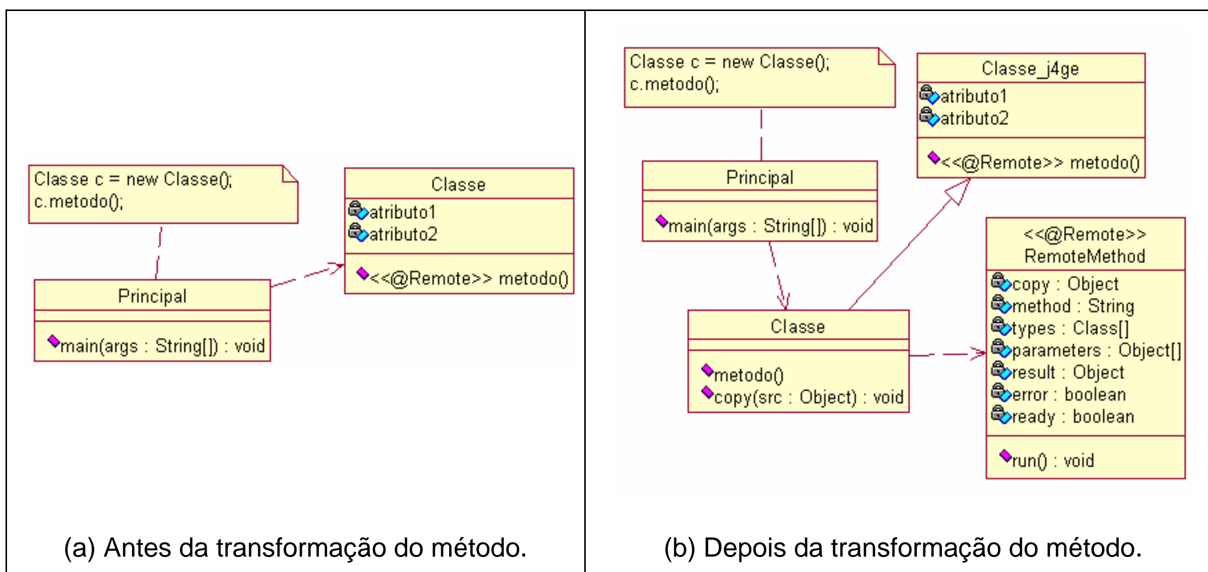
Figura 21 – Código-fonte com anotação em classe para o ambiente J4GE.

2.3.2.2 Estratégia para Método

Um método anotado no J4GE indica que a execução deste método deve ser realizada em um nó do *grid*, diferente do nó atual.

Para não comprometer a execução do método em um nó remoto, é importante que os atributos que esse método tem acesso estejam disponíveis. Portanto, a estratégia de cópia/restaura foi adotada como solução [5]: (i) copia o objeto e seus atributos para o nó remoto; (ii) executa o método do objeto copiado no nó remoto e (iii) copia o objeto remoto e seus atributos de volta para o nó local.

Essa solução foi desenvolvida usando o mecanismo de anotação em classe já construído, conforme mostram os diagramas de classes da Figura 22.



(a) Antes da transformação do método.

(b) Depois da transformação do método.

Figura 22 – Diagramas de classes com a solução de um método anotado.

A partir do momento que a anotação em um método é identificada, uma nova classe é criada com o mesmo nome da classe original, e o nome da classe original é alterado (acréscimo de *_j4ge*), relacionando-as por generalização/especialização. Nessa nova classe, apenas os métodos anotados são reescritos para que utilizem a classe anotada *RemoteMethod*. Para o código original, a instância de objeto criada é da classe especialista, tornando transparente o uso dos métodos da classe generalista.

Ao chamar um método anotado, fica transparente a execução deste em um outro nó do *grid*. Na verdade, quando esse método é chamado, um objeto da classe *RemoteMethod* é criado e, devido a anotação, é alocado remotamente. Para essa instância, é necessário informar a cópia do objeto (*copy*), o método a ser executado (*method*), os parâmetros (*parameters*) e seus tipos de dados (*types*). Remotamente, por reflexão, o método solicitado é executado na cópia do objeto e a classe *RemoteMethod* mantém o estado dessa solicitação (*ready* e *error*), sabendo que o resultado da execução do método ou a exceção de sua execução pode ser retornado (*result*).

Quando o resultado é enviado de volta ao nó, é acionado um método específico para restaurar os valores dos atributos da instância local a partir do objeto remoto. Essa atualização é feita trocando-se todos os atributos locais pelos atributos do objeto remoto, observando também a hierarquia de classes existente (herança).

O exemplo de código relativo à Figura 22(a) pode ser visto na Figura 23.

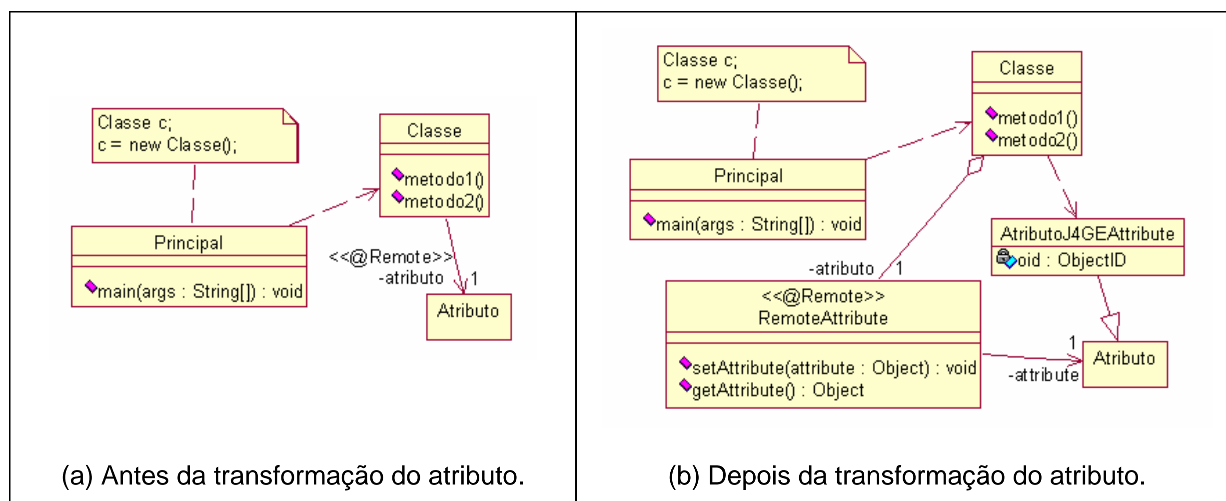
```
01  import org.j4ge.Remote;
02
03  public class Classe {
04
05          public Object atributo1;
06
07          public Object atributo2;
08
09          @Remote
10          public void metodo() {
11                  // ...
12          }
13  }
```

Figura 23 – Código-fonte com anotação em método para o ambiente J4GE.

2.3.2.3 Estratégia para Atributo

Um atributo anotado no ambiente J4GE indica que este atributo deve ser alocado em um nó do *grid*, diferente do nó atual.

Devido aos mecanismos já existentes no J4GE, a estratégia dessa anotação utiliza anotação de classe, conforme pode ser visto na Figura 24.



(a) Antes da transformação do atributo.

(b) Depois da transformação do atributo.

Figura 24 – Diagramas de classes com a solução para anotação de atributo.

O código da classe original é alterado para que o atributo anotado seja substituído pela classe *RemoteAttribute*. Quando esta classe for instanciada, conforme o mecanismo de classe anotada, o objeto será alocado remotamente, juntamente com o atributo anotado da classe original. Dessa forma, quando a classe original acessar o atributo, na verdade ela acessa uma referência para um atributo remoto através de uma classe especializada *J4GEAttribute*, recuperada da classe *RemoteAttribute*.

O exemplo do código relativo à Figura 24(a) pode ser visto na Figura 25.

```
01  import org.j4ge.Remote;
02
03  public class Classe {
04
05          @Remote
06          public Atributo atributo;
07
08          public void metodo1() {
09                  // ...
10          }
11
12          public void metodo2() {
13                  // ...
14          }
15  }
```

Figura 25 – Código-fonte para anotação em atributo para o ambiente J4GE.

2.3.3 Engenharia de *Bytecodes*

A instrumentalização dos *bytecodes* da aplicação Java é feita durante o carregamento das classes pela JVM [88]. Essas classes são analisadas utilizando a ferramenta JavaAssist [97], e reescrita de acordo com as estratégias apresentadas na seção anterior. Porém, além dessas estratégias, é necessário definir o ponto inicial de execução da aplicação no *grid*.

Essa escolha é baseada em mecanismos internos de eleição. Nesta versão do ambiente, o nó que receber primeiro a aplicação deve executar sua parte inicial. Esse nó é identificado pelo ambiente quando a aplicação é executada em qualquer um dos nós. Dessa forma, os demais nós terão o código da aplicação alterado para que apenas reaja às mensagens enviadas pelas outras partes.

A aplicação também é alterada para que possua duas estruturas de execução: uma responsável pelo recebimento de mensagens e outra responsável pelo recebimento de respostas a mensagens enviadas anteriormente. Essas duas estruturas utilizam o mecanismo de inscrição/notificação no Serviço de Mensagem.

Independente da parte da aplicação, o ambiente gerencia as classes instrumentalizadas, evitando a re-análise de classes já existentes e solicitando a criação de classes necessárias (i) para a execução de uma mensagem, (ii) para o envio de uma resposta, ou (iii) para o recebimento de uma resposta.

Nesse processo de intrumentalização, em todas as classes da aplicação, inclusive nas hierarquias (herança) e nos atributos, é adicionada uma interface que permite a

serialização das instâncias de objetos (*Serializable*) [23]. Esse mecanismo foi utilizado para que o Serviço de Mensagem não precise conhecer as classes, requisito da JVM, mas apenas tenha um vetor de *bytes* como parte da mensagem (como mostra a Figura 18). Dessa forma, o ambiente é responsável pelo empacotamento de objetos e o desempacotamento para objetos. Neste último, a identificação de classes que ainda não existem é importante para que a aplicação utilize corretamente as classes que são geradas dinamicamente.

2.4 Considerações sobre o Ambiente J4GE

O J4GE possui algumas vantagens em relação aos ambientes atuais de programação orientada a objetos para *grid*. Dentre essas vantagens, pode-se citar: (i) facilidade de desenvolvimento e manutenção de aplicativos, utilizando apenas anotações para sinalizar os mecanismos de distribuição pelo *grid*; (ii) transparência no acesso a objetos e métodos das estruturas que foram distribuídas pelo *grid*, não necessitando de classes ou bibliotecas explícitas para esse acesso; (iii) utilização do mesmo código-fonte para execução local ou execução no ambiente de *grid*, de acordo com as necessidades do usuário; e (iv) execução da aplicação nos diversos níveis do *grid* (*clusters* de computadores e computadores, em diversos Domínios), havendo transparência na comunicação entre a aplicação.

É importante ressaltar que a decisão sobre quais serão as estruturas distribuídas ainda é responsabilidade do programador, na mesma proporção que a criação de *threads* também é sua responsabilidade. Porém, essa decisão não interfere na lógica ou estrutura do código da aplicação, mas permite que essas estruturas sejam melhor alocadas no *grid*.

Esse processo de alocação de recursos é feito em duas etapas. Na primeira, são escolhidos os nós onde a aplicação será executada. Esta etapa é feita através das ferramentas que um *grid* oferece como listagem dos nós do *grid*, cópia dos arquivos necessários para execução, submissão da solicitação de execução, dente outras ferramentas existentes no *middleware* para *grid*.

A segunda etapa é realizada pelo próprio J4GE. Uma vez reservado os nós no *grid* para a execução da aplicação através das ferramentas do *grid*, é

responsabilidade do ambiente a escolha e a alocação dos objetos entre esses nós pré-alocados. Isso significa que o J4GE pode utilizar mecanismos de escalonamento para a atribuição dos objetos, métodos e atributos nestes nós do *grid*.

Além disso, o Serviço de Mensagem, necessário para uma infra-estrutura de *grid* suportar o J4GE, permite que a comunicação entre os objetos distribuídos seja realizada de forma transparente.

A partir do momento que essa transparência permite a referência a objetos, mesmo estes estando em nós remotos, o acesso concorrente pode existir, havendo a necessidade de seções críticas durante esse acesso. Portanto, é responsabilidade do programador utilizar os mecanismos de monitor ou semáforo existentes na plataforma Java, garantindo a exclusividade de acesso aos objetos, métodos e atributos dentro desse ambiente. De forma geral, se estes mecanismos já foram utilizados em um código-fonte, o J4GE apenas os adapta para acesso distribuído pelo *grid*, garantindo seu funcionando neste novo ambiente.

3 TESTES E ANÁLISE DE RESULTADOS

Para validar a proposta do ambiente de programação orientado a objetos, as estratégias de distribuição e a transparência na comunicação entre objetos no ambiente J4GE, foi construída uma infra-estrutura de *grid* usando máquinas de dois Domínios administrativos diferentes. A Figura 26 apresenta essa infra-estrutura.

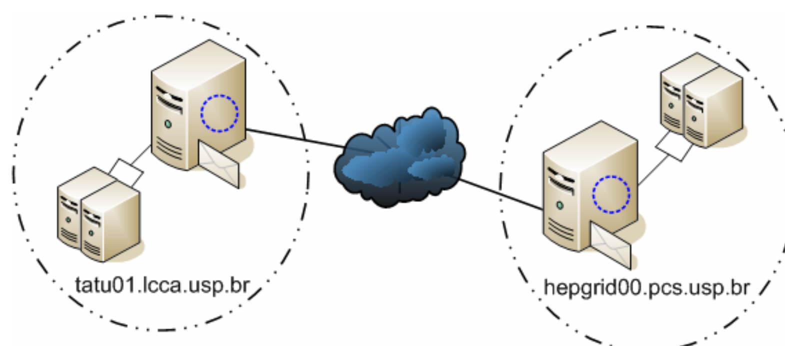


Figura 26 – Infra-estrutura de *grid* para validação do ambiente J4GE.

O Globus Toolkit [6] foi usado como *middleware* de *grid* para essa infra-estrutura e o Serviço de Mensagem foi adicionado aos serviços oferecidos pelo Globus. Cada Domínio possui um *cluster* de computadores como recurso compartilhado para o *grid*, sabendo que, nessa configuração, a máquina de entrada também faz parte do *cluster*. A descrição detalhada desses recursos é apresentada na Tabela 3.

Tabela 3 – Descrição dos recursos disponíveis em cada Domínio administrativo para a infra-estrutura de *grid* usada na validação do J4GE.

Domínio administrativo	Recurso compartilhado	Especificação dos nós	LRM
hepgrid00.pcs.usp.br	*cluster* de computadores	3 x Dual-XEON 2Ghz, 2Gb RAM	Condor [79]
tatu01.lcca.usp.br	*cluster* de computadores	3 x Pentium 4 HT 3Ghz, 512Mb RAM	PBS [78]

Para a execução dos testes de validação do ambiente, foram considerados disponíveis os seis nós existentes no *grid*. Dessa forma, usando as ferramentas de envio e submissão de tarefas do Globus, a aplicação foi enviada para todos esses nós.

Em seguida, deu-se início aos testes do ambiente. Esses testes foram divididos em duas etapas. A primeira etapa consiste em validar a transparência no acesso aos

objetos da aplicação, considerando os principais relacionamentos que existem no paradigma orientado a objetos. A segunda etapa consiste em acompanhar o desempenho de aplicações distribuídas e paralelas, considerando as construções orientadas a objetos, a troca de mensagens entre esses objetos e pontos de sincronização em dados compartilhados.

3.1 VALIDAÇÃO DO MODELO ORIENTADO A OBJETOS

Para validação das construções orientadas a objetos e do acesso transparente às instâncias de objetos foram especificados diversos casos de testes de tal forma que estes casos explorassem o comportamento dos objetos no ambiente J4GE distribuídos pelo *grid* [84].

Com o resultado dos casos de testes, é possível saber também se o processamento das informações foi realizado de forma correta [84].

Para a construção dos casos de testes, foram utilizados os três tipos básicos de relacionamento entre classes e objetos: dependência, generalização e associação [74].

Apesar desses três tipos, ainda foram necessários outros casos de teste para validar o modelo orientado a objetos. Devido às anotações de execução de método remoto e de alocação de atributo remoto, foram criados novos casos de testes para validar o comportamento das classes da aplicação e, conseqüentemente, da estratégia de distribuição.

Esses casos de testes são descritos nas seções seguintes.

3.1.1 Dependência entre Objetos

A dependência entre objetos é definida como um relacionamento de utilização de objetos, ou seja, uma instância de objeto utiliza as informações e operações de outra instância de objeto. Esse relacionamento é representado no diagrama de classes através de uma linha tracejada ligando um objeto dependente ao outro objeto [74].

A dependência fica mais clara quando analisado um exemplo. O exemplo mais comum aplicado em classes é quando uma classe faz parte da assinatura de uma operação (método). Outro exemplo é quando o comportamento de uma operação necessita de outra classe para seu funcionamento.

A partir dessa definição, foram criados casos de teste para validar esse tipo de relacionamento no J4GE.

3.1.1.1 Caso de Teste 1 – Dependência simples

Nesse caso de teste, o objetivo é validar a instanciação de um objeto e, conseqüentemente, a referência remota criada. Essa dependência simples acontece no comportamento da operação que inicia a aplicação. No caso, em Java, essa operação é denominada *main()*. O diagrama de classe que apresenta essa relação de dependência pode ser visto na Figura 27.

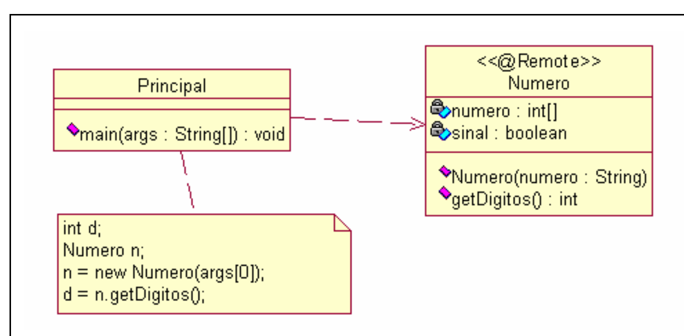


Figura 27 – Diagrama de classes com dependência simples entre objetos.

Para esse caso de teste foi criada uma classe que consegue armazenar números inteiros que ultrapassem as limitações de representação da linguagem. Esse armazenamento é feito em um vetor e as operações básicas são realizadas a partir desse vetor [98]. O uso desse objeto é apenas para verificar se a referência remota consegue contabilizar a quantidade de dígitos que um determinado número possui (*getDigitos()*).

Os resultados para esse teste, juntamente com as entradas usadas e as saídas obtidas, podem ser observados na Tabela 4.

As saídas indicadas como erradas, na verdade, são respostas do aplicativo para as situações que não satisfazem à instanciação de um objeto do tipo *Numero*.

Nesses casos, a entrada do CT1-04 apresenta um símbolo (+) no meio do número, o CT1-05 apresenta uma letra (p) no meio do número e o CT1-06 não informa um número (ou nulo).

Tabela 4 – Entradas e saídas para teste da dependência simples.

Caso de Teste	Entrada (args[0])	Saída (d)
CT1-01	987653827645138438928374	24
CT1-02	-7264957384759208	16
CT1-03	+316293478962983465	18
CT1-04	-23728437809+23949239845	Mensagem de erro
CT1-05	7219384769243p082109762	Mensagem de erro
CT1-06	(vazio)	Mensagem de erro

3.1.1.2 Caso de Teste 2 – Dependência entre duas ou mais classes

O objetivo deste caso de teste é validar a referência de um objeto remoto utilizado como parte do funcionamento de uma operação de uma outra classe. A classe responsável pelo início do programa (classe *Principal*) é considerada a terceira classe nesse relacionamento, não sendo objetivo deste caso de teste.

Dessa forma, esse caso de teste foi construído para que, dada uma instância de um *Numero*, outra classe consiga criar uma representação formatada para visualização dos dígitos.

O diagrama de classes que apresenta essa relação de dependência pode ser visto na Figura 28.

Nesse diagrama, a dependência entre as classes *Formato* e *Numero* fica clara, pois a primeira classe possui uma operação que utiliza a segunda como parte da assinatura.

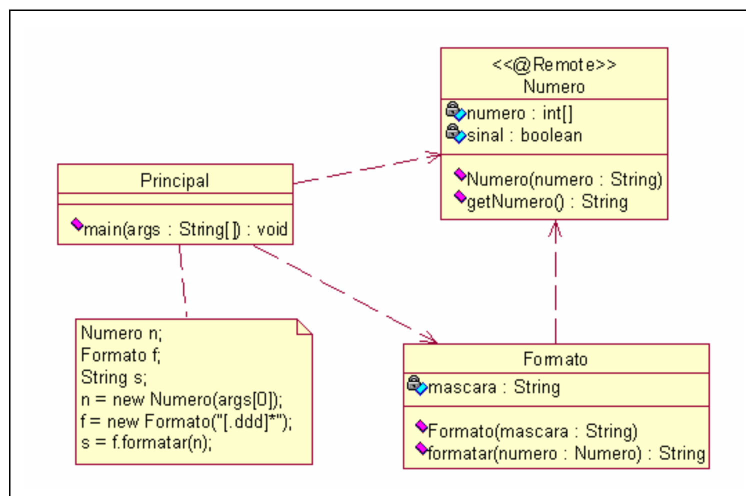


Figura 28 – Diagrama de classes com dependência entre duas classes.

Além dessa solução, foi construída uma outra que permite que cada instância de classe, tanto de *Numero* quanto de *Formato*, seja alocada em nós remotos e diferentes no *grid*. Essa outra solução pode ser vista no diagrama de classe apresentada na Figura 29.

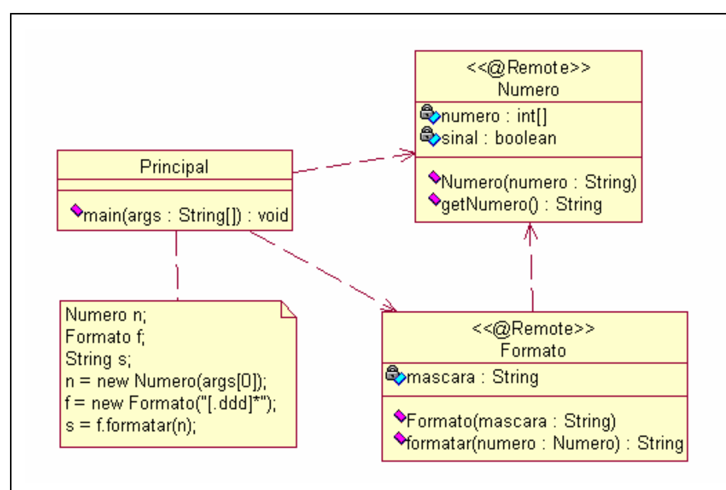


Figura 29 – Diagrama de classes com dependência entre duas classes remotas.

Nesta solução, a instância de classe *Formato* (*f*) está alocada em um nó diferente da classe *Principal* e da instância de classe *Numero* (*n*). Para formatar a saída (*s*), o objeto remoto *n* faz acesso ao objeto remoto *f*, uma vez que a dependência existe na especificação do diagrama de classes e é oferecido pelo ambiente J4GE (através da estratégia de anotação em classe).

Os resultados para esse teste, juntamente com as entradas usadas e as saídas obtidas, podem ser observados na Tabela 5. Nesse conjunto de teste não foram especificados casos em que o aplicativo tratasse erros, uma vez que isso já foi feito no Caso de Teste da seção anterior.

Tabela 5 – Entradas e saídas para teste da dependência entre duas classes.

Caso de Teste	Entrada (args[0])	Saída (s)
CT2-01	987653827645138438928374	987.653.827.645.138.438.928.374
CT2-02	-7264957384759208	-7.264.957.384.759.208
CT2-03	+986293478962983465	+986.293.478.962.983.465

3.1.2 Generalização de Objetos

A generalização de objetos é a relação entre objetos de tal forma que há nessa relação um objeto mais geral e um outro objeto mais específico. Esse relacionamento também é chamado de “é-um” ou “herança”. Nessa relação surgem os temos “classe-mãe” ou “superclasse”, para a classe mais geral, e “classe-filha” ou “subclasse”, para a classe mais especialista. Essa relação permite que uma instância de classe-filha seja utilizada em lugares que a classe-mãe apareça, mas não o contrário. Uma propriedade importante dessa relação é que as classes especialistas herdam as características e operações da classe mãe. Graficamente, esta relação é apresentada como uma linha sólida entre as classes e uma seta triangular apontando para a classe-mãe [74].

No caso específico da linguagem Java, a herança de comportamento acontece somente de uma única classe-mãe. Porém, é possível haver diversas heranças de tipos ou assinaturas, através das classes denominadas *interface* [83].

Os casos de testes que foram gerados para a generalização de objetos visam explorar a relação entre as classes envolvidas, garantindo as características e os comportamentos herdados. Esses casos de testes podem ser vistos nas seções a seguir.

3.1.2.1 Caso de Teste 3 – Generalização de Objetos Personalizados

Neste caso de teste, o objetivo é validar a generalização entre classes que existe em uma aplicação simples. Dessa forma, além de explorar a herança (generalização), a

instanciação e a referência remota também são validadas para esse tipo de relacionamento.

A Figura 30 apresenta o diagrama de classes com a relação de herança da aplicação.

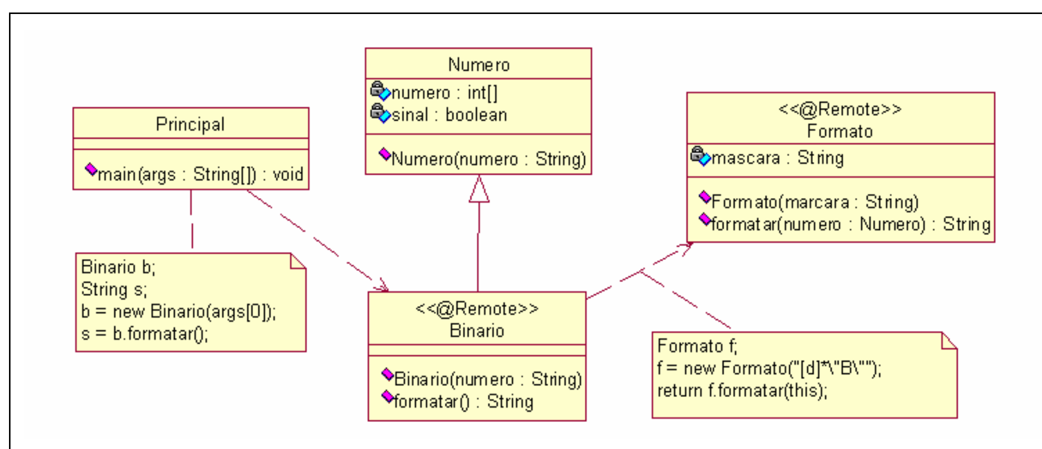


Figura 30 – Diagrama de classes com generalização de objeto personalizado.

Nesse caso, a aplicação possui suas próprias classes, ou seja, personalizou a relação entre classes para resolver seu problema. Em particular, foi criada uma classe para armazenamento de números binários (*Binário*) que é uma especialização da classe *Numero* descrita anteriormente. Além disso, essa classe especialista possui uma operação para formatação do valor binário que, neste caso, é uma relação de dependência para com a classe *Formato*.

Esse teste é importante para comprovar o funcionamento tanto da classe-mãe quanto da classe-filha, além de garantir o funcionamento da referência para objeto remoto.

A Tabela 6 apresenta os resultados dos testes realizados nessa aplicação.

Tabela 6 – Entradas e saídas para teste de herança em classes personalizadas.

Caso de Teste	Entrada (args[0])	Saída (s)
CT3-01	1001111100101011101011	1001111100101011101011B
CT3-02	1110111111010100000101011	1110111111010100000101011B

3.1.2.2 Caso de Teste 4 – Generalização de Objetos de Biblioteca e de *Threads*

Este caso de teste apresenta a relação de herança de uma classe-filha com uma classe-mãe da biblioteca Java. Nesse exemplo, a classe-mãe é responsável pela criação de *threads* na plataforma Java. A partir dela, é possível especializar a classe-filha para fornecer um comportamento de execução concorrente descrito na operação *run()*.

Este exemplo é importante, pois, a partir dessa especialização é possível criar *threads* no ambiente de *grid*, através do J4GE. A Figura 31 apresenta o digrama de classes que representa um exemplo de aplicação.

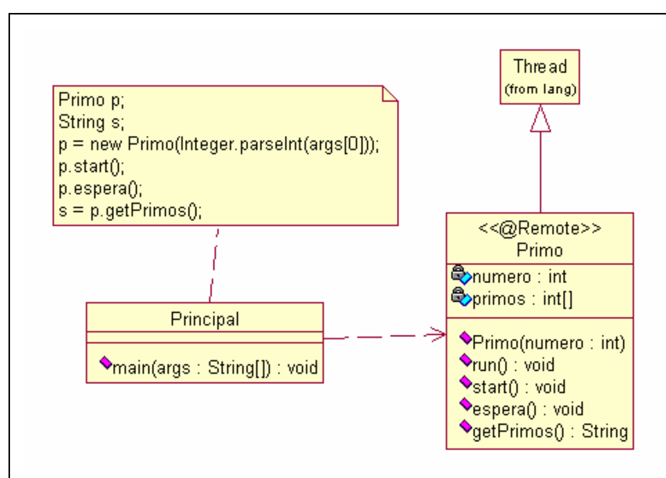


Figura 31 – Diagrama de classes com especialização da classe *Thread*.

A classe especialista *Primo* encontra todos os números primos dentro do intervalo fechado entre um e o valor informado para a aplicação. Esses valores são armazenados e podem ser recuperados através da operação *getPrimos()*.

Apesar dessa facilidade, o ambiente não suporta alguns tipos de operações definidas em classes, tanto personalizadas quanto de bibliotecas. Esses tipos de operações são aquelas definidas como *final*, ou seja, não permitem a sobrescrita (polimorfismo) de operações. Nesse caso, a estratégia adotada pelo ambiente J4GE para a criação de uma classe do padrão Proxy não pode ser realizada. No exemplo, a operação *espera()* representa o comportamento de espera até o término de execução de uma *thread*. Nesse caso, ela encapsula a chamada de uma operação já existente na biblioteca *Thread*, porém, definido como *final*: *join()*.

Apesar desse problema, foi possível realizar os testes, desde que desenvolvido corretamente as operações apresentadas no digrama de classes. Esses testes podem ser vistos na Tabela 7.

Tabela 7 – Entradas e saídas para teste de especialização da classe *Thread*.

Caso de Teste	Entrada (args[0])	Saída (s)
CT4-01	20	1, 2, 3, 5, 7, 11, 13, 17, 19
CT4-02	97	1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97

3.1.3 Associação entre Objetos

Uma associação entre objetos é uma relação estrutural que especifica que um objeto está conectado ao outro, e vice-versa. Essa associação pode ser ainda do tipo agregação e composição, possuir papeis e nome e apresentar multiplicidade da relação. Além disso, pode haver a representação de navegabilidade que, por padrão, acontece para ambos os lados. Uma associação é representada graficamente como uma linha sólida entre os objetos envolvidos na relação [74].

Apesar dos diversos tipos de associação, na construção do código-fonte de uma classe, essa relação normalmente se torna um atributo [74]. Dessa forma, os casos de testes serão gerados pensando nos atributos das classes.

3.1.3.1 Caso de Teste 5 – Associação simples

Este caso de teste tem como objetivo validar a relação de associação entre duas classes. Essa associação é denominada simples, pois estão envolvidas apenas duas classes e apenas uma delas deve ser alocada remotamente.

A Figura 32 apresenta o diagrama de classes para a aplicação exemplo.

Nesse exemplo, a instância da classe *Pessoa* é criada remotamente. Esta classe possui um atributo *telefone* devido a sua associação com a classe *Telefone*. Dessa

forma, quando houver uma mudança no valor dessa instância através da operação *setNumero()*, ela reflete no atributo da classe.

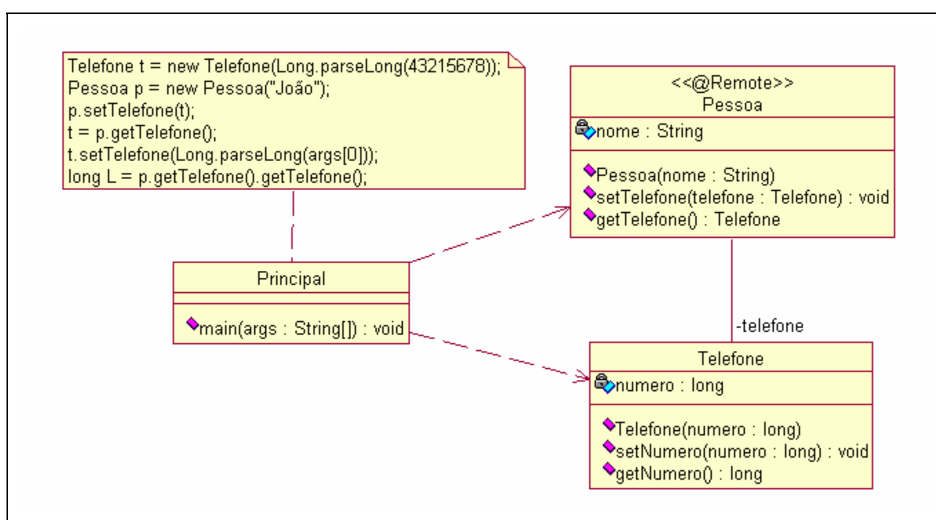


Figura 32 – Diagrama de classes para associação simples.

Na versão atual do ambiente J4GE, é possível detectar apenas que um atributo está sendo recuperado de dentro de uma classe. Sendo assim, é possível perceber que a mudança do valor deverá refletir no atributo remoto somente depois que ele foi recuperado do objeto remoto. Se uma instância ainda for local e for usada para mudar seu valor, o ambiente não tem condições de detectar essa ação de mudança, pois esta instância foi criada localmente.

No exemplo, isso pode ser percebido através da instância do objeto *t*. Ele foi criado localmente e, se fosse utilizado para alterar seu valor através da operação *setNumero()*, o atributo remoto estaria desatualizado. Porém, ao se fazer o *getTelefone()* para a variável *t*, este passa a ser a referência correta para o atributo remoto, conforme a estratégia definida no ambiente J4GE. Há uma classe *J4GEAttribute* criada dinamicamente para o tratamento dessa referência.

Dessa forma, é possível criar os casos de teste para esse exemplo de aplicação, conforme a Tabela 8.

Tabela 8 – Entradas e saídas para teste da associação simples entre classes.

Caso de Teste	Entrada (args[0])	Saída (L)
CT5-01	56784321	56784321
CT5-02	98761234	98761234

3.1.3.2 Caso de Teste 6 – Associação entre Classes Remotas

Este outro caso de teste apresenta uma aplicação com relação de associação entre duas classes remotas. Além de terem referência uma da outra, ou seja, navegabilidade, elas também utilizam operações, garantindo que as referências foram feitas de forma correta.

A Figura 33 apresenta o diagrama de classe da aplicação exemplo que foi testada.

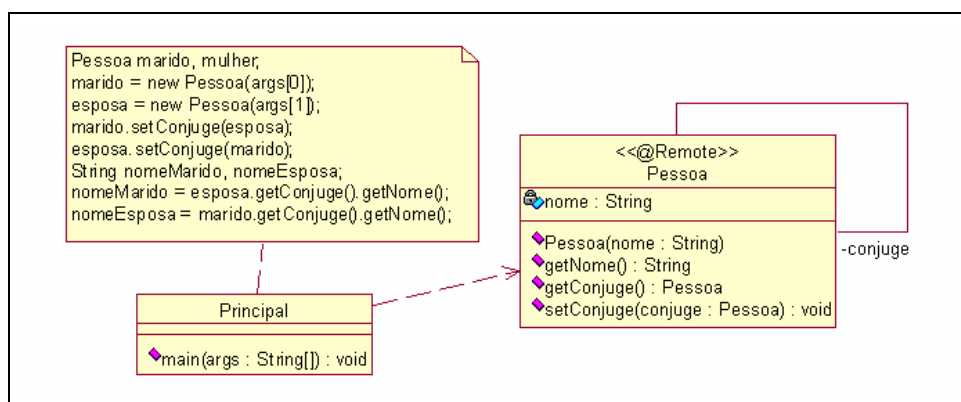


Figura 33 – Diagrama de classes para associação entre classes remotas.

Nesse exemplo, existe uma associação a um objeto da mesma classe denominado *conjuge*. Porém, devido à anotação de classe remota utilizada, essa associação será feita entre objetos remotos através do J4GE.

O resultado da execução desse teste valida a referência que cada objeto possui do outro, recuperando o nome do cônjuge através da operação *getConjuge()*. A Tabela 9 apresenta os casos de testes realizados para esse exemplo de aplicação.

Tabela 9 – Entradas e saídas para teste da relação de associação entre duas classes remotas.

Caso de Teste	**Entradas**		**Saídas**	
	args[0]	**args[1]**	**nomeMarido**	**nomeEsposa**
CT6-01	“Jose”	“Maria”	“Jose”	“Maria”
CT6-02	“Abraao”	“Sara”	“Abraao”	“Sara”

3.1.4 Outros Casos de Teste

Apesar da validação dos principais relacionamentos entre classes, ainda existem duas outras estratégias que precisam ser verificadas para o ambiente J4GE: método remoto e atributo remoto. Para estas estratégias, foram gerados casos de testes que validam seu comportamento em um ambiente de *grid*, garantindo sua integridade e referência. Esses casos de testes são apresentados nas seções seguintes.

3.1.4.1 Caso de Teste 7 – Método Remoto

Este caso de teste valida o uso de uma operação (método) anotada como remoto, ou seja, seu comportamento será executado em algum nó do *grid*, através do ambiente J4GE. Nesse caso, os testes serão criados para que a integridade dos atributos de uma instância de objeto seja garantida.

O exemplo de aplicação apresentado na Figura 34 encontra os números primos no intervalo fechado entre um e o valor informado para a aplicação. Ao término da procura, os números primos estão armazenados no vetor de *primos*. Esses valores são recuperados através da operação *getPrimos()* e a operação de procura pelos números (*run()*) é anotada como remota.

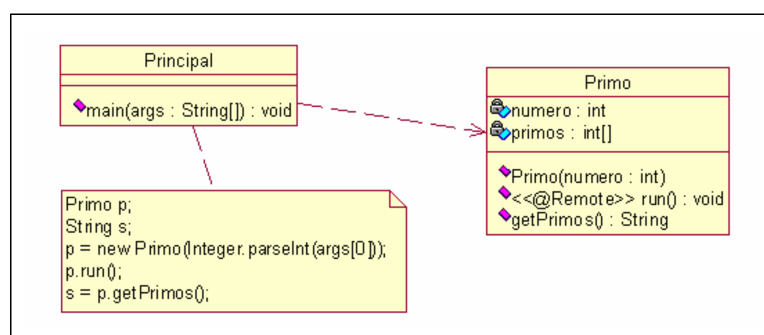


Figura 34 – Diagrama de classes para teste da estratégia de método remoto.

Como essa procura acontece remotamente, ao terminar a execução da operação *run()*, o vetor *primos* local deverá conter todos os valores encontrados. Os casos de testes apresentados na Tabela 10 exploram essa integridade.

Tabela 10 – Entradas e saídas para teste da estratégia de método remoto.

Caso de Teste	Entrada (args[0])	Saída (s)
CT7-01	1	1
CT7-02	47	1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47
CT7-03	111	1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109

3.1.4.2 Caso de Teste 8 – Atributo Remoto

O caso de teste para validação de atributo remoto considera a integridade desse atributo e, conseqüentemente, a sua referência transparente através do J4GE.

Dessa forma, os casos de teste criados exploram essa integridade, através da aplicação apresentada na Figura 35. Nesse exemplo, uma instância da classe *Pessoa* possui uma associação com um objeto *Telefone*. Este objeto, por ser um atributo remoto de *Pessoa*, será alocado em algum nó do *grid*, de acordo com a estratégia definida pelo J4GE.

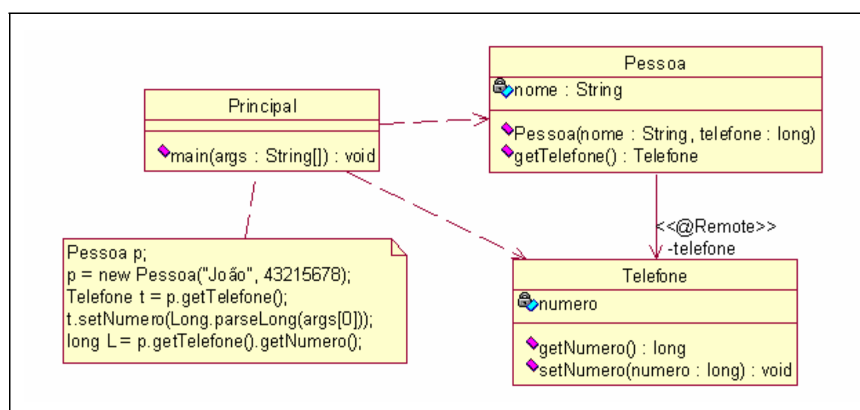


Figura 35 – Diagrama de classes para teste da estratégia de atributo remoto.

Para validar o comportamento da aplicação no *grid*, garantindo a referência e a integridade desse atributo, os testes da Tabela 11 foram realizados.

Tabela 11 – Entradas e saídas para teste da estratégia de atributo remoto.

Caso de Teste	Entrada (args[0])	Saída (L)
CT8-01	34560987	34560987
CT8-02	87651234	87651234

3.2 VALIDAÇÃO DO MODELO DISTRIBUÍDO E PARALELO

Apesar do processo de validação do modelo orientado a objetos explorar as estratégias de distribuição de objetos, métodos e atributos do ambiente J4GE, ainda é necessário apresentar casos de teste para validar, juntamente com o comportamento distribuído, a execução paralela de aplicações. Nesse novo processo de validação, o custo da troca de mensagem também é analisado, pois os objetos distribuídos passam a utilizar esse mecanismo de comunicação.

Além disso, é importante explorar os mecanismos de sincronização existentes na aplicação, verificando se estes realmente garantem a integridade dos dados compartilhados.

3.2.1 Caso de Teste 9 – Aplicação Distribuída Simples

Esse caso de teste visa comprovar a distribuição da aplicação no *grid*, utilizando os mecanismos do J4GE. Por ser uma aplicação simples, é possível medir também o custo das ações do J4GE como, por exemplo, nas chamadas de métodos do objeto remoto utilizando o Serviço de Mensagem.

A Figura 36 apresenta a organização das classes da aplicação. Nesse exemplo, a classe *Primo* calcula a quantidade de números primos existentes no intervalo fechado entre um e o valor informado pela aplicação, utilizando o crivo de Eratóstenes [98].

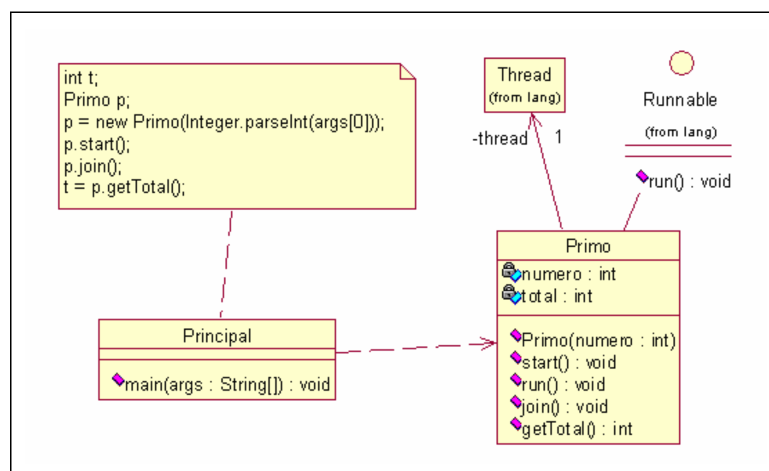


Figura 36 – Diagrama de classes especificando a criação de um objeto-*thread*.

Essa classe *Primo* foi construída para que tenha o comportamento de uma *thread*. Em Java, o relacionamento com a interface *Runnable* permite que uma instância de um objeto seja uma *thread*, criado a partir da classe *Thread* [23]. Dessa forma, a aplicação poderá ter, além do comportamento distribuído, um comportamento paralelo. Utilizando o padrão *Adapter* [95] no atributo *thread*, foi encapsulado o comportamento dos métodos *start()* e *join()*.

Como nesse exemplo existe apenas um único objeto-*thread* remoto, os resultados obtidos servem não só para confirmar a criação de *threads* remotas, mas também para mostrar o custo das mensagens trocadas entre esses objetos.

Os resultados dos testes podem ser vistos na Tabela 12.

Tabela 12 – Entradas e saídas para teste de objeto-*thread* remoto.

Caso de Teste	Entrada (args[0])	Saída (t)
CT9-01	10.000	1230
CT9-02	100.000	9593
CT9-03	1.000.000	78499
CT9-04	10.000.000	664580

Além da validação dos resultados encontrados, foi possível contabilizar o tempo de execução da aplicação no *grid* e na máquina local. A máquina local utilizada foi o nó do cluster com maior poder de processamento. Dessa forma, é possível analisar as diferenças do tempo entre execuções da mesma aplicação. Esses tempos podem ser vistos na Tabela 13.

Tabela 13 – Tempo de execução (segundos) da procura por números primos, tanto em uma máquina local quanto no J4GE.

Ambiente	**10.000**	**100.000**	**1.000.000**	**10.000.000**
Local	0,0075	0,1053	2,5906	68,4044
J4GE	5,0105	5,1814	7,3028	72,8414

Nesta tabela, é possível perceber que a diferença entre as execuções se manteve praticamente a mesma, em um valor aproximado de 4,7 segundos, para a primeira execução. Nesse tempo estão contabilizados a instrumentalização da classe *Primo*, o uso do Serviço de Mensagem para a instanciação do objeto e para a chamada de três métodos (*start()*, *join()* e *getTotal()*), como mostra a Tabela 14. Metade do tempo de execução do método *start()* e do método *join()* são contabilizados juntamente com a execução da *thread* remota, não podendo separá-los. Porém, baseado nos resultados dos outros métodos, pode-se afirmar que esses dois métodos possuem tempo similar.

A partir desses tempos, é possível perceber que a instrumentalização da classe é feita muito rapidamente. E, na segunda vez que a classe é usada, esse tempo diminui ainda mais (aproximadamente 10 vezes mais rápido, desconsiderando as instruções em código para a contagem desse tempo), pois a classe já foi carregada para a memória pela JVM.

A primeira vez que a instanciação do objeto acontece (*new*), a conexão entre os Serviços de Mensagens envolvidos precisa ser estabelecida pela primeira vez. Esse tempo de conexão inicial aumenta o tempo de execução dessa primitiva. Esse tempo diminui nas próximas vezes em que o Serviço de Mensagem é utilizado, ficando em torno de 0,45 segundos. Isso é percebido também na 2ª execução da instanciação de um objeto remoto, apresentado na Tabela 14. Acredita-se que esse aumento de desempenho também esteja relacionado a alguma técnica de *cache* do *container* do Globus, já que não existe mudança no tratamento entre instanciação ou chamada de métodos no Serviço de Mensagem.

Tabela 14 – Tempo de execução (segundos) de diversas ações do J4GE.

Ação	**Tempo 1ª execução**	**Tempo 2ª execução**
Instrumentalização – classe *Primo*	0,000008	0,000001
Instanciação – operador *new*	3,9227	0,4610
Chamada de método – start()	0,4491	0,5031
Chamada de método – *join()*	0,4491	0,5031
Chamada de método – *getTotal()*	0,3503	0,2694

3.2.2 Caso de Teste 10 – Aplicação Distribuída e Paralela

Este último teste tem como objetivo explorar uma aplicação paralela, distribuída por diversos nós do *grid*. Além disso, haverá troca de mensagens entre as diversas partes da aplicação.

Para a construção dessa aplicação foi escolhido o problema do Caixeiro Viajante (TSP). Esse problema consiste em percorrer todas as cidades de um determinado mapa no menor percurso, sem repetir qualquer cidade, e retornando à cidade de partida, formando um ciclo. Normalmente, esse problema é resolvido utilizando técnicas de grafos e, devido à sua complexidade, ele consome um tempo de processamento exponencial [99].

Nesta aplicação de teste, adaptou-se um algoritmo já existente que combina diversas cidades obtendo caminhos parciais [100]. Na verdade, os caminhos parciais são formados a partir do arranjo simples das cidades tomadas três a três.

Dessa forma, a aplicação foi especificada de acordo com o diagrama de classes apresentado na Figura 37.

Os conjuntos de dados de entrada utilizados na aplicação são fornecidos pela biblioteca TSPLIB [101]. Cada conjunto de entrada utiliza um mapeamento de diversos lugares do mundo. A identificação do formado de entrada de dados foi construída na classe *TSPLIB95*.

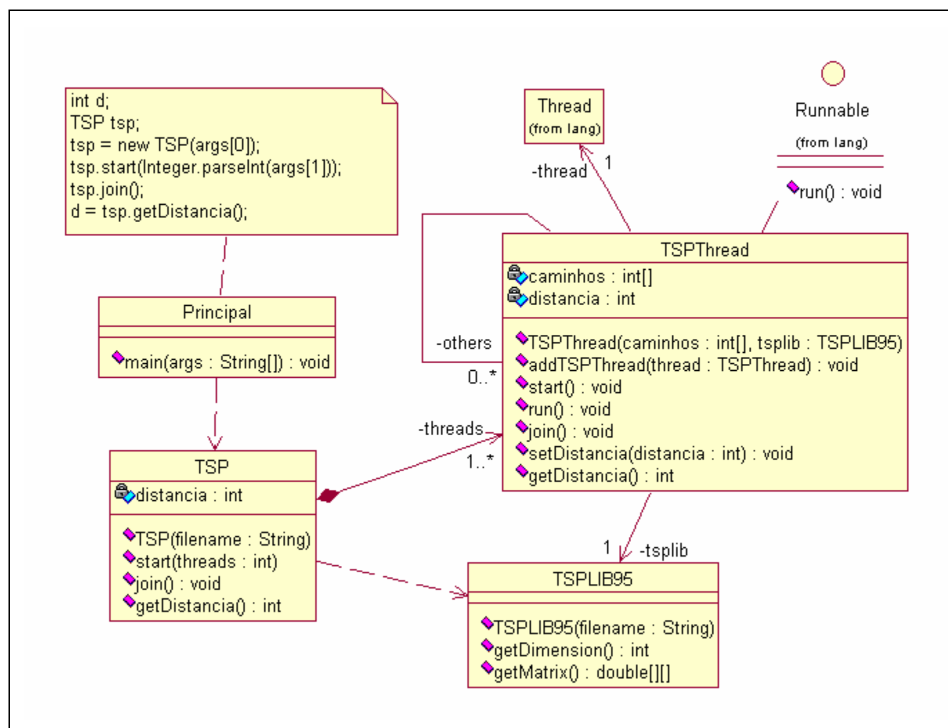


Figura 37 – Diagrama de classes com a especificação da aplicação TSP.

Após a entrada dos dados, a matriz de adjacência é construída e, a partir dela, é gerado o arranjo das cidades tomadas três a três. Esse conjunto é construído na classe *TSP*.

Em seguida, baseado na quantidade de *threads* que serão criadas, é feita a divisão desse conjunto de tal forma que cada *thread* recebe uma parte dessa divisão. Nesta aplicação, a criação da *thread* é feita através da classe *TSPThread* (padrão *Adapter*) e todas as *threads* criadas possuem referência para as demais.

Cada *thread*, para iniciar a procura de um caminho, escolhe um elemento do subconjunto recebido. Em seguida, continua a procurar as próximas possibilidades de caminho a partir do trio de cidades escolhido. Se, durante a procura, a distância do caminho parcial for maior que a distância mínima já encontrada, esse caminho é descartado. Porém, se um caminho possuir distância menor que a distância mínima anterior, a distância mínima é atualizada. Além disso, todas as outras *threads* são notificadas com essa nova distância mínima. Essa notificação é feita através do método *setDistancia()*, utilizando as referências para as outras *threads*.

Durante a atualização da distância mínima, têm-se duas seções críticas: quando a *thread* encontra um caminho e atualiza sua distância local, e quando a distância é atualizada através do método *setDistancia()*. Nesse caso, como o controle é local a

cada *thread*, o comportamento do monitor Java utilizado é similar ao mecanismo de comunicação inter-processo denominado leitores/escritores [24].

Os primeiros testes foram realizados utilizando conjuntos de dados de entrada gerados manualmente. Em seguida, utilizou-se um conjunto de dados da TSPLIB. Os tempos de execução podem ser observados na Tabela 15.

Tabela 15 – Tempo de execução (segundos) da aplicação TSP.

Arquivo TSPLIB	Seqüencial	J4GE (1+4 nós)
exemplo-12	253,0571	93,9366
exemplo-15	223,2628	115,5094
gr-17	329.435,8439	105.363,0479

Os testes seqüenciais foram realizados no nó do cluster com o maior poder de processamento. Já os testes realizados no *grid* utilizaram apenas cinco máquinas, das quais uma máquina executa a aplicação principal e as outras quatro executam as *threads*.

Durante a procura do menor caminho, as *threads* trocaram informações, diminuindo seu desempenho total. No caso do *exemplo-12*, o algoritmo explora todas as possibilidades, até o último nível do grafo, pois a menor distância é igual em qualquer caminho adotado. Dessa forma, a troca de mensagem é menor, tendo um ganho de 2,7 vezes o tempo da execução seqüencial. Vale lembrar ainda que o *grid* é formado por computadores heterogêneos, alterando também o tempo de execução final.

No caso do *exemplo-15*, as *threads* trocam mais mensagens devido ao formato do mapa da cidade. Nesse caso, o custo destas mensagens aumenta o tempo total de execução, permitindo apenas um desempenho 1,9 vezes maior que o seqüencial.

Para o exemplo real *gr-17*, a troca de mensagens acontece apenas no início da procura pelo menor caminho. O tempo total da aplicação paralela e distribuída é influenciado pela heterogeneidade dos computadores do *grid*, uma vez que o tempo de execução da versão seqüencial foi calculado na melhor máquina do *grid*, resultando em um desempenho 3,1 vezes maior. Esse caso exemplifica a dificuldade de se calcular desempenho em *grid*, uma vez que diversos tipos de equipamentos podem ser utilizados para a execução da aplicação.

Os casos de teste utilizados na aplicação podem ser vistos na Tabela 16.

Tabela 16 – Casos de testes para a aplicação TSP.

Caso de Teste	Entradas		Saída (d)
	args[0]	args[1]	
CT10-01	exemplo-12.tsp	4	12
CT10-02	exemplo-15.tsp	4	28
CT10-03	gr-17.tsp	4	2085

3.3 CONSIDERAÇÕES SOBRE OS RESULTADOS

A construção de casos de testes para o paradigma orientado a objetos valida o software para as estratégias básicas especificadas na plataforma J4GE [84]. Alguns cenários dentro da plataforma Java, como as *inner-classes* [23], não foram objetivo do processo de validação do ambiente.

Devido à restrição do ambiente para as anotações na herança, não é possível criar casos de testes para um relacionamento de especialização em que a classe-mãe é anotada com alguma das anotações do J4GE. Nesse caso, as anotações em uma especialização nunca são herdadas para as classes-filhas, pois estas devem ser analisadas, pelo programador, individualmente no contexto de *grid*.

O relacionamento de generalização também precisa de algum tratamento especial, uma vez que as assinaturas de operações modificadas pela sintaxe Java podem interferir no funcionamento do ambiente, como *native*, *final* e *static*. Apesar destas, o modificador *synchronized*, para seções críticas, mostrou comportamento satisfatório.

Na associação, não é possível que uma instância local seja alterada para uma instância remota. Dessa forma, se algum objeto não-remoto foi criado e utilizado para modificar algum atributo de classe remota, aquela instância continua sendo local. Para evitar problemas de inconsistência, após o envio do objeto para a classe remota, é necessário recuperá-lo, para assim poder ter a referência remota correta. Esta estratégia adotada pelo J4GE não sobrecarrega a engenharia dos *bytecodes* nas classes, uma vez que, para detectar esse tipo de operação, é necessário percorrer todo o código da classe. Na versão apresentada nesse trabalho, o J4GE

utiliza instrumentalização sob-demanda, ou seja, apenas modifica aqueles itens necessários para o comportamento da aplicação no *grid*.

Do ponto de vista da distribuição e do paralelismo, é possível criar aplicações utilizando os recursos que a linguagem Java oferece, como as *threads*, de tal forma que o ambiente J4GE transforma a aplicação para ser executada distribuída e paralelamente no *grid*. É fato que o custo da transmissão das mensagens interfere no tempo total da aplicação, porém, as vantagens que se tem em utilizar diversos recursos, em diversos níveis, de diferentes Domínios, para executar uma aplicação superam essa desvantagem.

Além disso, fatores do *grid*, como a heterogeneidade dos recursos, podem interferir no desempenho da aplicação.

Comparando o J4GE com os demais trabalhos na área de orientação a objetos e *grid*, pode-se construir um resumo comparativo, conforme mostra a Tabela 17.

Apesar deste trabalho não contemplar migração e tolerância a defeitos, o J4GE já possui suporte para essas características. Para a migração, o ambiente já dispõe de um mecanismo de escalonamento, sendo possível acrescentar o reenvio de um objeto para outro nó, baseado em novas políticas, atualizando as referências existentes sobre esse objeto. E, para a tolerância a defeitos, o próprio Serviço de Mensagem oferece um histórico da troca de mensagens entre objetos, possibilitando a criação de mecanismos para resgatar o último estado consistente de um objeto, restaurando parte da aplicação.

Tabela 17 – Tabela comparativa entre o J4GE e os demais trabalhos.

	J4GE	**Javelin [65]**	**JaDiMa [68]**	**ProActive [69]**
Comunicação	SOAP (*webservice*)	*socket*	MPI	RMI
Transparência no uso da API Java	sim	não (classe *JavelinClient*)	não (classes *mpiJava*)	não (classe *ProActive*)
Migração	inexistente (trabalho futuro)	inexistente	inexistente	através da classe *ProActive*
Tolerância a defeitos	inexistente (trabalho futuro)	inexistente	*checkpoint/recover*	*checkpoint/recover*
Infra-estrutura	Serviço de Mensagem	própria	SUMA/G	Globus, LSF, PBS, dentre outros
Níveis	sim	sim	não	não

4 CONCLUSÃO

Apesar dos avanços nos estudos sobre *computational grid*, a construção de aplicações para execução em *grids* ainda é dependente de paradigmas de passagem de mensagens ou de bibliotecas proprietárias, dificultando o desenvolvimento dessas aplicações.

Além disso, os recursos disponíveis no *grid* estão espalhados em níveis, organizados em Domínios administrativos, impossibilitando que parte de uma aplicação alocada em um *cluster* consiga trocar mensagens com sua outra parte alocada em outro *cluster*, em Domínios diferentes.

Apresentando uma solução para esses problemas citados, essa tese propôs um ambiente de programação orientado a objetos distribuídos e paralelos para grades computacionais, denominado J4GE.

Esse ambiente oferece mecanismos para o comportamento do modelo orientado a objetos em um *grid*, obedecendo às estratégias (i) de distribuição da instância de uma classe, (ii) da execução de um método ou (iii) da alocação de um atributo em nós diferentes do *grid*.

Com essas estratégias, o ambiente J4GE garante o modelo orientado a objetos, de acordo com suas principais características: dependência, associação, generalização. Ele também oferece outros mecanismos de distribuição, como execução remota de método e armazenamento distribuído de atributos, disponibilizando novos recursos para a organização da aplicação em um *grid*.

A forma de se usar as estratégias durante a construção de uma aplicação no J4GE também é uma característica importante. Ao contrário de outros ambientes ou trabalhos, que exigem a mudança da aplicação para um novo paradigma ou o aprendizado e entendimento de novas bibliotecas para o *grid*, o J4GE permite que a construção das aplicações seja feita com os conceitos já definidos na linguagem Java, acrescentando apenas anotações. Essas anotações não têm poder de modificar a lógica ou o algoritmo da aplicação, mas permite que o ambiente tome conhecimento sobre como a distribuição da aplicação sobre um *grid* deve acontecer.

No contexto da linguagem Java, assim como a escolha das estratégias de programação paralela de uma aplicação é de responsabilidade do programador, o

ambiente oferece mecanismos facilitados para que o programador também decida sobre como a aplicação deve ser distribuída no *grid* de forma eficiente.

Dentre essas decisões, o programador pode reutilizar as bibliotecas da plataforma Java para a criação de *threads*, juntamente com as facilidades das anotações do ambiente J4GE, permitindo que a aplicação, além de estar distribuída pelo *grid*, também seja executada em paralelo. Nesse caso, os mecanismos de sincronização e de seção crítica também são de responsabilidade do programador, e que, uma vez definidos, tem seu comportamento mantido pelo ambiente J4GE.

A camada de comunicação existente entre as partes da aplicação distribuída permite que as referências a objetos sejam feitas de forma transparente. Conseqüentemente, o acesso aos objetos remotos e suas operações são também realizadas de forma transparente, permitindo que o ambiente distribuído se comporte como um ambiente local.

Essa transparência possibilita que a aplicação seja distribuída nos diversos níveis existentes no *grid* (*cluster* de computadores e computadores, em diversos Domínios), permitindo que os recursos disponíveis sejam melhor aproveitados tanto pelo ambiente J4GE quanto para a execução de aplicações.

4.1 CONTRIBUIÇÕES

As principais contribuições deste trabalho são:

- definição de uma estratégia para a distribuição de objetos, métodos e atributos pelo *grid*, mantendo as propriedades do modelo orientado a objetos e garantindo a execução de aplicações desse paradigma no *grid*;
- utilização das estratégias de distribuição, juntamente com os recursos oferecidos pela plataforma Java, para a construção de aplicações com objetos distribuídos pelo *grid* e de execução em paralelos, concorrentemente;
- estruturação de uma camada transparente de comunicação, garantindo a troca de mensagem entre as partes da aplicação alocadas em recursos de diferentes níveis, em diferentes Domínios, ultrapassando os limites da infra-estrutura física de redes de computadores, mas garantindo as políticas administrativas e de uso da organização virtual;

- especificação de metadados, ou anotações, utilizados no código-fonte para a escolha da estratégia de distribuição de objetos, minimizando o esforço na construção e/ou manutenção de aplicações e suprimindo a necessidade de se aprender novas bibliotecas e/ou paradigmas;
- construção de um mecanismo para o aproveitamento da plataforma Java existente, inclusive dos mecanismos de sincronização e dos recursos para execução em paralelo (*threads*), com transparência no acesso à referência de objetos distribuídos, visando a especialização dessa plataforma para *grid*.

4.2 TRABALHOS FUTUROS

A partir dessa tese é possível enumerar diversas possibilidades de estudos e trabalhos futuros:

- estudar políticas, estratégias e mecanismos de escalonamento de objetos para uma melhor escolha dos nós disponíveis para a execução da aplicação no *grid*;
- definir mecanismos de coleta de lixo distribuído, para reaproveitamento de espaço em memória dos objetos que não são mais utilizados pela aplicação, durante seu ciclo de vida de execução;
- viabilizar objetos adaptadores para os objetos com modificadores especiais (*native*, *final* e *static*), no processo de instrumentalização de *bytecodes*;
- estudar a viabilidade da mudança de estratégia da reengenharia de *bytecodes* para suportar, no código-fonte, referências a objetos que ainda não foram utilizados como atributos em objetos remotos, mas o serão em instruções futuras, suportando um caso particular de referência;
- explorar ainda mais os conceitos de metadados para as classes/objetos, métodos e atributos, aumentando as informações sobre cada um desses itens, como velocidade de processamento, capacidade de memória, dentre outros, para uma melhor aplicação das estratégias de distribuição no *grid*;
- definir mecanismos de tolerância a defeitos no ambiente, propondo políticas para seu funcionamento como replicação, *checkpoint*/*recover*, dentre outros;

- permitir a migração de objetos e seus estados para outros nós existentes no *grid*, aumentando o desempenho da aplicação e garantindo o balanceamento de carga;
- estender as funcionalidades do modelo orientado a objetos existente no J4GE para suportar o modelo baseado em componentes, diminuindo ainda mais o esforço de desenvolvimento e manutenção da aplicação;
- integrar o ambiente J4GE com a infra-estrutura de grade computacional desenvolvido no LAHPC.

REFERÊNCIAS

1 Foster, I; Kesselman, C. *The Grid: Blueprint for a New Computing Infrastructure*. Elsevier, 2003.

2 Foster, I.; Kesselman, C.; Nick, J.; Tuecke, S. *The physiology of the grid: an open grid services architecture for distributed systems integration*. Disponível em http://www.globus.org/research/papers/ogsa.pdf . Acessado em 06/2004.

3 Tuecke, S.; Czajkowski, K.; Foster, I.; Frey, J.; Graham, S.; Kesselman, C. T. Maguire, T.; Sandholm, P.; Vanderbilt, D. *Grid Service Specification. Open Grid Service Infrastructure (OGSI)*. Version 1.0. Disponível em http://www.globus.org/research/papers/Final_OGSI_Specification_V1.0.pdf . Acessado em 06/2004.

4 Czajkowski, K.; Ferguson, D.; Foster, I.; Frey, J.; Graham, S.; Maguire, T.; Snelling, D.; Tuecke, S. *From Open Grid Services Infrastructure to WSResource Framework: Refactoring & Evolution*. Disponível em http://www.globus.org/wsrf/specs/ogsi_to_wsrf_1.0.pdf . Acessado em 07/2007.

5 Coulouris, G.; Dollimore, J.; Kindberg, T. *Sistemas Distribuídos – Conceitos e Projeto*. Bookman, 2007.

6 Foster, I.; Kesselman, C. *Globus: A Metacomputing Infrastructure Toolkit*. International Journal of Supercomputer Applications, 11(2):115-128, 1997.

7 Java. *Sun Java Technology*. Disponível em http://java.sun.com/ . Acessado em 12/2007.

8 Foster, I. *A Globus Primer*. Draft Paper. Disponível em http://www.globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf . Acessado em 07/2007.

9 von Laszewski, G.; Foster, I. T.; Gawor, J. *CoG kits: a bridge between commodity distributed computing and high-performance grids*. Java Grande 2000: 97-106.

10 Getov, V.; von Laszewski, G.; Philippsen, M.; Foster, I. T. *Multiparadigm communications in Java for grid computing*. Communications of the ACM 44(10):118-125 (2001).

11 De Rose, C. A. F; Navaux, P. O. A. *Fundamentos de Processamento de Alto Desempenho*. 4ª Escola Regional de Alto Desempenho, páginas 41-66. SBC, 2004.

12 Baduel, L.; Baude, F.; Caromel, D. *Object-Oriented SPMD*. Anais do IEEE International Symposium Cluster Computing and Grid (CCGrid), pág. 824-831, 2005.

13 Cardinale, Y.; Blanco, E.; Oliveira, J. *JaDiMa: Java Applications Distributed Management on Grid Platforms*. Anais do International Conference on High Performance Computing and Communications. LNCS vol. 4208, pág. 905-914, 2006.

14 Globus. *Globus Toolkit 4.0 Release Manuals*. Disponível em http://www.globus.org/toolkit/docs/4.0/ . Acessado em 12/2007.

15 gLite. *EGEE > gLite > Documentation*. Disponível em http://glite.web.cern.ch/glite/documentation/default.asp . Acessado em 02/2008.

16 JGF. *Java Grande Forum*. Disponível em http://www.javagrande.org/ . Acessado em 12/2007.

17 SBLP'04 .*Simpósio Brasileiro de Linguagens de Programação*. Disponível em http://sblp2004.ic.uff.br/ . Acessado em 12/2007.

18 SBLP'06. *Simpósio Brasileiro de Linguagens de Programação*. Disponível em http://sblp.ime.eb.br/ . Acessado em 12/2007.

19 OOPSLA'07. *International Conference on Object-Oriented Programming, Systems, Languages and Applications*. Disponível em http://www.oopsla.org/oopsla2007/ . Acessado em 12/2007.

20 JAVAPD'07. *9th International Workshop on Java and Components for Parallelism, Distribution and Concurrency*. Em conjunto com International Parallel and Distributed Processing Symposium (IPDPS 2007). Disponível em http://www.labri.fr/perso/chaumett/conferences/iwjpdc/2007/iwjpdc2007.html . Acessado em 03/2008.

21 JCP. *The Java Community Process*. Disponível em http://www.jcp.org .Acessado em 12/2007.

22 IBM. *IBM Java technology*. Disponível em http://www.ibm.com/developerworks/java . Acessado em 12/2007.

23 Deitel, Harvey M. *Java: como programar*. 6ª edição. Prentice Hall, 2007.

24 Tanenbaum, Andrew S. *Sistemas Operacionais Modernos*. 2ª edição. São Paulo: Prentice Hall, 2003.

25 GCJ. *The GNU compiler for the Java Programming Language*. Disponível em http://gcc.gnu.org/java/ . Acessado em 12/2007.

26 Java. *About the Java Technology*. Disponível em http://java.sun.com/docs/books/tutorial/getStarted/intro/definition.html . Acessado em 01/2008.

27 Hsieh, C.-H. A.; Gyllenhaal, J. C.; Hwu, W. W. *Java bytecode to native code translation: the Caffeine prototype and preliminary results*. 29th IEEE/ACM International Symposium on Microarchitecture, 1996.

28 Fitzgerald, R.; Knoblock, T. B.; Ruf, E.; Steensgaard, B.; Tarditi D. *Marmot: an optimizing compiler for Java*. Software Practice and Experience, 30(3):199-232, 2000.

29 Excelsior JET. *Java Virtual Machine (JVM) and Native Code Compiler*. Disponível em http://www.excelsior-usa.com/jet.html . Acessado em 03/2008.

30 Paleczny, M.; Vick, C.; Click, C. *The Java HotSpot server compiler*. 1st Java Virtual Machine Research and Technology Symposium (JVM'01), páginas 1-12, Monterey, USA, 2001.

31 Hotspot VM. *Java SE HotSpot at a Glance*. Disponível em http://java.sun.com/javase/technologies/hotspot/ . Acessado em 03/2008.

32 Aho, A. F.; Sethi, R.; Ullman, J. D. *Compiladores: princípios, técnicas e ferramentas*. Rio de Janeiro: LTC, 1995.

33 Choi, J.; Gupta, M.; Serrano, M. J.; Sreedhar, V. C.; Midkiff, S. P. *Escape Analysis for Java*. Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), páginas 1-19, 1999.

34 Nagarajayya, N.; Mayer, J. S. *Improving Java Application Performance and Scalability by Reducing Garbage Collection Times and Sizing Memory Using JDK 1.4.1*. Technical report, Sun Microsystems, 2002.

35 Ossia, Y.; Ben-Yitzhak, O.; Goft, I.; Kolodner, E. K.; Leikehman, V.; Owshanko, A. *A parallel, incremental and concurrent GC for servers*. ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI'02), páginas 129-140. ACM Press, 2002.

36 Buyya, R. *High Performance Cluster Computing – Volume 1: Architectures and Systems*. Prentice-Hall, 1999.

37 NUMALink. *SGI® NUMAlink™ Interconnect Fabric*. Disponível em http://www.sgi.com/products/servers/altix/numalink.html . Acessado em 03/2008.

38 Beowulf. *The Beowulf cluster site*. Disponível em http://www.beowulf.org/ . Acessado em 03/2008.

39 Zigman, J. N.; Sankaranarayana, R. *Designing a Distributed JVM on a Cluster*. 17th European Simulation Multiconference (ECM'03), páginas 363-370. Disponível em http://www.scs-europe.net/services/esm2003/ . Acessado em 03/2008.

40 cJVM. *Cluster Virtual Machine for Java*. Disponível em http://www.haifa.ibm.com/projects/systems/cjvm/ . Acessado em 03/2008.

41 Aridor, Y.; Factor, M.; Teperman, A.; Eilam, T.; Schuster, A. *Transparently obtaining scalability for Java applications on a cluster*. Journal of Parallel and Distributed Computing – Special Issue on Java on Clusters, 60(10):1159-1193, Outubro 2000.

42 Zhu, W. *Distributed Java Virtual Machine with Thread Migration*. Tese de Doutorado. Universidade de Hong Kong, 2004.

43 Zhu, W.; Wang, C.; Lau, F. C.M. *JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support*. Anais do IEEE Fourth International Conference on Cluster Computing (CLUSTER 2002), páginas 381-388, Chicago, USA, 2002.

44 Kaffe.org. *A free Java virtual machine*. Disponível em http://kaffe.org/ .Acessado em 03/2008.

45 Veldema, R.; Bhoedjang, R. A. F.; Bal, H. E. *Jackal, A Compiler Based Implementation of Java for Clusters of Workstations*. Relatório técnico. Universidade Erlangen-Nurnberg, Alemanha, 2001.

46 Veldema, R.; Hofman; R. F. H.; Bhoedjang, R. A. F.; Bal, H. E. *Runtime Optimizations for a Java DSM Implementation*. ACM-ISCOPE Conference on Java Grande, páginas 153–162, Palo Alto, CA, 2001.

47 Factor, M.; Schuster, A.; Shagin, K. *JavaSplit: A Runtime for Execution of Monolithic Java Programs on Heterogeneous Collections of Commodity Workstations.* Anais do IEEE Fifth International Conference on Cluster Computing (CLUSTER 2003), páginas 110-117, Hong Kong, Japão, 2003.

48 Factor, M.; Schuster, A.; Shagin, K. *A distributed runtime for Java: yesterday and today*. Anais do IEEE 18th Parallel and Distributed Processing Symposium – Workshop 5, páginas 159a, 2004.

49 Dahm, M. *The Byte Code Engineering Library*. Disponível em http://bcel.sourceforge.net/ . Acessado em 03/2008.

50 The Apache Jakarta Project. *BCEL: The Byte Code Engineering Library*. Disponível em http://jakarta.apache.org/bcel/ . Acessado em 03/2008.

51 Yu, W.; Cox, A. L. *Java/DSM: A platform for heterogeneous computing*. Concurrency: Practice and Experience, 9(11):1213-1224, 1997.

52 Antoniu, G.; Boug'e, L.; Hatcher, P.; MacBeth, M.; McGuigan, K.; Namyst, R. *The Hyperion system: Compiling multithreaded Java bytecode for distributed execution*. Parallel Computing, 27(10):1279-1297, 2001.

53 Philippsen, M.; Zenger, M. *JavaParty – transparent remote objects in Java*. Concurrency: Practice and Experience, 9(11):1225-1242, 1997.

54 Keleher, P.; Dwarkadas, S.; Cox, A. L.; Zwaenepoel, W. *Treadmarks: Distributed shared memory on standard workstations and operating systems*. Em Winter 1994 USENIX Conference, pág. 115–131, 1994.

55 Buyya. R. *Economic-based Distributed Resource Management and Scheduling for Grid Computing*. Tese de Doutorado. Universidade de Monash, 2002.

56 Cirne Filho, W. C.; Santos Neto, E. *Grids Computacionais: Da Computação de Alto Desempenho a Serviços sob Demanda*. Mini-curso do 23º Simpósio Brasileiro de Redes de Computadores. Porto Alegre, RS: SBC: Sociedade Brasileira de Computação, 2005, v. 23, p. 15-65.

57 Foster, I. *What is the grid? a three point checklist*. GRID Today, vol. 1, Julho de 2002.

58 Krauter, K.; Buyya, R.; Maheswaran, M. *A taxonomy and survey of grid resource management systems for distributed computing*. Software: Practice and Experience, 32(2):135-164, 2002.

59 von Laszewski, G; Foster, I. T.; Gawor, J.; Lane, P. *A Java commodity grid kit*. Concurrency and Computation: Practice and Experience 13(8-9): 645-662 (2001).

60 von Laszewski, G.; Gawor, J.; Lane, P.; Rehn, N.; Russell, M. *Features of the Java Commodity Grid Kit*. Concurrency and Computation: Practice and Experience 14(13-15): 1045-1055 (2002).

61 TIOBE Index. *TIOBE Programming Community Index*. Disponível em http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html . Acessado em 05/2008.

62 Java Task Force. *ACM Java Task Force*. Disponível em http://jtf.acm.org/ . Acessado em 05/2008.

63 JTF-CC2005. *Computing Curricula 2005*. Disponível em http://www.computer.org/portal/cms_docs_ieeecs/ieeecs/education/cc2001/CC2005-March06Final.pdf . Acessado em 05/2008.

64 Christiansen, B. O.; Cappello, P. R.; Ionescu, M. F.; Neary, M. O.; Schauser, K. E.; Wu, D: *Javelin: Internet-based Parallel Computing using Java*. Concurrency: Practice and Experience 9(11): 1139-1160 (1997).

65 Neary, M. O.; Brydon, S. P.; Kmiec P.; Rollins, S.; Cappello, P. R. *Javelin++: Scalability Issues in Global Computing*. Java Grande 1999: 171-180

66 Neary, M. O.; Phipps, A.; Richman, S.; Cappello, P. *Javelin 2.0: Java-based parallel computing on the Internet*. Anais do 6th Internacional Euro-Par Conference on Parallel Processing. LNCS vol. 1900, pág. 1231-1238, 2000. Disponível em http://repositories.cdlib.org/postprints/40 . Acessado em 07/2007.

67 Hernández, E.; Cardinale, Y.; Figueira, C.; Teruel, A. *SUMA: A Scientific Metacomputer*. Parallel Computing. Parco 99. Delft, Holanda. 1999.

68 Cardinale, Y.; Hernández, E. *Parallel Checkpointing on a Grid-enabled Java Platform*. Anais do ECG 2005 – European Grid Conference. LNCS vol 3470, pág. 741-750, 2005.

69 Baduel, L.; Baude, F.; Caromel, D. *Asynchronous Typed Object Groups for Grid Programming*. International Journal of Parallel Programming, 35(6): 573-614 (2007).

70 Foster, I. *A Globus Primer*. Draft Paper. Disponível em http://www.globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf . Acessado em 07/2007.

71 GRID Infoware. *Grid Computing Info Centre (GRID Infoware)*. Disponível em http://www.gridcomputing.com/ . Acessado em 05/2008.

72 Matsui, A. A. M. *Um ambiente de desenvolvimento e execução de aplicações grid escritas totalmente em Java*. Dissertação de Mestrado. Escola Politécnica, USP, 2006.

73 Sommerville, I. *Engenharia de Software*. 8ª ed. Pearson Addison-Wesley, 2007.

74 Booch, G.; Rumbaugh, J.; Jacobson, I. *UML: guia do usuário*. 2ª ed. Elsevier, 2005.

75 Dahm, M. *Byte Code Engineering*. Java-Informations-Tage, 1999: 267-277.

76 GT4. *GT4 Admin Guide*. Disponível em http://www.globus.org/toolkit/docs/4.0/admin/docbook/ . Acessado em 02/2008.

77 VDT. *VDT Documentation*. Disponível em http://vdt.cs.wisc.edu/documentation.html . Acessado em 06/2007.

78 PBS. *Portable Batch System*. Disponível em http://www.openpbs.org/ . Acessado em 04/2008.

79 Condor. *Condor Project*. Disponível em http://www.cs.wisc.edu/condor/ . Acessado em 03/2008.

80 Tivoli Workload Scheduler LoadLeveler. *IBM Cluster software: Tivoli Workload Scheduler LoadLeveler*. Disponível em: http://www-03.ibm.com/systems/clusters/software/loadleveler/index.html . Acessado em: 06/2008.

81 Condor-G. *Condor: High Throughput Computing*. Disponível em: http://www.cs.wisc.edu/condor/condorg/ . Acessado em 03/2008.

82 Karonis, N.; Toonen, B.; Foster, I. *MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface*. Journal of Parallel and Distributed Computing, 2003.

83 Szyperski, C.; Gruntz, D.; Murer, S. *Component software: beyond object-oriented programming*. 2º ed. Addison-Wesley, 2005.

84 Pressman, R. *Engenharia de Software*. 6ª ed. McGraw-Hill, 2008.

85 AspectJ. *The AspectJ Project*. Disponível em http://www.aspectj.org/ . Acessado em 06/2008.

86 Brown, A. W.; Wallnau, K. C. *The Current State of CBSE*. IEEE Software 15 (5): 37-46 (1998).

87 Annotations. *Annotations (The Java Tutorials > Learning the Java Language > Classes and Objects)*. Disponível em http://java.sun.com/docs/books/tutorial/java/javaOO/annotations.html . Acessado em 03/2008.

88 JVM^tm Tool Interface. *Deploying Agents*. Disponível em: http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html#deployingAgents . Acessado em 07/2008.

89 Gridway. *Metascheduling Technologies for the Grid*. Disponível em http://www.gridway.org . Acessado em 02/2008.

90 Sotomayor, B. *The Globus Toolkit 4 Programmer's Tutorial*. Disponível em http://gdp.globus.org/gt4-tutorial/ . Acessado em 11/2005.

91 Annotations. *Annotations*. Disponível em: http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html . Acessado em 11/2008.

92 Kodali, R. R.; Wetherbee, J.; Zadrozny, P. *Begining EJB 3 Application Delelopment: from novice to professional*. Apress, 2006.

93 Keith, M.; Schincarial, M. *Pro EJB 3: Java Persistence API*. Apress, 2006.

94 Lorimer, R. J. *Instrumentation: Modify Applications with Java 5 Class File Transformations*. Disponível em http://www.javalobby.org/java/forums/t19309.html . Acessado em 03/2008.

95 Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

96 Trail: The Reflection API. *Trail: The Reflection API (The Java™ Tutorials)*. Disponível em: http://java.sun.com/docs/books/tutorial/reflect/index.html . Acessado em 07/2008.

97 JavaAssist. *Java Programming Assistant*. Disponível em http://www.csg.is.titech.ac.jp/~chiba/javassist/ . Acessado em 08/2008.

98 Knuth, D. E. *The Art of Computer Programming*. vol 2. Addison-Wesley, 1998.

99 Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. *Algoritmos – Teoria e Pratica*. Campus, 2002.

100 Bal, H. E. *Programming Distributed Systems*. Prentice-Hall, 1991.

101 TSPLIB. *TSPLIB*. Disponível em: http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95 . Acessado em 10/2007.