MobiGrid: arcabouço para agentes móveis em ambiente de grades computacionais

Rodrigo Moreira Barbosa

DISSERTAÇÃO APRESENTADA
AO INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA UNIVERSIDADE DE SÃO PAULO
PARA OBTENÇÃO
DO GRAU DE MESTRE EM CIÊNCIAS

Área de Concentração : **Ciência da Computação**
Orientador : **Prof. Dr. Alfredo Goldman vel Lejbman**

São Paulo, março de 2007

Este trabalho foi financiado por FAPESP (Processo N. 02/11165-6).

MobiGrid: arcabouço para agentes móveis em ambiente de grades computacionais

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por Rodrigo Moreira Barbosa e aprovada pela Comissão Julgadora

São Paulo, março de 2007

Banca Examinadora:

Prof. Dr. Alfredo Goldman vel Lejbman (orientador) - IME/USP
Prof. Dr. Marcelo Finger - IME/USP
Prof. Dr. Francisco José da Silva e Silva - UFMA

Agradecimentos

Em primeiro lugar, gostaria de agradecer a Deus, por tudo. Aos meus pais, Djalma e Leonor, quero agradecer pela formação que me proporcionaram, não medindo esforços para me dar a melhor educação possível. Aos fabulosos médicos, Dr. Vicente Odone Filho, Dra. Telma Murias, Dra.Liliam Cristófani, Dra. Gilda Porta e Dra. Renata Pugliese, quero agradecer por serem exemplos inspiradores de dedicação, abnegação, compaixão e competência. Foram eles que me proporcionaram estar aqui hoje livre do câncer e com saúde. À Dra. Telma, quero agradecer de forma especial, pois foi ela quem acompanhou mais de perto a minha enfermidade, sempre com muito carinho e zelo. Ao meu orientador, Prof. Alfredo Goldman, meus agradecimentos pela amizade, pelo aconselhamento e também pela paciência. Ao Prof. Fabio Kon, quero agradecer pelo aconselhamento e pelas idéias. À minha namorada, Livia, que é a pessoa a quem mais escuto, quero agradecer pelo apoio, carinho, amor e compreensão. Aos meus amigos não citados aqui, por serem muitos, quero agradecer pela amizade e pelo incentivo.

Resumo

Este texto apresenta nosso projeto de implementação de um arcabouço de suporte a agentes móveis dentro de um ambiente de grade denominado InteGrade. Nosso arcabouço - MobiGrid - foi criado de forma a permitir que aplicações seqüencias longas possam ser executadas em uma rede de estações de trabalho pessoais. Os agentes móveis são utilizados para encapsular essas aplicações com longo tempo de processamento. O encapsulamento de uma aplicação com longo tempo de processamento dentro de um agente móvel é o que denominamos como *tarefa*. Sendo assim, as *tarefas* podem migrar sempre que a máquina é requisitada por seu usuário local, já que são providas com capacidade de migração automática. Nosso arcabouço também fornece ao usuário um gerente que rastreia as *tarefas* por ele submetidas. Baseados no ambiente de execução de *tarefas* descrito, criamos um modelo matemático para efetuarmos simulações de como se comportariam muitas *tarefas* submetidas a uma grade com grande quantidade de estações de trabalho. Neste trabalho apresentamos também esse modelo, bem como os resultados das simulações nele baseadas.

Abstract

This text presents a project which focuses on the implementation of a framework for mobile agents support within a grid environment project, namely InteGrade. Our framework - MobiGrid - was created in such a way that time consuming sequential applications can be executed on a network of personal workstations. The encapsulation of a long processing application by a mobile agent is what we call *task*. Hence, the *tasks* can migrate whenever the local machine is requested by its local user, since they are provided with automatic migration capabilities. Our framework also provides the user with a manager that keeps track of the submitted agents. Based on the execution environment described above, we have created a mathematical model which allows us to simulate how a great quantity of *tasks* submitted to a grid with many workstations would behave. In this text, we also present our model, as well as the results of our simulations.

Sumário

Agradecimentos **iii**

Resumo **v**

Abstract **vii**

1 Motivação **1**

1.1 Aglomerados e o Beowulf 1

1.2 Grades Computacionais 2

1.3 Agentes Móveis 3

1.4 Organização do Texto 3

2 Projeto InteGrade e Objetivos **5**

2.1 InteGrade 5

2.1.1 Diferenciais do InteGrade em Relação a Outros Projetos de Grades 6

2.1.2 Linhas de Pesquisa 7

2.2 Objetivos 9

2.2.1 Presença Transparente 10

2.2.2 Utilização Otimizada dos Recursos 10

2.2.3 Agentes Móveis no InteGrade 10

3 Agentes Móveis e Seus Sistemas **11**

3.1 Agentes Móveis 11

3.1.1 MASIF 11

3.1.2 Definições 13

3.1.3 Tipos de Migração 14

3.2 Sistemas de Agentes Móveis 18

3.2.1 Grasshopper 19

3.2.2 JADE 21
3.2.3 Aglets 23
3.3 Utilização do Aglets 30

4 MobiGrid 31
4.1 Arquitetura do Arcabouço 31
4.1.1 Principais Desafios 31
4.1.2 Arquitetura 33
4.2 Estrutura de Classes do Arcabouço 36
4.2.1 Descrição das Classes e Diagrama de Classes UML . . . 36
4.2.2 Exemplo de Código 38
4.3 Casos de Uso 40
4.3.1 Submissão de *tarefa* 40
4.3.2 Volta do Usuário Local 41
4.3.3 Monitoramente do Gerente 41
4.4 Questões Mais Específicas 42

5 Estado de Execução e Vivacidade 43
5.1 Preservação do Estado de Execução 43
5.1.1 Solução Proposta 43
5.1.2 Limitações 44
5.1.3 Um exemplo 44
5.1.4 Experimentos para Medição de Sobrecarga 46
5.2 Controle de Vivacidade 48
5.2.1 Migração 49
5.2.2 Clonagem 50
5.2.3 Políticas para Falhas 51
5.3 Funcionamento do MobiGrid 52

6 Modelo Matemático e Experimentos 53
6.1 Modelo Matemático 53
6.1.1 Experimentos com Clonagem e Migração 53
6.1.2 Experimentos com Migração e Clonagem 54
6.1.3 Experimentos com Clonagem e Execução 54
6.1.4 Hipóteses de Simplificação 55
6.1.5 Modelo Matemático 56
6.1.6 Implementação da Simulação 59

6.2 Experimentos com o Simulador 60
6.2.1 Estudos dos Parâmetros de Tarefas 62
6.2.2 Estudos dos Parâmetros de Ambiente 68
6.3 Resumo dos Resultados da Simulação 74

7 Trabalhos Futuros e Conclusão 75
7.1 Agentes Móveis *versus* Objetos Distribuídos 75
7.2 Agentes Móveis em Ambiente de Grade 76
7.3 MobiGrid e MAG 77
7.4 Situação Atual e Trabalhos Futuros 78
7.5 Resumo das Contribuições 78
7.6 Conclusão 79

Lista de Figuras

3.1 Taxonomia para Migração Forte proposta por Illmann et al. . . . 17
3.2 Arquitetura do JADE 22
3.3 Visão geral da API do Aglets 25
3.4 Tela do Tahiti 28
3.5 Exemplo de Seriação 28

4.1 Arquitetura Geral do Arcabouço 35
4.2 Diagrama de Classes do Arcabouço 38

5.1 Gráfico dos experimento da Tabela 5.1 47
5.2 Protocolo de Migração 49
5.3 Protocolo de Clonagem 51

6.1 Diagrama de Classes UML da Implementação do Simulador . . . 60
6.2 Variação de l e o tempo de execução 63
6.3 Variação de l e o número de *grupos de tarefas* que terminaram . 63
6.4 Variação de TM e o tempo de execução 65
6.5 Variação de TM e o número de *grupos de tarefas* que terminaram 65
6.6 Variação de TE e o tempo de execução 66
6.7 Variação de TE e o número de *grupos de tarefas* que terminaram 67
6.8 Variação de p_1 e o tempo de execução 69
6.9 Variação de p_1 e o número de *grupos de tarefas* que terminaram 69
6.10 Variação de p_2 e o tempo de execução 71
6.11 Variação de p_2 e o número de *grupos de tarefas* que terminaram 71
6.12 Variação de m e o tempo de execução 73
6.13 Variação de m e o número de *grupos de tarefas* que terminaram 73

Lista de Tabelas

5.1 Experimento para o `AgletTask` do *Insertion Sort* 46
5.2 Experimento para `AgletTask` do *makespan* 47

6.1 Experimentos com migração e clonagem (tempo em ms) 54
6.2 Experimentos com o algoritmo de ordenação (tempo em ms) . . 55
6.3 Valores Default para os Parâmetros 61
6.4 Experimentos com a Variação de l 62
6.5 Experimentos com a Variação de TM 64
6.6 Experimentos com a Variação de TE 66
6.7 Experimentos com a Variação de p_1 68
6.8 Experimentos com a Variação de p_2 70
6.9 Experimentos com a Variação de m 72

Capítulo 1

Motivação

No passado, a computação de alto desempenho era realizada somente em super-computadores. Esses computadores eram basicamente computadores paralelos, compostos de muitos processadores com memória compartilhada ou distribuída, interconectados por um barramento de alta velocidade. Todavia, esse tipo de computador tem um preço elevadíssimo ainda hoje, devido à sua forma de fabricação, que deve superar limitações físicas importantes. Quando não estão sendo utilizados, há um gigantesco desperdício de recursos, visto que uma quantidade considerável de tempo de computação está sendo perdida.

1.1 Aglomerados e o Beowulf

Enfrentando esses problemas de preço e de desperdício, os pesquisadores procuraram um novo paradigma que possibilitasse a construção de computadores acessíveis de alto desempenho: os *aglomerados*. Um *aglomerado* não é nada mais que um conjunto de computadores simples - normalmente PCs - interconectados por uma rede de alta velocidade.

Talvez o exemplo mais conhecido e bem-sucedido de aglomerado tenha sido o **Beowulf** [BTBCS04]. O **Beowulf** é um conjunto de ferramentas que permitem que vários computadores ligados em uma rede de alta velocidade formem um *aglomerado*. Os **Beowulfs** rodam sistemas operacionais de *software livre*, usualmente **Linux** ou **FreeBSD**.

Não existe um programa chamado **Beowulf**. De fato, o **Beowulf** é formado por diversos programas que permitem que os vários computadores funcionem como um *aglomerado*. Alguns exemplos desses programas são [SSBS99]:

- MPICH [PMI04]: Biblioteca para computação paralela baseada em MPI

[MPIs04]. MPI é sigla para *Message Passing Interface*, ou seja, Interface para Troca de Mensagens, e constitui-se como um padrão para a realização de computação paralela baseada em troca de mensagens.

- LAM/MPI [PC04]: Outra biblioteca para computação paralela baseada em MPI.
- PVM [PVM04]: PVM é sigla para *Parallel Virtual Machine* - Máquina Virtual Paralela. PVM representa uma biblioteca para computação paralela.
- *Kernel* do Linux: Modificado com algumas otimizações tais como permitir que várias placas de rede possam formar uma placa de rede virtual, a qual possui uma largura de banda igual à soma das larguras das placas físicas que a compõem, ou ainda permitir que todas as máquinas do *aglomerado* sejam vistas pelo usuário como uma única máquina paralela.

Apesar de o problema de preço ter sido enfrentado por essa solução, o problema de desperdício persiste: quando um *aglomerado* não está sendo utilizado, muitos recursos estão ainda sendo desperdiçados.

1.2 Grades Computacionais

A idéia de *grade computacional* foi claramente inspirada pelos *aglomerados*, no sentido que temos muito computadores interconectados por uma rede, a fim de proverem maior poder computacional. No entanto, uma *grade* não depende de redes de alta velocidade e o *hardware* necessário à sua criação é mais acessível e disponível; ela pode ser composta por computadores espalhados pelo mundo, interconectados pela Internet, por exemplo. A idéia básica é fornecer poder computacional de maneira análoga à maneira que obtemos energia elétrica [FK99]. Quando se deseja energia, simplesmente conecta-se algum dispositivo à rede elétrica (*power grid* em inglês); quando se quer recursos computacionais, conecta-se um dispositivo à *grade computacional*.

No entanto, o software necessário para a criação de uma *grade computacional* é muito mais complexo do que o software de um *aglomerado*. Isso porque ele enfrenta uma série de situações inexistentes no *aglomerado*, tais como largura de banda limitada e constante entrada e saída de nós. Além disso, é evidente que o custo pelo relaxamento por uma rede dedicada de alta velocidade deve

ser pago. Uma *grade* obteria desempenho inferior a um *aglomerado* no caso de aplicações paralelas que exijam muita comunicação. Para fazer frente a esse problema, algumas táticas podem ser adotadas, como utilizar algoritmos paralelos BSP/CGM [CMS04] [BA04], que tentam minimizar as rodadas de comunicação, ou ainda tentar alocar nós pertencentes a uma mesma subrede para executar uma aplicação paralela.

Mesmo não tendo o mesmo desempenho de um *aglomerado*, uma visão otimista sobre as *grades* argumenta que qualquer ganho obtido ao se executar uma aplicação paralela nesse ambiente, em comparação a executá-lo em apenas uma máquina dedicada, já é uma vantagem, visto que estamos ganhando desempenho às custas de um tempo ocioso que seria perdido de qualquer forma sem o ambiente de *grade*.

Sendo assim, uma grande vantagem da *grade* é que seu conceito enfrenta diretamente o problema do desperdício, tendo em vista que sempre que um computador está ocioso, seu poder computacional pode ser disponibilizado para a *grade*. Além disso, outra grande vantagem da *grade* é seu custo reduzido, que pode chegar a zero, visto que podemos constituí-la somente com computadores já disponíveis conectados a uma rede, utilizando para isso, ferramentas de software livre.

1.3 Agentes Móveis

Nesse contexto, a idéia de agentes móveis é interessante. Eles podem ser usados para encapsular aplicações oportunistas, as quais podem usar até o pouco tempo computacional eventualmente disponível em estações de trabalho, migrando para outra máquina sempre que o usuário local solicita sua máquina, preservando sempre o processamento já realizado. Dessa maneira, agentes móveis podem ser considerados como uma ferramenta complementar para diminuir ainda mais o tempo ocioso de uma *grade*.

1.4 Organização do Texto

Este texto propõe nossa solução para fornecer um ambiente de agentes móveis para *grades* e é organizado da seguinte maneira:

- O Capítulo 1 explica o que é uma *grade computacional* bem como quais são as motivações em fornecer um ambiente de agentes móveis para uma

grade, especificamente o InteGrade [Int07].

- O Capítulo 2 apresenta o projeto do InteGrade, mostrando seus diferenciais em relação a outros projetos de *grades* e localiza nossa linha de pesquisa dentro desse projeto, explicando nossos objetivos.

- O Capítulo 3 introduz alguns conceitos importantes acerca de agentes móveis e faz algumas considerações sobre os desafios da migração; também apresenta alguns sistemas de agentes móveis e explica nossa escolha por **Aglets** [Agl04].

- O Capítulo 4 introduz nosso arcabouço para utilização de agentes móveis em ambiente de grade e explica as soluções escolhidas para alguns problemas. Nesse capítulo, apresentamos também a estrutura de classes de nosso arcabouço, bem como alguns detalhes específicos com um pequeno exemplo de código.

- O Capítulo 5 mostra alguns testes que fizemos para medição da sobrecarga de preservação do estado de execução e mostra uma solução que encontramos para auxiliar o programador nessa tarefa. A seguir, explicamos melhor como funcionam nossas estratégias de controle de *vivacidade*.

- O Capítulo 6 apresenta o modelo matemático que criamos para simular uma *grade* com grande quantidade de nós, bem como as *tarefas* a ela submetidas. Além disso, mostra os resultados de alguns experimentos realizados com a simulação.

- O Capítulo 7 mostra o potencial de nosso arcabouço dentro do InteGrade e provê idéias para trabalhos futuros.

Capítulo 2

Projeto InteGrade e Objetivos

2.1 InteGrade

O projeto InteGrade [GKG+04], em desenvolvimento no IME-USP, visa a construir uma infra-estrutura genérica de *middleware* que permita a utilização do tempo ocioso das máquinas já disponíveis em instituições públicas e privadas para a resolução de diversos problemas paralelizáveis, incluindo aplicações fortemente acopladas.

A importância do InteGrade é evidenciada quando analisamos as necessidades de um país carente de recursos, como o Brasil. Enquanto pesquisadores necessitam de recursos computacionais robustos e caros, na própria instituição já podem existir dezenas ou centenas de máquinas subutilizadas. O tempo ocioso de tais máquinas poderia ser usado pelos pesquisadores, resultando em economia de recursos e eliminação de desperdício.

O InteGrade está sendo construído utilizando modernas tecnologias de sistemas de objetos distribuídos, padrões da indústria e protocolos de computação distribuída de alto desempenho.

Os principais requisitos que são considerados no desenvolvimento de InteGrade são os seguintes:

- O sistema deve auto-conhecer-se: Isso implica a necessidade de manutenção de uma base de dados atualizada dinamicamente que contenha informações sobre as máquinas que fazem parte do sistema, plataformas de hardware e software de cada máquina, tipo de rede local que interliga as máquinas de um agrupamento, tipo de rede de média e longa distância que interliga os diversos agrupamentos, além do estado dinâmico da

grade, isto é, quantidades disponíveis de recursos como disco, processador, memória e largura de banda.

- Sobrecarga quase imperceptível pelos clientes: o *middleware* deverá ser capaz de aproveitar os recursos ociosos disponíveis nas máquinas dos clientes com o menor impacto possível no desempenho percebido pelos usuários das máquinas clientes. Ou seja, sempre que o usuário utilizar a sua máquina, ele deverá ter prioridade máxima no uso dos recursos e a sobrecarga imposta pelo *middleware* deverá ser imperceptível. Caso contrário, será mais difícil obter o consentimento dos usuários para integrar a Grade.

- Garantias de segurança: já que será possível carregar dinamicamente código executável nas máquinas do cliente, é importante garantir que tal código não prejudicará o correto funcionamento de outras aplicações sendo executadas na máquina do cliente e que esse novo código não irá modificar ou mesmo ter acesso a informações pessoais, possivelmente confidenciais, armazenadas nas máquinas dos clientes.

2.1.1 Diferenciais do InteGrade em Relação a Outros Projetos de Grades

- Reutilização da base instalada: é um princípio chave do projeto InteGrade. Tal característica também pode ser observada em outros projetos, porém de maneira mais discreta em **Globus** [Glo04] [FK97] e **Legion** [Leg04] [GWF+94], e mais acentuada em **Condor** [Con04] [LLM88]. O diferencial é que InteGrade considera essa característica no desenvolvimento de sua arquitetura. Além disso, há uma tendência à utilização de recursos dedicados, sobretudo para computação paralela. InteGrade não exigirá recursos dedicados, mas poderá utilizá-los caso essa seja a vontade dos proprietários de recursos.

- Utilização de tecnologias modernas de objetos distribuídos: Característica única do InteGrade, que utiliza **CORBA** [OMG04] na sua construção. **Condor** e **Globus** são escritos em **C**, ao passo que **Legion** utiliza **MPL** [Leg04], variante de **C++**, mas não **CORBA** na sua construção.

  Com isso, obtemos duas vantagens:

1. Podemos reutilizar os diversos serviços já disponíveis na arquitetura CORBA;
2. É mais fácil a integração de outros serviços e aplicações à grade dado que esta utilizará uma tecnologia padrão de interconexão orientada a objetos.

- Baixa sobrecarga: Como já descrito, o *middleware* do InteGrade vai priorizar o proprietário do recurso. Sendo assim, é necessário que o sistema seja leve de modo a não ocupar recursos que deveriam estar direcionados ao usuário. Um exemplo de situação indesejada seria uma máquina lerda para o usuário, por conta desta estar rodando o sistema.

2.1.2 Linhas de Pesquisa

A equipe de pesquisadores deste projeto foi composta de forma a incluir especialistas em todas as áreas essenciais descritas acima. Em particular, a equipe possui uma larga experiência nas áreas de algoritmos paralelos e paralelização de código, middleware **CORBA** e **Java**, agentes móveis, gerenciamento de recursos em sistemas distribuídos e qualidade de serviço em comunicação na Internet. A seguir, apresentamos as seis principais linhas de pesquisa que compõem o projeto [Int07]:

- Requisitos e arquitetura geral do sistema: O objetivo dessa linha de pesquisa é fazer um levantamento dos requisitos do sistema, definir uma arquitetura de referência inicial e especificá-la através de diagramas UML.

- Paralelização de problemas computacionalmente difíceis: Esta linha de pesquisa objetiva criar um arcabouço e interfaces genéricas que serão utilizados para implementar a maior gama possível de algoritmos paralelos. Atualmente, já existe implementado no InteGrade suporte à comunicação através de memória compartilhada no BSP [GQKG04]. BSP [Wor04] é sigla para *Bulk Synchronous Parallel* e representa um modelo de computação paralela de granulosidade grossa, baseado em rodadas de sincronização. Existem diversas implementações de bibliotecas para o padrão BSP; a mais famosa é a Oxford BSP Toolset [OP04].

- Segurança: Questões como controle de acesso, criação de papéis e usuários da Grade, autenticação dos usuários e do código móvel, criptografia de dados confidenciais que transitam por ela são aqui tratadas.

- Identificação de padrões de acesso de usuários e de disponibilidade de recursos: Para a identificação de padrões de acesso, deve-se desenvolver um processo de *aprendizado constante* [Mit97, LM95]. Tal processo tem início com o monitoramento constante dos seguintes recursos básicos da máquina:

  - quantidade de memória disponível;
  - quantidade da área de *swap* disponível;
  - espaço em disco disponível;
  - percentagem da CPU ociosa.

  De posse desses dados, aplicam-se algoritmos de *clustering* [JW83], que visam a derivar categorias que agrupam padrões semelhantes. Essas categorias são atualizadas de acordo com um processo de *aprendizado constante*, de modo a garantir que o modelo não se defase. Com esse processo, pode-se tentar prever por quanto tempo uma máquina manter-se-á ociosa [Bez06].

- Gerenciamento de recursos inter-aglomerados: O objetivo desta linha é desenvolver protocolos e algoritmos que permitam ao InteGrade organizar seus nós em uma coleção estruturada de aglomerados, cada um composto de nós individuais. Através desta organização, permitimos que aplicações fortemente acopladas possam ser alocadas no mesmo aglomomerado, ou em aglomerados interconectados por uma boa largura de banda, com o intuito de minimizar a sobrecarga de comunicação.

- Ambiente de desenvolvimento para aplicações de grade: Atualmente, boa parte do desenvolvimento de aplicações para o InteGrade é feito de uma maneira manual e muito cansativa para o programador. O objetivo desta linha de pesquisa é facilitar esse desenvolvimento através de uma ferramenta baseada no Eclipse [Ecl06] que unifica, em único ambiente, tarefas como projeto, codificação, testes, submissão e monitoramento das aplicações.

- Acesso a computadores localizados por detrás de NATs e *firewalls*: Dentro dessa linha, enfrentam-se os problemas relativos a *firewalls* e NAT. Para os *firewalls*, o problema resume-se em como permitir que a comunicação entre componentes localizados em redes diferentes possa trafegar

por *firewalls* sem a necessidade de uma pré-configuração dos mesmos e com uma eficiência razoável. Com relação a NAT, o problema é mais sutil. NAT é sigla para *Network Address Translation* e constitui-se como uma técnica que permite que computadores de uma subrede com IPs falsos possam acessar a Internet através do mascaramento de seus endereços por um *gateway*. Logo, as máquinas que estão por detrás de NAT não possuem um IP real e, portanto, não podem ser acessadas diretamente. Sendo assim, um objetivo dessa linha de pesquisa é fornecer uma maneira pouco intrusiva que permita que essas máquinas possam ser disponibilizadas ao InteGrade.

- Mobilidade de Código e Computação Ubíqua: É dentro desta linha de pesquisa que nosso projeto de mestrado está engajado. Ela objetiva estudar questões decorrentes da necessidade de código móvel no sistema InteGrade. Uma motivação adicional seria usar o nosso *middleware* para computação ubíqua. Usuários móveis poderiam utilizar seus PDAs e computadores móveis para interagir com os dispositivos computacionais fixos no espaço físico local de forma a utilizar os recursos lá disponíveis para executar as partes pesadas do seu código. O PDA poderia exportar código móvel para um PC local e usar a CPU mais potente do PC e o seu espaço em disco para executar as componentes mais pesadas.

  Dentro desta linha, um outro grupo de pesquisadores da Universidade Federal do Maranhão também se engajou. O resultado de sua pesquisa culminou com a criação do **MAG** [Lop06]: *Mobile Agents Technology for Grid Computing Environments* - Tecnologia de Agentes Móveis para Ambientes de Grade Computacional.

2.2 Objetivos

Este projeto de mestrado constitui-se na implementação de uma infra-estrutura de suporte a agentes móveis dentro do InteGrade. Agentes móveis podem servir a vários aspectos do InteGrade, da utilização maior dos recursos computacionais, a um suporte para aplicações trivialmente paralelizáveis, ou seja, aplicações cujas cargas de trabalho podem ser trivialmente distribuídas em diversos processadores, com muito pouca ou nenhuma necessidade de comunicação entre eles.

A capacidade de migração de agentes móveis vem atender a dois requisitos primordiais do InteGrade:

1. A presença do sistema deve ser transparente aos usuários das máquinas, isto é, os usuários devem ter prioridade com relação às aplicações do InteGrade;
2. Os recursos ociosos devem ser utilizados da melhor forma possível.

2.2.1 Presença Transparente

No caso da necessidade de liberação dos recursos da máquina para o usuário local, por exemplo, se o usuário acessa uma máquina que está rodando código móvel, esse código pode migrar para outra máquina sem perder o processamento já realizado. Nesse processo, a arquitetura do InteGrade fornece os dados sobre o estado da rede e das máquinas do InteGrade, permitindo que agente móvel escolha uma máquina com recursos computacionais mais adequados e disponíveis. Nessa decisão, padrões de utilização de recursos das outras máquinas podem ser utilizados.

2.2.2 Utilização Otimizada dos Recursos

Dada a característica oportunista dos agentes móveis, sua utilização fica muito indicada para melhor utilização de recursos ociosos, já que esses agentes podem utilizar até períodos curtos de ociosidade, caminhando no sentido da ociosidade próxima de zero almejada pelo InteGrade.

A possibilidade de migração para uma máquina com recursos computacionais mais poderosos, caso alguma máquina com recursos melhores tenha ficado ociosa, também é muito interessante, exceção feita à possibilidade de o padrão de comportamento dela indicar que sua ociosidade não durará muito tempo.

2.2.3 Agentes Móveis no InteGrade

Neste contexto, os agentes móveis seriam utilizados de forma complementar às aplicações do InteGrade, permitindo um melhor aproveitamento dos recursos computacionais. Entre as aplicações que poderiam ser executadas usando agente móveis estão aplicações paralelas similares ao SETI@home [SET04] e às aplicações executadas com o BOINC [BOI04], ou, ainda, aplicações seqüenciais sem restrições quanto ao término de sua execução.

Capítulo 3

Agentes Móveis e Seus Sistemas

3.1 Agentes Móveis

Para discorrermos sobre a nossa infra-estrutura, será interessante apresentarmos definições formais de alguns conceitos da área de agentes móveis. Basearemos nossas definições na especificação **MASIF** [OMG98] [MBB+98] da *OMG* [OMG05]. Sendo assim, é interessante que expliquemos melhor o que vem a ser essa especificação.

Além disso, neste capítulo, pretendemos também abordar uma questão que tem um papel muito relevante em nosso trabalho, que são os tipos de migração possíveis para um agente móvel.

3.1.1 MASIF

MASIF é a sigla para *The OMG Mobile Agent System Interoperability Facility* - Recurso para Interoperabilidade entre Sistemas de Agentes Móveis - e descreve um padrão para a tecnologia de agentes móveis que foi adotado pela **OMG**. O objetivo desse padrão é oferecer interoperabilidade entre diferentes sistemas de agentes móveis. Ele é descrito da forma mais genérica e mais simples possível, de forma a não impedir o avanço das tecnologia de agentes móveis. **MASIF** define duas interfaces básicas:

1. **MAFAgentSystem**: interface para gerenciamento e transferência de agentes móveis;

2. **MAFFinder**: interface para serviço de nomes e localização.

Inicialmente, **MASIF** aborda apenas as questões mais simples para a interoperabilidade. O padrão não almeja a interoperabilidade entre diferentes linguagens, mesmo porque os principais sistemas de agentes móveis atuais são implementados em **Java** [Mic04], o que leva a crer que essa seja a linguagem padrão dessa tecnologia. A interoperabilidade almejada por **MASIF** é baseada no conceito de perfil (*profile*). O agente carrega consigo esse perfil, que define as características e recursos que um sistema de agentes deve possuir para receber o agente. Sendo assim, o sistema de agentes móveis deve possuir todas as características descritas no perfil do agente, de modo a poder recebê-lo. Fica evidente, então, que um desenvolvedor de sistemas de agentes que almeje uma grande interoperabilidade entre seu sistema com outros sistemas deve garantir que ele atenda a maior quantidade possível de perfis.

MASIF preocupa-se somente com a interação entre sistemas de agentes. O padrão não define as operações no nível dos agentes, tais como interpretação, seriação/desseriação, etc. Essa abordagem baseada em interfaces deixa claro a prioridade pela interoperabilidade. Atualmente, a especificação cobre apenas o requisito primário para interoperabilidade, que é o transporte de informações do agente, o que inclui seu perfil. A maneira pela qual o sistema de agentes atende esses requisitos é de responsabilidade de quem o implementa.

Os tipos de interoperabilidade definidos em **MASIF** são:

- gerenciamento de agentes: permite que um sistema de agentes possa controlar agentes de outros sistemas. Para isso, operações básicas para migração, suspensão e terminação devem ser fornecidas através de uma interface bem definida. Esse tipo de interoperabilidade é relativamente fácil de ser implementado nos sistemas de agentes;

- rastreamento de agentes: fornece recursos para o rastreamento dos agentes entre os diversos sistemas. Para esse rastreamento é necessária a utilização de recursos do **MAFFinder**. Esse tipo de interoperabilidade também pode ser implementado facilmente nos sistemas de agentes;

- comunicação entre agentes: a especificação **MASIF** não trata desse tipo de interoperabilidade, delegando-a para o serviço de comunicação entre objetos da especificação **CORBA** [OMG04];

- transporte de agentes: esse tipo de interoperabilidade trata de como transportar agentes de um sistemas de agentes para outro. Questões

como transferência de dados, descritores de classes são aqui tratadas. É o tipo de interoperabilidade mais difícil de ser conseguida e exige uma interação entre os diversos implementadores para alguma padronização dos sistemas.

3.1.2 Definições

Embora **MASIF** não seja uma especificação implementada completamente pela maioria dos sistemas de agentes, suas definições são amplamente aceitas e muito adequadas ao escopo deste trabalho. Definiremos aqui alguns conceitos fortemente baseados na especificação **MASIF**.

- *agente*: agente é um programa de computador que age de forma autônoma representando uma pessoa ou organização;

- *agente estacionário*: é um agente que executa somente no ponto onde começou sua execução, não realizando jamais uma migração;

- *agente móvel*: é um agente que pode suspender sua execução, migrar para um destino através de uma rede e lá retomar sua execução, carregando consigo seu estado de execução. Essa habilidade permite que o agente móvel migre em direção aos recursos de que necessita. A diferença primordial entre um sistema de agentes móveis e um sistema de objetos distribuídos é que, no sistema de agentes móveis, um agente pode migrar para o local onde se encontra o recurso de que ele necessita e lá fazer uso do mesmo, ao passo que, em um sistema de objetos distribuídos, se um programa necessita de um recurso remoto, esse programa interage com um objeto local a esse recurso. No paradigma de objetos distribuídos, existe a possibilidade de mobilidade de código, através do carregamento dinâmico de código remoto. Todavia, essa mobilidade tem a limitação importante de não carregar consigo o estado de execução;

- *sistema de agentes*: um sistema de agentes é uma plataforma que pode criar, interpretar, executar, transferir e terminar agentes;

- *estado do agente*: o estado do agente pode ser representado pelo pelo estado de execução do agente, ou por valores de atributos que permitam a retomada da execução sem perda de processamento já realizado;

- *estado de execução*: representa o valor atual do contador do programa (*program counter*), bem como das pilhas de execução do programa;

- *seriação/desseriação*: seriação é o processo de extração serial do estado do agente. Esses dados extraídos podem ser armazenados em memória, persistidos em algum meio, ou transportados via rede para outra localização. O processo de desseriação consiste no inverso, ou seja, na reconstrução do agente através da reconstrução do seu estado, permitindo a retomada de sua execução. Na reconstrução dos agente, o sistema deve saber as classes dos atributos.

- *base de código* (*codebase*): a base de código representa as localizações dos dados descritores das classes utilizadas por um agente. No processo de reconstrução de um agente, após sua migração, o sistema precisa saber onde buscar os dados descritores de classe dos atributos.

3.1.3 Tipos de Migração

Um dos objetivos do InteGrade é oferecer transparência em termos de desempenho para o usuário local da máquina. Por isso mesmo, a utilização de agentes móveis é interessante, visto que os agentes podem migrar liberando os recursos para o usuário local. Além disso, os agentes móveis podem fazer um uso oportunístico de recursos que não estarão ociosos por muito tempo, migrando quando os mesmos são requisitados localmente. No entanto, nesse processo de migração, os agentes móveis não podem perder o processamento realizado. Nesse contexto, é necessário que a infra-estrutura de agentes móveis forneça algum mecanismo de migração que permita a retomada de execução aproveitando o processamento já realizado. Illmann et al. [IWKK00] definiram uma taxonomia de requisitos para a migração em **Java**, a qual apresentaremos a seguir.

Migração Forte e Migração Fraca

Migração forte é definida como a migração completa de um agente, ou seja, o agente migra carregando consigo seu código e seu estado de execução. Nesses termos, a migração forte implica total transparência para o desenvolvedor do agente. O agente pode ser interrompido em qualquer ponto de sua execução, sofrer uma migração, e retomar sua execução, de forma totalmente transpa-

rente. A definição de migração fraca é feita em termos do complementar da migração forte. Em outras palavras, *migração fraca* é toda migração que não é forte.

Conceitos Importantes

Para entendermos melhor os desafios para se oferecer migração forte, é imperativo o entendimento de alguns conceitos básicos.

Para que a migração forte seja alcançada, é necessário que os dados relativos a código, objetos e execução, bem como referências a objetos e recursos (remotos ou locais) e as linhas de execução (*threads*) sejam restaurados corretamente no destino do agente móvel. Sendo assim, é conveniente definirmos esses conceitos:

- *dados de código*: representam o código do programa em si, sem o qual não seria possível uma retomada de execução;
- *dados de objetos*: os dados dos objetos são constituídos não só pelas variáveis de instância e de classes, mas também pelas variáveis locais dos métodos do objeto que se encontram na pilha de chamadas;
- *dados de execução*: os dados de execução são constituídos por:
  - pilha de chamadas: armazena a ordem dos métodos executados em cascata pela linha de execução;
  - contador do programa: armazena a posição em que a linha de execução está no método do topo da pilha de chamadas.
- *referência a outros objetos*: todo o grafo de referências diretas e indiretas do agente para objetos locais e objetos remotos deve ser reconstruído em seu destino. Por exemplo, tome um objeto A que referencia diretamente um objeto B, o qual referencia um objeto C: $A \rightarrow B \rightarrow C$. A preservação das referências diretas e indiretas faz com que, no caso da migração de A, que as referências de A para B e de B para C sejam preservadas, bem como outras referências que C venha a possuir e assim sucessivamente;
- *referência a recursos*: referências para recursos, tais como, bancos de dados, arquivos, etc. também devem ser preservados, quando do momento da migração.

Java é uma linguagem interpretada e segura, o que significa que, em tempo de execução, o código é interpretado sob restrições de segurança. Sendo assim, a linha de execução só consegue ter acesso limitado a informações nativas e internas, tais como o contador do programa, itens da pilha de execução e dados de outras linhas de execução do programa.

Taxonomia para Migração Forte

- *migração de código*: consiste na migração do código do agente, bem como da migração de código de objetos referenciados pelo agente. Para fins de otimização, código de classes disponíveis localmente no destino do agente, tais como classes disponibilizadas pela API de **Java**, não necessita ser transferido;

- *migração de estado*: é divida em dois aspectos:

  1. *migração de execução*: a migração de execução depende de dois aspectos:

     (a) *migração do contador do programa*: que marca o ponto atual de execução do programa;

     (b) *migração de linhas de execução (threads)*: que engloba o estado de cada linha de execução (suspensa, executando, bloqueada), além dos mecanismos de sincronização das mesmas (semáforos, variáveis de condição, etc.).

  2. *migração de dados*: a migração de dados depende de três aspectos:

     (a) *migração de membros*: consiste na reconstituição, no destino, dos objetos referenciados direta ou indiretamente pelo agente e pelas linhas de execução. Sendo assim, este tipo de migração está diretamente relacionado à migração de código e migração de linhas de execução. Vale notar que a reconstituição é feita sobre o estado do objeto imediatamente antes do início da migração;

     (b) *migração de pilha*: migração da pilha de chamadas de métodos da linha de execução atual. Cada item da pinha, armazena os valores das variáveis locais ao método. Esse tipo de migração tem uma ligação muito estreita com a migração do contador do programa;

(c) *migração de recursos*: consiste na migração de recursos utilizados pelo agente, tais como referências a objetos remotos, bancos de dados abertos, arquivos locais, sockets abertos, etc.

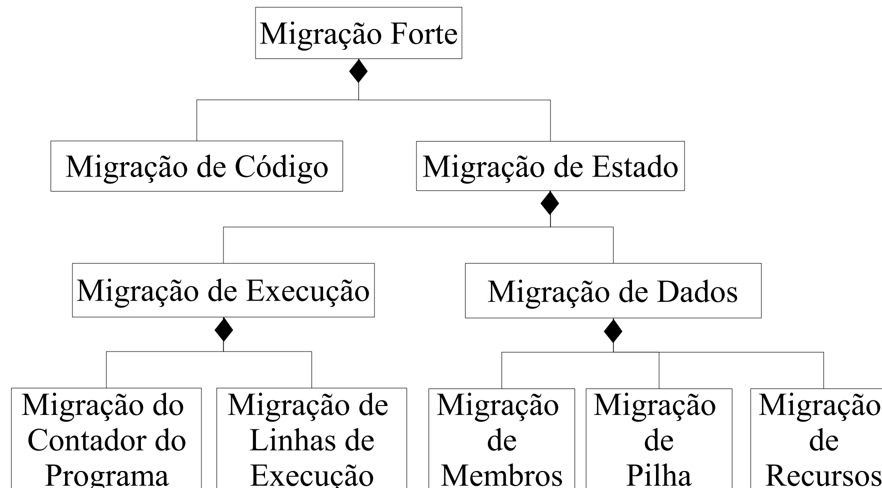

Figura 3.1: Taxonomia para Migração Forte proposta por Illmann et al.

Obtendo Migração Forte

Pelo diagrama descrito anteriormente, é claro perceber que um ambiente que forneça o mecanismo de migração forte deve oferecer migração de código e migração de estado. Por sua vez, a migração de estado só é obtida se existirem mecanismos para migração de execução e migração de dados. A migração de execução exige migração de contador do programa e migração de linhas de execução. A migração de dados exige migração de membros, migração de pilha e migração de recursos. De fato, ainda não se tem conhecimento de um ambiente que atenda todos esses requisitos para a migração forte.

Dentre os tipos de migração descritos anteriormente, dois são facilmente obtidos através da API (*Advanced Program Interface*) de **Java** [Mic04]: a migração de código e a migração de membros.

A migração de código é obtida através do mecanismo de carregamento dinâmico de classes. Com esse mecanismos, é possível a definição, em tempo de execução, de novas classes, através de seus *bytecodes*. A implementação de um carregador dinâmico de classes via rede já serviria como um serviço de migração de código.

A migração de membros também é algo relativamente fácil de ser obtido através dos mecanismos de seriação/desseriação proporcionados por **Java**.

Através desse mecanismos é possível se obter um representação seriada das variáveis de instância de uma classe, sejam elas tipos primitivos, ou referências para outros objetos. O processo de seriação é recursivo e opera sobre o grafo de referência entre os objetos, criando uma representação completa do mesmo. A implementação de um recurso que transmita essa representação por rede para outro nó que a receba e faça a partir dela a reconstituição do objeto seriado pode ser considerada como um serviço de migração de membros.

O obtenção dos outros tipos de migração já é um problema muito mais complexo que exige abordagens mais engenhosas, tais como o pré-processamento de código [IWKK00], instrumentação de *bytecode* [Lop06] ou modificação da JVM (*Java Virtual Machine*) [Bou01].

O pré-processamento de código trata-se da inserção automática, através de um pré-compilador, de código que efetua o armazenamento das informações necessárias à retomada de execução em algum objeto auxiliar. Esse objeto terá seu conteúdo preservado em uma eventual migração, devido aos serviços já citados de carregamento dinâmico de classes e seriação/desseriação de objetos.

A instrumentação de *bytecode* age em um nível mais baixo, modificando o *bytecode* gerado pela compilação do programa. Essa abordagem tem a vantagem de ser mais limpa, pois não mexe no código fonte do programador, além de ser mais eficiente, pois, agindo no nível do *bytecode*, consegue efetuar algumas otimizações, além de poder fazer uso de recursos de mais baixo nível da JVM.

A terceira opção - modificação da JVM - trata de inserir na JVM as funcionalidades necessárias à migração forte, oferecendo-as via API, contornando, assim, as restrições de segurança já citadas. Essa opção é a mais eficiente, pois evita o armazenamento constante de dados relativos ao estado atual do programa. Todavia, possui uma desvantagem muito séria, pois basearia o sistema todo em uma versão não padrão da JVM, a qual, com a constante evolução da especificação da JVM determinada pela Sun [Mic05], acabaria por tornar-se bastante obsoleta.

3.2 Sistemas de Agentes Móveis

Em um relatório pioneiro da IBM [CHK95], Chess et al. fazem uma análise das potencialidades dos agentes móveis e lançam, entre outras idéias, a possibilidade de se utilizar agentes móveis para aproveitar recursos computacionais ociosos. Isso seria feito mediante a utilização de agentes que encapsulariam pro-

cessos que migrariam por uma rede sempre à procura dos recursos de que eles necessitam. Nesse processo, levariam sempre consigo seus estados de execução, criando um novo paradigma que leva o programa em direção aos dados e não os dados em direção aos programas, como costuma acontecer. Ainda nesse relatório, é salientada a conveniência de que o suporte a agentes móveis seja feito sobre uma linguagem interpretada, o que enfrenta o problema da preservação do estado da aplicação, bem como a heterogeneidade de plataformas. Outro relatório "e-Science Gap Analysis"[FW03], o qual faz um estudo profundo da utilização de grades para o modo de se fazer ciência na atualidade, ou e-Science, aponta, entre outras falhas, a ausência de suporte a código móvel nas infra-estruturas de grades existentes.

Esses artigos nos encaminharam para a idéia da criação de um arcabouço na linguagem de programação **Java** [Mic04] de suporte a agentes móveis. A escolha por **Java** foi feita por dois motivos:

1. O principal deles é o fato de Java ser uma linguagem robusta e moderna, além de popular, o que facilitaria sua utilização por programadores.

2. O segundo motivo veio do artigo "Using Mobile Agents in Real World: A Survey and Evaluation of Agent Platforms" [AGK+01], o qual relata os resultados de uma comparação realizada entre vários ambientes de agentes móveis. Nesse relatório, onde notas mais baixas representam melhor desempenho, os ambientes **Grasshopper** [Gmb01] e **Aglets** [Agl04], ambos em Java, estão entre os melhores colocados, com notas 9,25 e 10,15 respectivamente.

Na época da realização dessa comparação, **Jade** [BCPR05] - outro sistema que se vem tornando muito popular na área de agentes móveis - estava ainda em estágio muito inicial de seu desenvolvimento, obtendo uma nota muito alta - 15,75 - com relação ao **Aglets** e ao **Grasshopper**. No entanto, durante o desenvolvimento de nosso trabalho, observamos uma grande evolução dessa plataforma, a qual vem sendo cada vez mais utilizada por pesquisadores da área. De fato, atualmente, o projeto **Jade** parece ser o mais ativo de todos os projetos de desenvolvimento de ambientes para agentes móveis.

3.2.1 Grasshopper

Grasshopper é uma plataforma de agentes móveis para Java, que segue o padrão **OMG MASIF** [OMG98]. Foi desenvolvida pela IKV++ Technologies

[AG04]. O tipo de migração oferecido por **Grasshopper** é a migração fraca.

Os principais conceitos do Ambiente Distribuído de Agentes do **Grasshopper** (DAE) serão aqui explicitados [Gmb01]:

- agentes: para **Grasshopper**, agente é um programa de computador autônomo com relação a pessoas ou organizações. Existem dois tipos de agentes:
    1. agentes móveis: são agentes capazes de migrar de uma rede física para outra. Para **Gasshopper**, sua característica essencial é a migração, ou seja, a capacidade de um agente mudar sua localização durante sua execução, retomando sua execução do ponto em que parou antes de sua mudança de localização;
    2. agentes estacionários: não podem migrar de uma rede física para outra. Estão associados com uma localização específica.
- estado dos agentes: os agentes podem estar em três estados distintos:
    1. ativo: um agente está ativo enquanto executa sua tarefa;
    2. suspenso: um agente está suspenso quando a execução de sua tarefa está temporariamente suspensa;
    3. descarregado: um agente não mais ativo cujos dados relevantes foram descarregados em disco. A partir dessa informação descarregada, ele pode ser reativado.
- agência: é o ambiente de execução para agentes móveis e estacionários. É dividida em duas partes:
    1. núcleo da agência: É responsável por prover a funcionalidade mínima que uma agência deve possuir para dar suporte à execução de agentes. Oferece os seguintes serviços:
        - serviço de comunicação;
        - serviço de registro;
        - serviço de gerenciamento;
        - serviço de transporte;
        - serviço de segurança;
        - serviço de persistência.

2. lugar: fornece uma maneira de agrupar logicamente funcionalidades dentro de uma agência.

- região: as agências, bem como os lugares que elas contêm, podem ser associadas com uma região específica, sendo registradas dentro desta última. Esse registro cuida automaticamente de todo agente que é hospedado por uma agência associada com a região.

Instalando e utilizando o **Grasshopper**, constatamos sua excelente documentação e padronização. Todavia, apesar dessas vantagens, seu fabricante restringe a utilização de seu produto dentro de outros projetos sem o pagamento de direitos. Além disso, proíbe publicação de métricas de desempenho de seus produtos. Esses fatores foram cruciais e impediram a utilização desse ambiente em nosso projeto.

3.2.2 JADE

JADE [BCPR05] é sigla para *Java Agent Development Framework*. É um sistema para agentes móveis baseado em **Java** e desenvolvido pela **TILAB** [TILHPI05]. **JADE** é regido pela licença de código aberto LGPL [LGPL05] e permite o desenvolvido de aplicações multi-agentes através do paradigma de comunicação *peer-to-peer*. Isso equivale a dizer que os agentes de **JADE** podem descobrir dinamicamente referências para outros agentes de modo a poder comunicar-se com eles. Os agentes podem estar hospedados em nós de uma rede com fio, ou em nós de rede sem fio.

Os principais requisitos de **JADE** são:

- interoperabilidade: **JADE** respeita as especificações **FIPA** [TFfIPA05], as quais são uma outra iniciativa para a padronização de sistemas para agentes móveis. Sendo assim, **JADE** pode interoperar com todos os sistemas que respeitem esse padrão;

- uniformidade e portabilidade: **JADE** fornece o mesmo conjunto de APIs independentemente da versão de **Java** e tipo de rede. Em outras palavras, **JADE** fornece as mesmas APIs para **J2SE**, **J2EE** e **J2ME**, o que permite que uma mesma aplicação seja portada para essas diferentes arquiteturas;

- transparência: a complexidade do sistema de agentes **JADE** é totalmente transparente para o programador, o que significa dizer que o programador não precisa ter conhecimento algum das especificidades de implementação de **JADE**.

O modelo de arquitetura de **JADE** é divido em duas partes principais:

1. bibliotecas de desenvolvimento: representam a API das classes disponibilizadas ao programador para a criação de um sistema multi-agentes;

2. ambiente de execução: fornece os serviços necessários para que um nó possa executar os agentes.

Cada ambiente de execução que compõe o sistema é denominado contêiner. O conjunto de todos os contêineres é denominado plataforma. Sendo assim, a plataforma fornece uma camada de abstração homogênea que encobre as especificidades das camadas inferiores: hardware, sistema operacional, versão da JVM, etc. A Figura 3.2 esquematiza a arquitetura do **JADE**.

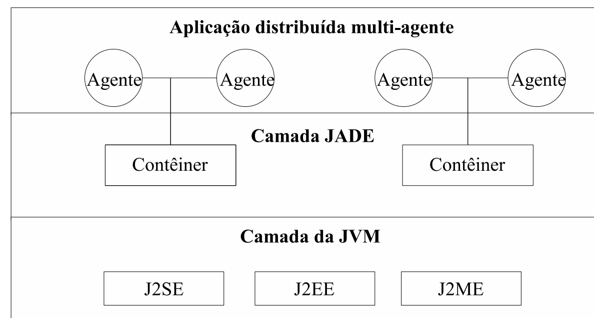

Figura 3.2: Arquitetura do JADE

Os ambientes de execução são disponibilizados para as diferentes versões de **Java**, tais como **J2ME**, **J2SE**, **J2EE**. Todavia, os mecanismos de migração são oferecidos somente para **J2ME** e **J2SE**. O tipo de migração oferecido por **JADE** é a migração fraca.

Conforme já citado, na época da comparação reportada em [FW03], **Jade** estava ainda em estágio muito inicial de seu desenvolvimento, obtendo uma nota muito ruim - 15,75 - com relação ao **Aglets** e ao **Grasshopper** - 9,25 e 10,15 respectivamente.[1] Além disso, havia em nosso Instituto uma maior experiência com a utilização de **Aglets**. Esses fatores aliados à licença excessivamente restritiva do **Grasshopper** encaminharam-nos para a utilização do **Aglets** em nosso arcabouço.

[1] Como já citado, as notas mais baixas representam um desempenho melhor.

3.2.3 Aglets

Aglets é um ambiente **Java** para o desenvolvimento e a implantação de agentes móveis. O **Aglets Software Development Kit**,**ASDK**, foi prototipado e criado pela **IBM** [IBM04] no seu Laboratório de Pesquisas de Tóquio. O projeto foi coordenado por Mitsuro Oshima e Danny Lange. A **IBM** coordenou a maior parte das versões 1.x. Com o tempo, o projeto foi transformado em uma iniciativa de código aberto, regida pela licença da IBM para código aberto, que permite a utilização e modificação do código, além de liberar trabalhos e comparações sobre o produto. O projeto passou a ser hospedado no SourceForge [Sou05] e, inicialmente, as primeiras versões lançadas pela comunidade visavam apenas a corrigir *bugs* do produto. Com o lançamento das versões 2.x, questões de gerenciamento de segurança foram melhoradas, além da introdução de mecanismos de registro (*log*) baseados no **Log4J** [Pro05]. Depois do lançamento de algumas versões 2.x, o desenvolvimento parou. A partir de outubro de 2004, as atividades da comunidade foram reiniciadas comandadas por Luca Ferrari, um então aluno de PhD da Universidade de Modena e Reggio Emilia [Hom07], Itália.

A documentação do produto, embora não muito completa, é relativamente organizada, e o ambiente oferece todos os recursos básicos para a criação do nosso arcabouço, além de recursos mais avançados. Recursos para a criação, migração, clonagem, hospedagem e visualização de agentes móveis são oferecidos, além de recursos avançados de segurança, sincronização e troca de mensagens. O sistema segue parcialmente a especificação **MASIF** [OMG98]. **Aglets**, tal como **Glasshopper** e **Jade**, oferece somente o mecanismo de migração fraca, pois garante somente a migração de código e a migração de membros, sendo que a migração de recursos é oferecida apenas parcialmente (referências a certos recursos como *sockets* e arquivos locais não podem ser migradas).

A utilização de **Aglets** para computação em grade não é nova. Aversa et al [AMMoV01] já utilizaram esse ambiente em um estudo de caso, implementando uma versão distribuída e com balanceamento de carga dinâmico do algoritmo para a resolução do *problema da mochila (0 - 1)*, o qual é um problema combinatório.

Estrutura de Classes do Aglets

Aglets oferece uma estrutura de classes cujos principais elementos são citados a seguir [OK97] [Fer04]:

- `Aglet`: é a classe que representa o agente móvel. Fornece os métodos para migração, clonagem, suspensão de execução, descarte, entre outros. O agente móvel é denominado *aglet*. Todos os agentes móveis devem estender esta classe.

  Os principais métodos disponibilizados são:

  - `dispose()`: descarta o *aglet*;
  - `dispatch(URL url)`: despacha o *aglet* para a URL especificada;
  - `getAgletID()`: obtém o `AgletID` desse *aglet*.

- `AgletProxy`: representa o procurador do *aglet*. Serve como um intermediário entre o *aglet* e o objeto que o referencia. Todas as chamadas ao *aglet* são intermediadas por ele, que faz consultas ao gerenciador de segurança, verificando se essas chamadas são autorizadas ou não. Além disso, o procurador torna transparente o processo de manipulação de um *aglet* remoto;

- `AgletID`: identificador global único do *aglet*;

- `AgletContext`: representa o hospedeiro dos *aglets*. Podemos ter vários *aglets* dentro de um contexto e vários contextos dentro de um servidor;

- `Message`: os *aglets* comunicam-se através da troca de objetos da classe `Message`. Esse objeto pode ter seu tipo e argumentos especificados por uma `String`. A maneira mais simples de enviar uma mensagem síncrona para um *aglet* é através de `AgletProxy.sendMessage(Message)`. Existem métodos para envio assíncrono de mensagens também.

Como Estender a Classe Aglet

A classe `Aglet` é a principal classe do **Aglets**. Para criar um agente móvel, o programador deve estendê-la redefinindo os seguintes métodos:

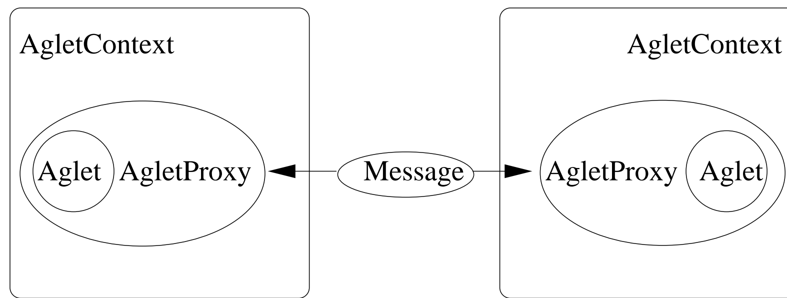

Figura 3.3: Visão geral da API do Aglets

- `void onCreation(Object init)`: devido ao fato de um objeto da classe `Aglet` ser carregado dinamicamente, a inicialização dele é feita através desse método. Esse método é chamado somente uma vez em todo ciclo de vida do `Aglet`;
- `void onDisposing()`: método chamado quando o *aglet* recebe uma requisição de descarte, ou seja, quando ele recebe a requisição de término imediato de sua execução;
- `void run()`: aqui é o ponto de entrada da linha de execução do algoritmo encapsulado pelo *aglet*. É onde o usuário deve definir seu programa. Esse método é chamado toda vez que o *aglet* chega em um `AgletContext`, ou logo após a execução de `onCreation()`, no momento da criação do *aglet*;
- `boolean handleMessage(Message m)`: toda mensagem enviada a um *aglet* é tratada por esse método. Se a mensagem for de um tipo esperado, deve devolver `true`; caso contrário, deve devolver `false`.

Um Exemplo

Mostramos agora um pequeno exemplo de implementação de um *aglet* do tipo *Hello World*.

```
public class HelloWorldAglet extends Aglet {
  public void onCreation_(Object init) {
    System.out.println (‘‘Fui criado agora!’’);
  }
  public void run () {
    System.out.println (‘‘Olá, Mundo!!!’’);
```

```
  }
  public boolean handleMessage (Message m) {
    if (m.sameKind(‘‘digaOla’’)) {
      System.out.println (‘‘Olá, Mundo!!!’’);
      return true;
    }
    return false;
  }
}
```

Neste pequeno exemplo de código, o método `onCreation()` é executado quando da criação do *aglet*; nesse momento, será exibido na saída padrão o texto "Fui criado agora!". Logo em seguida, o método `run()` será executado exibindo a mensagem "Olá, Mundo!!!". Esse mesmo método será executado sempre que esse *aglet* migrar para outro destino, fazendo com que a mensagem seja exibida. O método `handleMessage()` trata mensagens enviadas ao *aglet*. Se uma mensagem do tipo "digaOla" for enviada ao *aglet*, será exibido o texto "Olá, Mundo!!!" na saída padrão; se uma mensagem de outro tipo for enviada, o método devolve `false`, levantando uma `NotHandledException` para quem enviou a mensagem.

Ciclo de Vida de um Aglet

Em **Aglets**, todo objeto móvel é uma instância da classe `Aglet`, ou uma instância de uma subclasse dessa classe. A instanciação de um objeto `Aglet` é feita dinamicamente e pode ocorrer por duas maneiras:

1. através do método `AgletContext.createAglet(URL url, String n, Object init)`, onde `url` é a base do código do *aglet*, `n` é o nome dessa classe e `init` é uma referência para um objeto. Esse método faz com que o ambiente crie um *aglet* no contexto especificado, invocando logo a seguir o método `onCreation(Object init)` no *aglet* criado, onde a referência `init` é a mesma referência `init` passada para o `createAglet()`;

2. através do método `Aglet.clone()`, o qual cria um novo *aglet* com o mesmo estado do *aglet* já existente, mas com um `AgletID` diferente.

Depois de criado, um *aglet* pode sofrer três ações:

1. ser despachado para um servidor remoto;

2. ser desativado e persistido em disco, sendo reativado depois;

3. ser descartado.

Um *aglet* pode solicitar sua própria migração através do método `dispatch(URL url)`, onde `url` é o endereço de destino do *aglet*, formado pelo endereço do nó de destino (tipicamente um endereço IP), porta do servidor e nome do contexto dentro do servidor, visto que um servidor pode conter diversos contextos. Um exemplo seria: *atp://192.168.1.1:8834/contexto*.

O método `Aglet.deactivate(long timeout)` faz com que o *aglet* seja desativado e persistido em disco por `timeout` milissegundos. Após esse tempo, o *aglet* é reativado no mesmo contexto onde foi desativado.

O método `Aglet.dispose()` descarta o *aglet*, eliminando-o da máquina virtual. Diferentemente dos objetos **Java** normais que são recolhidos automaticamente pelo coletor de lixo (*garbage collector*) quando não mais referenciados, a eliminação de um *aglet* deve ser realizada explicitamente através deste método.

Todos esses métodos também são disponibilizados na classe `AgletProxy`, o que permite que os *aglets* sejam manipulados remotamente.

Tahiti

O **Aglets** também oferece uma ferramenta gráfica para a criação, hospedagem, controle e visualização dos aglets criados pelo usuário. Essa ferramenta é denominada **Tahiti**.

Tendo implementado seu *aglet*, o usuário o instância através da ferramenta e pode, de forma intuitiva, requisitar a migração desse agente para outros servidores. O **Tahiti** também compreende um servidor, o que significa que permite o recebimento de *aglets* provenientes de outras máquinas. Além disso, permite também a definição de políticas de segurança, tais como recursos locais que podem ser acessados por um *aglet*, além de cadastramento de hosts dos quais o servidor pode aceitar aglets migrantes.

Na foto apresentada na Figura 3.4 vemos a tela do **Tahiti**.

Migração de Estado em Aglets

Quando um *aglet* é despachado, suspenso ou clonado, ele é seriado através do mecanismo padrão de seriação de **Java**. Os objetos referenciados pelo aglet podem fazer referências a outros objetos e assim sucessivamente, o que

Figura 3.4: Tela do Tahiti

gera um grafo de objetos. É sobre esse grafo que o processo de seriação opera, preservando-o completamente. No entanto, para que isso ocorra, todos os objetos do grafo devem implementar a interface `java.io.Serializable`, ou a interface `java.io.Externalizable`, ou, ainda, serem definidos como `transient`. Se um objeto do grafo não atender uma dessas condições, a exceção `java.io.NotSerializableException` é lançada e o processo falha.

Como exemplo, tomemos o grafo orientado representado na Figura 3.5. Nesse grafo, os vértices representam objetos e os arcos representam uma referência de um objeto para outro. Suponha que o *aglet* seja representado pelo vértice A e que todos os objetos implementem a interface `java.io.Serializable`. No momento da seriação de A, todos os objetos desse grafo seriam também seriados de forma consistente. Isso significa, no exemplo, que no momento da desseriação, o grafo seria restaurado corretamente, ou seja, A e B continuarão referenciado o mesmo vértice D, e não cópias distintas dele.

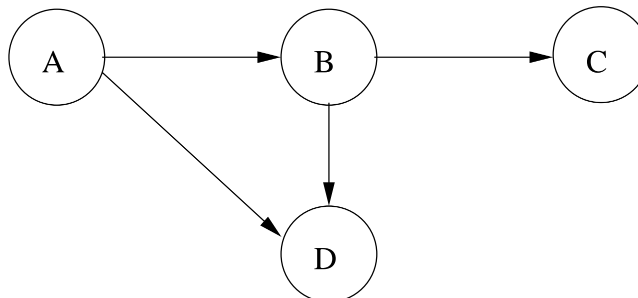

Figura 3.5: Exemplo de Seriação

Observemos, ainda, alguns exemplos de código:

```
public class MeuAglet extends Aglet {
  private Hashtable tabela;
  transient InputStream in = new InputStream();
  int i = 0;
}
```

Neste exemplo, `tabela`, por ser um *Hashtable*, será transferida. A variável `in`, por ter sido definida como `transient`, não será transferida; `i` será transferida por ser de um tipo primitivo.

Vejamos agora outro exemplo de código:

```
public class DadosTransfer implements java.io.Serializable {
  Hashtable dados;
}

public class DadosNaoTransfer {
  int dados;
}

public class MeuAglet extends Aglet {
  static int vc;
  private transient InputStream in = new InputStream();
  private int valor;
  private String s;
  private DadosTransfer t = new DadosTransfer();
  private AgletProxy p;

}
```

Neste exemplo, no momento da transferência de MeuAglet, `vc` não é transferido por ser uma variável de classe. A variável `in` não será transferida por ser `transient`. `valor` é transferido por ser uma variável de tipo primitivo. A variável `s` é transferida, porque `String` é uma classe que implementa `java.io.Serializable`, o mesmo acontecendo com a variável `t`. É interessante notar que o `AgletProxy p` continuará referenciando o mesmo aglet remoto, mesmo depois da transferência.

Vamos, agora, mudar o código anterior para algo como:

```
public class DadosTransfer implements java.io.Serializable {
  Hashtable dados;
}

public class DadosNaoTransfer {
  int dados;
}

public class MeuAglet extends Aglet {
  static int vc;
  private transient InputStream in = new InputStream();
  private int valor;
  private String s;
  private DadosTransfer t = new DadosTransfer();
  private AgletProxy p;
  private DadosNaoTransfer nt = new DadosNaoTransfer();
}
```

Neste exemplo, a tentativa de seriação de `MeuAglet` gerará o lançamento da exceção `java.io.NotSerializableException`, pois `nt` não implementa `java.io.Serializable`. Isso abortará o processo inteiro de seriação.

3.3 Utilização do Aglets

Após vermos alguns ambientes para agentes móveis e embasarmos a nossa escolha, vimos neste capítulo uma introdução ao funcionamento do sistema **Aglets**. No capítulo seguinte, veremos com detalhes como ele foi usado na nossa implementação.

Capítulo 4

MobiGrid

4.1 Arquitetura do Arcabouço

Dentro do InteGrade, encaminhamos nossos esforços para a criação de um arcabouço de suporte a agentes móveis, os quais são utilizados para encapsular aplicações seqüenciais com longo tempo de execução, que não necessitam de comunicação com outras aplicações e que dificilmente terminariam no tempo ocioso disponível em uma única máquina. Esse arcabouço foi desenvolvido sobre o ambiente **Aglets** e foi batizado com o nome de **MobiGrid** [BG04].

4.1.1 Principais Desafios

Na criação desse arcabouço, existiram três motivações principais:

1. Como evitar que os agentes móveis tenham sua execução subitamente interrompida por uma falha (por exemplo, falta de energia na máquina onde executam) sem perder processamento já realizado?

2. Como liberar a máquina para o usuário local, de uma forma rápida e que não perca o processamento já realizado?

3. Como usar **Java** nos clientes sem tornar a máquina lenta para o usuário local?

Clonagem e Redundância

Os problemas 1 e 2 estão intimamente relacionados. De fato, o problema 2 é um relaxamento do problema 1, já que a máquina local deve ser liberada, mas ainda há algum tempo para que o agente móvel possa tomar alguma ação.

No caso do problema 2, existem basicamente quatro ações possíveis para um agente móvel quando o recurso local não se encontra mais disponível:

1. migração;
2. suspensão;
3. terminação;
4. migração com redundância.

A migração pode causar ao usuário local uma perda de desempenho, visto que pode ser um processo caro: por exemplo, um agente móvel levando consigo uma grade quantidade de dados. Isso violaria um dos objetivos do **InteGrade**, que é a sobrecarga quase imperceptível para o usuário local. Além do mais, no caso do problema 1, com a morte da uma máquina, por exemplo, o agente móvel não teria tempo para migrar, visto que esse é um evento inesperado.

A suspensão do agente móvel, isto é, seu descarregamento em disco para uma posterior tomada de execução pode não ser um boa solução nem para o problema 1, nem para o problema 2, visto que a máquina poderia ficar indisponível por um período muito longo: dias, talvez. Dessa forma, o agente móvel demoraria muito tempo para retornar ao usuário que o submeteu. Uma outra solução baseada na suspensão poderia ser imaginada: o descarregamento remoto de tempos em tempos, via rede, do estado do agente, algo similar a *checkpointing*. Todavia, consideramos que essa solução pode causar uma grande sobrecarga na rede, no caso de agentes móveis carregando muitos dados consigo. Por último, a terminação não foi considerada, visto que não queremos perder processamento já realizado.

Diante desses argumentos, partimos para uma quarta opção, que seria a utilização da migração aliada à redundância. Com a redundância, sempre existem duas ou mais cópias de um agente móvel rodando independentemente; em caso de uma terminação súbita, pode-se clonar um dos agentes móveis restantes, migrando o clone para uma máquina ociosa. Essa solução falharia na improvável situação de uma morte súbita dos agentes móveis antes de o arcabouço tomar conhecimento da situação, mas essa probabilidade pode ser diminuída alocando-os em máquinas de redes diferentes. Além do mais, verificamos que a redundância pode ser uma ótima solução para o problema

2: quando o usuário local retornasse à sua máquina, a infra-estrutura poderia descartar os agentes que lá estivessem executando, clonando suas cópias e despachando-as para novas máquinas.

Obviamente, a redundância não constitui uma boa solução quando a quantidade de máquinas ociosas disponíveis na grade for muito pequena. Todavia, essa situação tende a ser muito rara em grades com grande quantidade de nós; além disso, dada a característica oportunística dos agentes móveis, até pequenas fatias de tempo ocioso podem ser por utilizadas, o que aumenta as possibilidades para a escolha de máquinas ociosas.

Assim sendo, optou-se unicamente pela migração com redundância, solicitando ao usuário uma nova submissão no caso de terminação súbita de todas as cópias de um agente móvel.

Utilização de um Daemon

Optou-se por atacar o problema 3 utilizando um *daemon*. Esse *daemon* pode ser inserido no LRM (*Local Resource Manager*, Gerenciado Local de Recursos) do InteGrade [GKG+04], o qual é responsável por monitorar os recursos locais. Os LRMs mandam essa informação periodicamente ao GRM (*Global Resource Manager*, Gerenciador Global de Recursos) que as usa para escalonar processos na grade. Esse *daemon* tem a função de ligar o servidor de hospedagem de agentes móveis somente quando a máquina está ociosa. Esse servidor contém trechos de código **Java** mais pesados. O *daemon* detectará quando o usuário local está de volta à sua máquina e informará ao servidor, o qual pode ser desligado.

Essa abordagem tem a vantagem de só ligar a JVM quando o usuário local não estiver fazendo uso de sua máquina. Sendo assim, evita-se que o usuário local perceba uma queda de desempenho por conta dos recursos utilizados pela JVM, principalmente memória.

4.1.2 Arquitetura

A idéia geral do nosso arcabouço é fornecer ao programador um ambiente para a programação de suas aplicações longas, as quais passaremos a chamar de *tarefas*.

A seguir, explicitaremos, item a item, quais os principais componentes do arcabouço em um nível bem alto:

- *tarefas*: aplicações longas, encapsuladas dentro de um agente móvel. Na implementação dessas *tarefas*, o programador deve ter a preocupação de salvar o estado da *tarefa* de tempos em tempos, visto que **Aglets** fornece o mecanismo de migração fraca;

- *gerente*: é o componente responsável pelo registro das *tarefas*. Deve estar sempre ativo na máquina do usuário que submeteu uma *tarefa* ao arcabouço. As duas principais funcionalidades do *gerente* são:

  1. migração: quando da submissão da *tarefa*, esse *gerente* consulta a infra-estrutura do InteGrade procurando uma máquina que está ociosa e tem certa probabilidade de permanecer nesse estado por algum tempo. De posse dessa informação, o *gerente* despacha a *tarefa* para tal máquina onde ela passa a ser executada. Quando alguma *tarefa* clone recém-criada necessita migrar, consulta o *gerente* para obter o endereço de uma máquina ociosa. O *gerente* obtém essa informação utilizando a infra-estrutura do *InteGrade*;

  2. *vivacidade*: o *gerente* também se encarrega de criar clones de cada tarefa e de despachá-los para uma máquina distinta da *tarefa* clonada, para garantir a *vivacidade* da mesma. Chamamos de *vivacidade* a qualidade de uma tarefa estar sendo executada em alguma máquina. Quando da morte de um dos gêmeos - que pode ocorrer por vários motivos, como, por exemplo, uma interrupção inesperada de energia na máquina onde a *tarefa* está executando - o *gerente* clona o gêmeo ainda vivo e o despacha para outra máquina. Utilizamos aqui a palavra *gêmeos* para representar as cópias de uma *tarefa*. A quantidade de *gêmeos* que uma tarefa possui é uma parâmetro configurável pelo usuário. A escolha por um determinado número de gêmeos depende de um compromisso entre a quantidade de máquinas disponíveis e o nível de garantia desejado para que uma *tarefa* não termine subitamente. Um número maior de *gêmeos* consumiria mais recursos que poderiam ser utilizados de outra maneira dentro da grade; por outro lado, garantiria uma probabilidade menor de uma terminação súbita de todos os gêmeos, sem que houvesse tempo para alguma reação do *gerente*.

- *servidor*: servidor instalado em cada uma das máquinas que oferece recursos ao arcabouço. Oferece um ambiente para execução das *tare-*

fas. Quando solicitado pelo *daemon*, o *servidor* é terminado, causando a morte das *tarefas* que eram hospedadas por ele. Ao perceberem a morte dessas *tarefas*, os *gerentes* delas providenciam a clonagem de alguma *tarefa* gêmea. Essas *tarefas*, por sua vez, consultam os *gerentes*, os quais se comunicam com o InteGrade à procura de máquinas ociosas. Quando obtêm tal informação, os *gerentes* providenciam o despacho dessas tarefas para novas máquinas;

- *daemon*: encarrega-se de verificar se a máquina está ociosa ou não. Quando ociosa, ele avisa ao InteGrade e levanta o *servidor*, para o recebimento de *tarefas*. Quando da requisição da máquina por parte do seu usuário, o *daemon* desliga o servidor. Assim, o *daemon* é responsável por informar ao InteGrade de que uma máquina está pronta para receber *tarefas*;

- *cliente*: componente utilizando para a submissão de *tarefas* ao arcabouço. Também oferece um ambiente de hospedagem ao *gerente*.

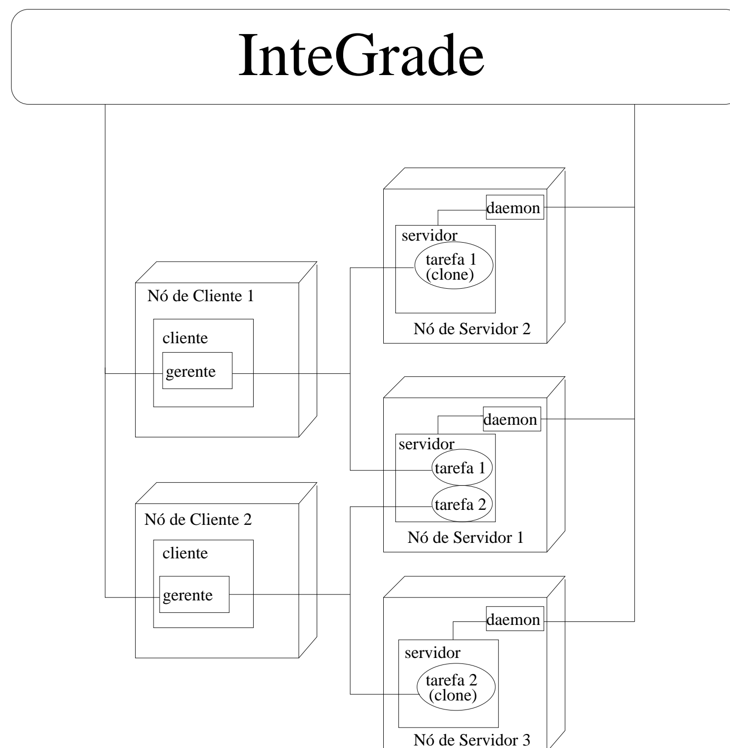

Figura 4.1: Arquitetura Geral do Arcabouço

Na Figura 4.1, podemos ter uma melhor visualização do arcabouço. Observe que cada um dos *clientes* hospeda um *gerente*, o qual se comunica com

o InteGrade. O *gerente* do **Cliente 1** gerencia a **Tarefa 1** e sua cópia; o *gerente* do **Cliente 2** gerencia a **Tarefa 2** e sua cópia. Note também que os *daemons* comunicam-se com o InteGrade para avisá-lo quando a máquina local está ociosa, disparando então a execução do servidor.

4.2 Estrutura de Classes do Arcabouço

4.2.1 Descrição das Classes e Diagrama de Classes UML

Diante da visão geral apresentada, explicitaremos melhor sua estrutura de classes, mostrando o que já se tem implementado, bem como as limitações ainda existentes. A estrutura inicial de classes do nosso arcabouço é a seguinte:

- `AgletTask`: essa classe representa, no arcabouço, as *tarefas* já citadas anteriormente. Para definir sua *tarefa*, o programador deve estender esta classe e redefinir o método **defineState()**, o qual deve devolver um `AgletState`, que encapsula a implementação da *tarefa*, bem como seu estado. A decisão por essa separação foi tomada por motivos de clareza. A classe `AgletTask` pode ocupar-se do serviço "sujo" de ler os dados necessários à criação de um objeto `AgletState`. Sendo assim, a classe `AgletState` fica mais "limpa", contendo somente código relativo a implementação da *tarefa* em si, bem como os dados sobre os quais ela opera. A classe `AgletTask` está associada com a classe `AgletState`;

- `AgletState`: Classe que representa o estado da *tarefa*, bem como sua implementação. O programador deve estendê-la e implementar os métodos:

  - `run()`: método que define a implementação da *tarefa*, propriamente dita, ou seja, aplicação de longa duração que o usuário quer submeter ao arcabouço. O programador deve tomar conta do estado atual do processamento dentro das seguintes premissas:

    1. conforme o processo de migração de estado em **Aglets** descrito na Subseção 3.2.3, o grafo de objetos do tipo `java.io.Serializable` e do tipo `java.io.Externalizable` referenciados como não `transient` no `AgletState` serão automaticamente preservados quando da migração da *tarefa*. As variáveis de tipo primitivo (por exemplo, `int`, `double`, etc.) desses objetos terão seus valores preservados também;

2. toda vez que o estado corrente do processamento chegar a um ponto consistente, ou seja, um ponto em que os invariantes do algoritmo estão preservados, uma chamada ao método `checkPoint()` deve ser efetuada. Esse método avisa ao arcabouço que o estado está consistente e devolve `true` se o `AgletTask` recebeu uma requisição de migração. Nesse caso, o programador deve implementar uma maneira de se interromper o algoritmo, para que o estado permaneça consistente, a fim de que a execução possa ser retomada futuramente. Caso o `AgletTask` não tenha recebido uma requisição de migração, o valor `false` é devolvido. Vale notar que esse método não realiza um processo de *checkpointing*, isto é, não realiza um descarregamento do estado corrente em disco. O nome do método está relacionado à idéia de ponto de verificação, ou seja, um ponto onde a infra-estrutura deve ser notificada de um fato relevante;
3. o programador também deve ter em mente que, ao final da execução da tarefa, deve efetuar a chamada do método `finish()`, para que a infra-estrutura seja notificada que a execução foi terminada.

- `printResults()`: método que entrega os resultados do processamento da *tarefa*. Chamado pelo arcabouço quando a tarefa terminou seu processamento e retornou ao *cliente*.

- `Server`: é a classe que implementa o *servidor* que hospeda os `AgletTasks`;

- `AgletManager`: representa o *gerente* já citado. O `AgletManager` não é nada mais do que um tipo especial de *aglet* que nunca migra. Ele foi implementado dessa forma a fim de usar as ferramentas do **Aglets** para troca de mensagens entre *aglets*;

- `Client`: faz as vezes do *cliente*, ou seja, permite a submissão de `AgletTasks` ao arcabouço. Também serve para hospedar o `AgletManager`, que nada mais é que um tipo especial de *aglet*. Por enquanto estamos usando o **Tahiti** como cliente, visto que essa ferramenta de visualização do **Aglets** fornece um ambiente para hospedar o `AgletManager` e ferramentas para submeter *tarefas* à infra-estrutura;

- `AgletProxy`: elemento do ambiente **Aglets**. É a classe que interme-

deia as comunicações entre *aglets*. Em nosso arcabouço, possibilita a comunicação entre o `AgletTask` e seu gerente `AgletManager`, entre o `AgletManager` e os `AgletTasks` que gerencia, e entre o `Server` e os `AgletTasks` que hospeda.

- `AgletListener`: Classe que executa as operações pré-migração (`onDispatching()`) e pós-chegada em uma nova máquina (`onArriving()`). Essa classe é de uso interno do arcabouço e é transparente ao usuário.

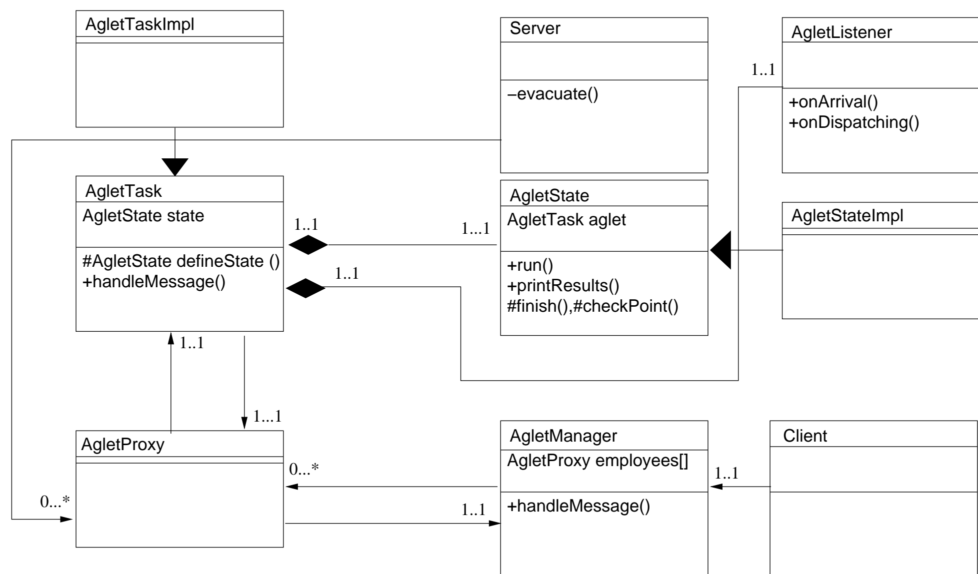

Figura 4.2: Diagrama de Classes do Arcabouço

Na Figura 4.2, `AgletStateImpl` e `AgletTaskImpl` são as implementações do usuário para, respectivamente, `AgletState` e `AgletTask`. Note que cada `AgletTask` possui um, e somente um, `AgletState` e `AgletListener`. O `Server` tem uma lista dos `AgletProxies` que representam os `AgletTasks` que ele hospeda. O `AgletTask`, possui uma, e somente uma, referência para o `AgletProxy` que representa seu `AgletManager`. O `AgletManager` possui uma lista de `AgletProxies` que representam os `AgletTasks` que ele gerencia.

4.2.2 Exemplo de Código

Mostraremos agora a implementação de uma *tarefa* do tipo *Hello World*. Essa implementação é meramente um exemplo de como utilizar nosso arcabouço, visto que ela não possui nenhuma utilidade prática. Note que, sempre que a *tarefa* migra, quando ela chega ao seu novo hospedeiro, o método

`run()` é executado. A *tarefa* sabe quantas vezes ela já disse "hello" em outras máquinas, preservando o processamento já executado. Quando ela migra, todas as variáveis de instâncias de `AgletStateImpl` são preservadas, a saber `quantasVezes`.

Toda vez que uma iteração do laço do método `run()` termina, uma chamada a `checkPoint()` é executada para avisar a infra-estrutura de que um ponto consistente foi alcançado. Se o método devolve o valor `true`, o laço é interrompido, já que a *tarefa* tem que migrar.

```
public class AgletTaskImpl extends AgletTask {
   public AgletState defineState() {
      String input = "";
      BufferedReader br = null;
      int vezes = 1;
      try {
         System.out.println("Quantas vezes devo dizer ``Olá''?");
         br = new BufferedReader(new InputStreamReader(System.in));
         input = br.readLine();
         vezes = Integer.parseInt(input);
      } catch (Exception e) {
         System.out.println("Erro na leitura da URL");
         e.printStackTrace();
      }
      return new AgletStateImpl(vezes);
  }
}
public class AgletStateImpl extends AgletState {
   int quantasVezes = 0;
   int max;
   public AgletStateImpl(int vezes) {
      max = vezes;
   }
   public void run() { /* diga ola até a hora de migrar */
      System.out.println(``Eu já disse ola '' +  quantasVezes +
                         ``vezes'');
      for (;quantasVezes < max; ++quantasVezes} {
```

```
            System.out.println(‘‘Olá!’’);
            if (checkPoint () == true) break;
        }
        if (quantasVezes >= max) finish();
    }
    public void printResults() { /* exibindo resultados */
        System.out.println(‘‘Estou de volta! Eu já disse olá ’’ +
                           max +  ‘‘ vezes.’’);
    }
}
```

Observe também, nesse exemplo de código, que o código de leitura dos dados para a criação de um objeto `AgletState` está confinado ao `AgletTask`.

4.3 Casos de Uso

Vamos descrever agora os principais casos de uso de nosso arcabouço, a fim de fornecermos uma visão melhor do seu funcionamento na prática. Selecionamos três casos de uso a saber:

1. Submissão de uma *tarefa* ao arcabouço por parte do usuário;
2. Usuário local volta à uma máquina que executa um *servidor* que hospeda uma *tarefa*;
3. Monitoramento do *gerente* ao perceber a morte de alguma *tarefa*.

4.3.1 Submissão de *tarefa*

Atores:

- usuário que quer submeter *tarefas*;
- *gerente*;
- InteGrade.

1. O usuário cria uma *tarefa* de tal forma que, de tempos em tempos, ela faça uma chamada ao método `checkPoint()`;
2. O usuário submete a *tarefa* ao *gerente*;

3. O *gerente* procura um *servidor* ocioso no InteGrade e despacha a *tarefa* para lá;
4. O *gerente* clona a *tarefa* e despacha o clone para outro *servidor* ocioso.

4.3.2 Volta do Usuário Local

Atores:

- usuário local;
- *daemon*;
- *servidor*.

1. O usuário local volta à sua máquina;
2. O *daemon* percebe a volta do usuário e requisita ao *servidor* que finalize sua execução;
3. O *servidor* termina sua execução, matando as *tarefas* que hospeda.

4.3.3 Monitoramente do Gerente

Atores:

- usuário local;
- *gerente*;
- *tarefa* a ser clonada;
- InteGade.

1. O *gerente* percebe a morte de uma *tarefa*;
2. O *gerente* procura uma *tarefa* gêmea e a clona;
3. O *gerente* pesquisa no InteGrade por um servidor ocioso e despacha a *tarefa* clone para lá.

4.4 Questões Mais Específicas

Apresentadas a arquitetura e a estrutura de classes do MobiGrid, trataremos, no próximo capítulo, de questões mais específicas no arcabouço a respeito de dois conceitos já apresentados: O estado de execução e a *vivacidade*.

Capítulo 5

Estado de Execução e Vivacidade

5.1 Preservação do Estado de Execução

Como já citado em 3.1.3, **Aglets** fornece somente o recurso de migração fraca, devido a restrições da JVM. Sendo assim, algum subsídio para auxiliar o programador na preservação do estado de execução deverá ser fornecido pelo nosso arcabouço.

5.1.1 Solução Proposta

Propomos uma solução que facilita o trabalho do programador em salvar o estado atual da *tarefa*. Essa solução é simples e praticamente não introduz sobrecarga significativa.

Oferece-se ao programador o método `checkPoint()`. A chamada a esse método deve ser feita toda vez que a lógica do algoritmo de `AgletState.run()` chega em um ponto que satisfaz as seguintes condições:

- os invariantes do algoritmo estão preservados;
- os dados sobre as quais o algoritmo opera estão consistentes.

Além disso, o algoritmo de `run()` deve ser implementado de forma que em uma re-execução desse método, no caso de uma migração para outra máquina, o processamento já realizado não seja perdido.

O método `checkPoint()` funciona da seguinte forma:

1. Toda vez que o `AgletTask` vai migrar ou ser clonado, a requisição de migração ou clonagem levanta uma condição de espera no objeto `AgletTask`;

2. O método `checkPoint()` faz um `notify()` no objeto `AgletTask` e devolve `true` se o objeto `AgletTask` está em espera e `false` caso contrário;

3. O programa deve verificar o valor devolvido por `checkPoint()` e deve finalizar a execução do algoritmo se o valor for `true`.

4. No caso da clonagem, o `AgletTask` clonado faz um nova chamada ao método `run()` do `AgletState` para a retomada da execução.

5.1.2 Limitações

Com essa semântica, garante-se que o `AgletTask` migra ou é clonado em um estado consistente, o que permite sua re-execução no destino.

Essa solução tem a limitação de não enfrentar o problema de preservação da pilha de execução no caso de recursões, por exemplo. Nesse caso, o programador deve abrir essas recursões, fazendo uso explícito de pilhas.

Além disso, o programador deve criar uma lógica que não fique um longo período de tempo sem fazer chamada ao método `checkPoint()`, pois quanto maiores os intervalos entre as chamadas, maior o tempo de processamento que pode ser eventualmente perdido. Além disso, maiores intervalos entre as chamadas acarretam maior tempo para liberação da máquina.

5.1.3 Um exemplo

Mostraremos agora como o algoritmo *Insertion Sort* pode ser implementado dentro de nosso arcabouço:

```
public class SaveAglet extends AgletTask {
    public AgletState defineState () {
        return new SaveAgletState ();
    }
}
public class SaveAgletState extends  AgletState {
    boolean termina;
    int []vetor;
```

```
int last;

public SaveAgletState () {
  this.last = 1;
  vetor = new int[100000];
  Random r = new Random (23);
  /* Criação de um vetor com números pseudo-aleatórios */
  for (int i = 0; i < vetor.length; ++i) {
    vetor[i] = r.nextInt ();
  }
}
public void run () {
  int i, j;
  this.termina = false;
  /* laço que realiza a ordenação */
  for (i = this.last; !this.termina && i < vetor.length; ++i) {
    int n = vetor[i];
    for (j = i - 1; j >= 0; --j) {
      if (vetor[j] > n)
       vetor[j + 1] = vetor[j];
      else
       break;
    }
    vetor[j + 1] = n;
    this.last = i + 1;
    if (this.checkPoint () == true)
      this.termina = true;
  }
  if (i >= vetor.length)
      this.finish ();
}
public void printResults () {
  for (int i = 0; i < vetor.length; ++i) {
    System.out.println (vetor[i]);
  }
}
```

```
}
```

Note que o laço principal de `run()`, faz `i` variar de `last` até o tamanho do vetor, o que garante que o processamento já realizado não seja perdido. Observe também que o método `checkPoint()` é chamado quando o invariante do algoritmo está preservado: $a[0] \leq a[1] \leq, ..., a[last - 1]$. Além disso, `vetor` está em um estado consistente, ou seja, não contém nenhum valor que atrapalhe a retomada da execução. Outro ponto interessante é a verificação que o programa faz pelo valor devolvido por `checkPoint()`, parando a execução do programa, caso o valor devolvido seja `true`, evitando que o algoritmo entre em um estado inconsistente.

5.1.4 Experimentos para Medição de Sobrecarga

Para avaliarmos o impacto de nossa estratégia baseada no método `checkPoint()` sobre o tempo de migração, realizamos alguns experimentos.

O primeiro experimento foi realizado com um `AgletTask` que encapsula o algoritmo *Insertion Sort* já descrito. Medimos o tempo (em ms) de migração de um `AgletTask` operando sobre 100.000, 200.000 e 300.000 inteiros, utilizando a nossa estratégia (E), o que significa que a infra-estrutura aguarda que o algoritmo chegue a um ponto consistente para só então liberar a migração. Medimos também o tempo de migração do mesmo *AgletTask* sem a utilização da estratégia (S), ou seja, quando da requisição de migração, o *AgletTask* migra imediatamente, o que não garante mais a consistência do estado de execução do algoritmo. Para cada instância do problema, realizamos 5 experimentos. A Tabela 5.1 apresenta a média e o desvio padrão dos tempos de migração para cada instância do problema. A Figura 5.1 apresenta o gráfico dos valores da Tabela 5.1.

Instância	Média(E)	$\delta(E)$	Média(S)	$\delta(S)$	Razão E/S
100.000	33203,2	1786,83	32913,2	1563,61	1,01
200.000	64156,4	2904,77	63649,4	2052,44	1,01
300.000	92794,2	293,42	95680,2	2855,04	0,97

Tabela 5.1: Experimento para o `AgletTask` do *Insertion Sort*

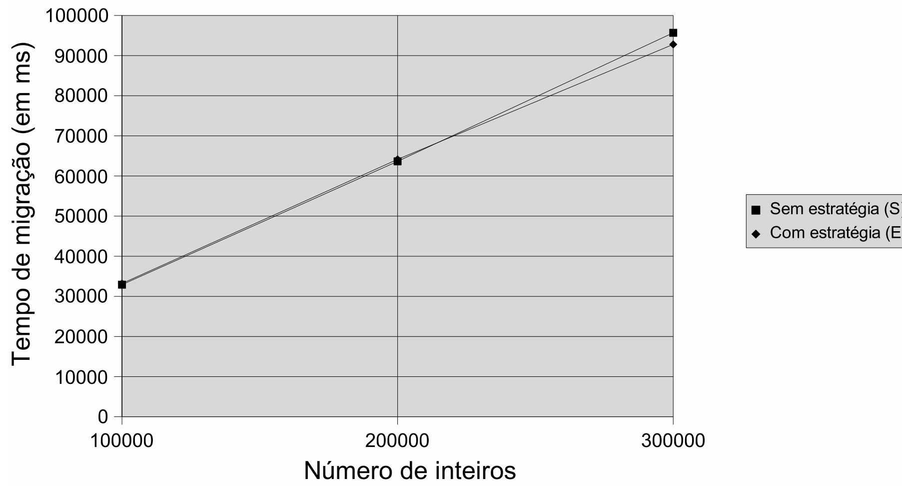

Figura 5.1: Gráfico dos experimento da Tabela 5.1

Realizamos também um experimento com um `AgletTask` que encapsula um algoritmo que resolve o *problema do makespan*[1]. Esse é um problema NP-difícil para o qual foi implementada uma solução força-bruta exponencial $O(m^n)$ que verifica todas as combinações possíveis. Comparamos o tempo médio de 5 migrações de um `AgletTask` operando sobre a instância $m = 2, n = 30$ do problema utilizando a nossa estratégia (E) e também sem utilizar nossa estratégia (S). A tabela 5.2 apresenta os resultados desse experimento.

Média(E)	$\delta(E)$	Média(S)	$\delta(S)$	Razão E/S
3671	181,17	3698	135,03	0,99

Tabela 5.2: Experimento para `AgletTask` do *makespan*

Podemos observar que, para algoritmos que fazem chamadas freqüentes ao método `checkPoint()`, o tempo de migração não sofre uma sobrecarga significativa. De fato, a sobrecarga máxima no tempo de migração para `AgletTasks` que usam a estratégia foi de 1%, sendo que na Tabela 5.1 e na Tabela 5.2 podemos ver que houve, em alguns casos, um tempo de migração médio menor para `AgletTasks` que faziam uso da estratégia (3% e 1% a menos); acreditamos que essa pequena variação inesperada esteja dentro da margem de erro de nossas

[1]Determinar uma distribuição de n tarefas independentes (note que aqui a palavra *tarefa* denota seu significado habitual, não o significado que adquire dentro do nosso arcabouço) com tempos de execução $t_1, t_2, ..., t_n \in \mathbb{Z}^+$ em m máquinas minimizando o tempo em que a última tarefa termina.

medições. Dessa forma, vemos que os experimentos reforçam o argumento de baixa sobrecarga de nossa estratégia.

5.2 Controle de Vivacidade

Dentro do arcabouço, existem dois recursos que garantem a *vivacidade* de uma *tarefa*: a migração e a clonagem.

A migração preserva a *vivacidade* no sentido em que permite que uma *tarefa* encontre um novo *servidor* para continuar sendo executada. Essa migração pode ser uma migração simples, ou seja, uma migração por motivos de esvaziamento de um *servidor*, ou uma migração de um clone recém criado de uma *tarefa*.

Por outro lado, a *clonagem* pode ser vista também como uma estratégia de recuperação de erro, na eventualidade de uma desligamento inesperado de um *servidor* que não permita a migração da *tarefa*. A clonagem funciona então, conforme já citado, como um tipo de redundância. Além disso, em nosso trabalho, percebemos também a potencialidade da clonagem como ferramenta de agilização da migração, conforme já descrito.

O conceito definido em nosso trabalho como "vivacidade" não é uma novidade na área de sistemas de computação. De fato, ao longo de nosso projeto encontramos alguns trabalhos que utilizam ou utilizaram algumas idéia semelhantes. O projeto **Somersault** [MFHV98] dos Laboratórios da **HP** em Bristol é um deles. O **Somersault** é uma plataforma para o desenvolvimento de componentes de software distribuídos tolerantes a falha. Processos críticos de uma aplicação são espelhados sobre uma rede. Em **Somersault** esse processos espelhados constituem uma unidade de tolerância a falha. Cada unidade é composta basicamente de dois processos, o primário e o secundário, onde o secundário é uma réplica fiel do primeiro. Na realidade, **Somersault** usa um redundância muito mais estrita que **MobiGrid**, visto que mantém as réplicas sincronizadas.

Mostraremos agora, de forma mais detalhada, como funcionam a migração e a clonagem em **MobiGrid**, bem como a interação desses recursos com os recursos de preservação do estado de execução.

5.2.1 Migração

Quando o `Server` necessita despachar seus `AgletTasks`, ele envia para cada um deles uma mensagem do tipo "evacuate". O método `handleMessage()` trata essa mensagem e o `AgletTask` envia uma mensagem assíncrona do tipo "evacuate" para o `AgletManager`.

Quando recebe a mensagem de despacho, o `AgletManager` promove o despacho do `AgletTask` e atualiza seu `AgletProxy`. O método `onDispatching()` do `AgletListener` ao receber a requisição de migração, espera que o `AgletState.run()` chame o método `checkPoint()`. Pelo fato de o método `onDispatching()` estar nessa espera, o método `checkPoint()` devolve o valor `true`, obtendo essa informação através do método `ìsEvacuating()`. O método `run()`, diante desse valor devolvido, interrompe sua execução e o `AgletTask` migra.

A Figura 5.2 esquematiza esse protocolo. Nessa figura, as interações entre o `AgletTask` e o `AgletManager` e vice-versa são intermediadas por objetos da classe `AgletProxy`. No entanto, para evitar poluição visual na figura, representamos essas interações como se ocorressem diretamente.

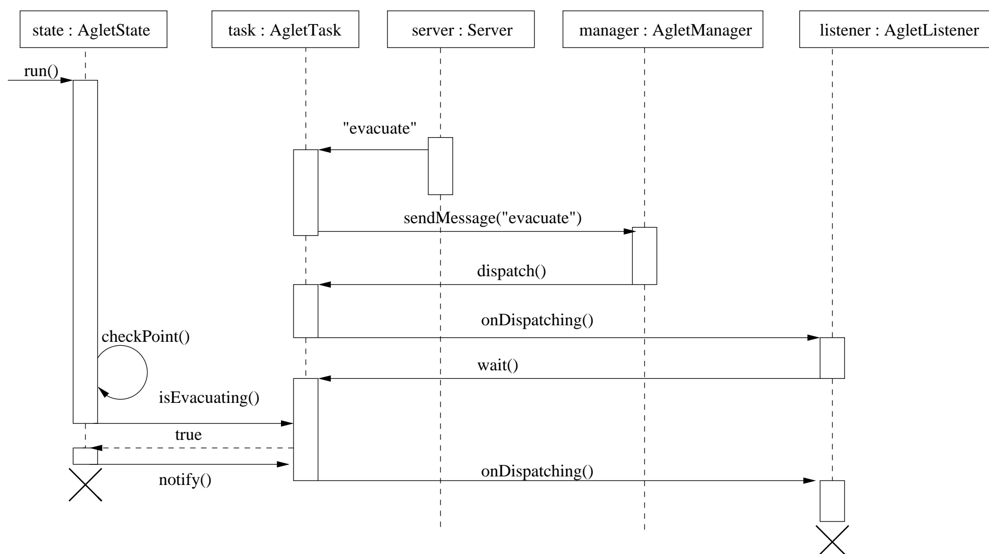

Figura 5.2: Protocolo de Migração

Observando-se o protocolo de migração, nota-se que uma otimização poderia ser feita. Se o método `isEvacuating ()` devolvesse `true` desde o momento do recebimento da mensagem "evacuate" pelo `AgletTask`, uma chamada anterior de `checkPoint()` já poderia terminar a execução de `run()` e permitir a migração assim que o `AgletManager` a requisitasse. Todavia, uma implementação desse tipo, amarraria todo o mecanismo de migração com a uti-

lização da mensagem "evacuate". Em outras palavras, o `AgletManager` só poderia requisitar a migração do `AgletTask` através do envio da mensagem "evacuate". Diante dessa limitação, preferimos manter a implementação na forma descrita, visto que ela permite uma migração consistente mesmo com uma chamada simples ao método `dispatch()` do `AgletProxy` representante, o que garante uma maior flexibilidade ao nosso arcabouço.

Além do mais, levando-se em conta os experimentos descritos no Capítulo 5, podemos verificar que nossa implementação de `checkPoint()` não traz uma sobrecarga significativa ao tempo de migração.

5.2.2 Clonagem

O protocolo de clonagem funciona de maneira muito parecida ao protocolo de migração.

A linha de execução de execução do `AgletTask` é representada pelo método `AgletState.run()`. Quando uma requisição de clonagem vem do `AgletManager`, ela é tratada pelo objeto método `onCloning()` da classe `CloneAdapter`, a qual está associada ao `AgletTask`. Nesse ponto, o método entra em espera por uma chamada a `checkPoint()` de `AgletState` por parte de `AgletState.run()`, o qual deve ter sua execução suspensa em seguida, devido ao valor `true`, devolvido por `checkPoint()`.

Nesse ponto, ocorre a clonagem. O `AgletTask` original volta a executar `run()` e o `AgletTask` clone executa `CloneAdapter.onClone()`, o qual envia ao `AgletManager` um `AgletProxy` seu via mensagem assíncrona do tipo "cloned". Esse envio de `AgletProxy` foi uma maneira que encontramos de contornar um problema do **Aglets** que, em sua documentação, afirma que o método `clone()` do `AgletProxy` devolve um `AgletProxy` para o `Aglet` clonado, o que não acontece na realidade. A classe `CloneAdapter` é um adaptador fornecido pelo **Aglets**, o qual pode ser associado a um *aglet*, para o recebimento e tratamento de eventos de clonagem.

Depois disso, o `AgletTask` clone passar a executar seu `run()` até receber ordem de despacho proveniente do `AgletManager`. A partir daí, a migração do clone ocorre de acordo com o protocolo de migração já descrito.

A Figura 5.3 esquematiza todo esse protocolo. Nessa figura, a exemplo da Figura 5.2, as interações entre o `AgletTask` e o `AgletManager` e vice-versa são intermediadas por objetos da classe `AgletProxy`.

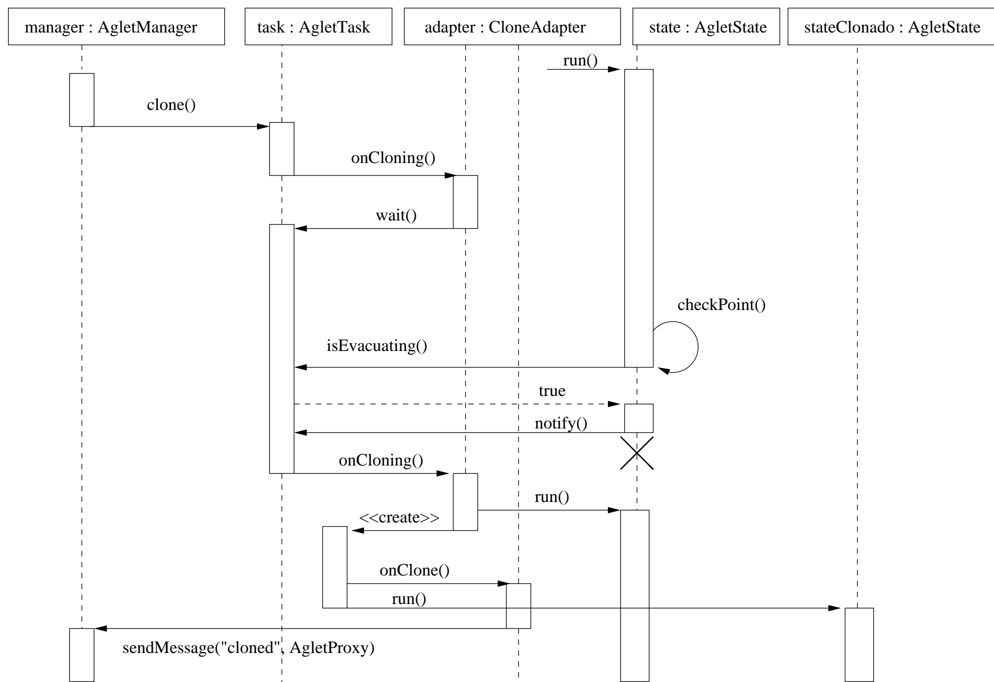

Figura 5.3: Protocolo de Clonagem

5.2.3 Políticas para Falhas

Ausência de Chamadas a `checkPoint ()`

Diante do protocolo descrito, resta uma decisão quanto ao que fazer com *tarefas* que não fazem chamadas freqüentes ao método `checkPoint ()`. Duas abordagens imediatas podem ser consideradas:

1. Uma abordagem conservadora, na qual o `gerente` efetuaria um controle da periodicidade com que as *tarefas* efetuam chamadas ao método `checkPoint()`, destruindo as que permanecessem muito tempo sem efetuar chamadas ao método `checkPoint()`;

2. Uma abordagem liberal, que não faz essa exigência mas, que numa situação de necessidade de migração ou clonagem, espera uma determinada quantidade de tempo pela chamada ao método `checkPoint()`, efetuando a clonagem ou migração à revelia, por *timeout*.

Decidimos pela segunda política, por essa ser mais flexível e simples, assemelhando-se à política de exceções de linguagens de programação modernas, que procuram manter os programas executando, mesmo com a ocorrência de erros. Dessa forma, alguma mensagem deve ser exibida ao usuário que submeteu a *tarefa*,

deixando-o ciente de que houve alguma migração e ou clonagem forçada, o que implica que a *tarefa* pode não ter chegado a um resultado correto. Dentro dessa abordagem, é preocupação do programador garantir que sua *tarefa* faça chamadas corretas ao método `checkPoint()` freqüentemente.

5.3 Funcionamento do MobiGrid

Abordadas as questões de preservação do estado de execução e de controle de *vivacidade* das *tarefas*, terminamos de definir, neste capítulo, o funcionamento do **MobiGrid**. Esse embasamento nos encaminha para uma próxima questão, que é como nosso arcabouço funcionaria na prática. Todavia, não dispomos de uma grade computacional suficiente grande para um experimento relevante. Dessa forma, no próximo capítulo, propomos um modelo matemático que visa a capturar o modo de funcionamento de nosso arcabouço, para que possamos realizar experimentos de simulação e, dessa forma, observar como nosso arcabouço se comportaria na prática, com a submissão de uma grande quantidade de *tarefas* em uma grade com grande quantidade de máquinas.

Capítulo 6

Modelo Matemático e Experimentos

6.1 Modelo Matemático

O objetivo desta seção é apresentar uma comparação das duas estratégias que foram utilizadas no **MobiGrid** para liberar a máquina para o usuário local: *migração* e clonagem com *vivacidade*. Baseados nesses experimentos, nós proporemos um modelo matemático cujo objetivo é simular uma rede de nós homogêneos com suas respectivas *tarefas* executando neles. O principal objetivo dessa simulação é estudar a *vivacidade* e seu impacto no sistema [BGK05]. Nosso modelo é uma simplificação da realidade, mas acreditamos que ele reflete a comportamento esperado na prática, visto que procura contemplar as principais características das *tarefas* e de *grades* com grande quantidade de nós.

6.1.1 Experimentos com Clonagem e Migração

Para criar um modelo matemático para nossa simulação, realizamos alguns experimentos que nos encaminharam para algumas simplificações. Nossos experimentos foram realizados em um computador **Athlon XP 2500** de 1830 MHz, com 512 Mb de RAM, executando **Debian/GNU Linux** com kernel 2.6.7. Note que, nesses experimentos, nós desligamos a otimização de migração do **Aglets**, a qual evita nova transmissão de dados descritores de classe já transmitidos. Essa otimização funciona através do mecanismo de *cache*, armazenando localmente dados descritores de classes, sempre que um novo agente

móvel chega ao servidor local.

6.1.2 Experimentos com Migração e Clonagem

O primeiro experimento foi realizado com uma *tarefa* que encapsula um algoritmo de *Insertion Sort*, já citada neste texto em 5.1.2. Medimos o tempo de 5 migrações e 5 operações de clonagem para uma *tarefa* operando sobre 100.000, 200.000 e 300.000 inteiros.

O segundo experimento usa uma *tarefa* que encapsula um algoritmo trivial exponencial para resolução do problema do *makespan* - a qual já foi citada neste texto também em 5.1.4. Esse algoritmo é executado para instâncias (m, n), onde m é o número de máquinas e n é o número de *tarefas*. Medimos também o tempo de 5 migrações e 5 operações de clonagem.

A Tabela 6.1 mostra o tempo médio da migração (M) e da clonagem (C), o desvio-padrão (σ) e a proporção entre o tempo médio de migração e tempo médio de clonagem (M/C).

Experimento	Instância	M	$\sigma(M)$	C	$\sigma(C)$	(M/C)
Experimento 1	100.000	33.203,2	1.786,83	135,6	1,34	244,86
Experimento 1	200.000	64.156,4	2.904,77	361,2	4,09	177,62
Experimento 1	300.000	92,794,2	293,42	427,8	18,1	216,91
(m, n)	(2, 26)	3.698	135,03	15,8	3,49	234,05
(m, n)	(2, 28)	3.769	248,62	15,6	2,3	241,62
(m, n)	(2, 30)	3.671	181,17	15,6	2,7	235,32

Tabela 6.1: Experimentos com migração e clonagem (tempo em ms)

Apesar de os experimentos terem sido realizados com diferentes tipos de *tarefas* - a *tarefa* de ordenação, que leva consigo uma quantidade significativa de dados, e a *tarefa* de *makespan*, que leva uma quantidade pequena de dados, obtivemos proporções similares. Na média, a clonagem foi aproximadamente 200 vezes mais rápida do que a migração. Isso acontece pois a clonagem depende basicamente de operações em memória, que são muito mais rápidas que as operações de E/S necessárias pela migração.

6.1.3 Experimentos com Clonagem e Execução

Este experimento usa a mesma *tarefa* de ordenação descrita acima. Medimos o tempo de 5 execuções completas sem interferência da migração para

100.000, 200.000 e 300.000 inteiros. Medimos também o tempo de 5 execuções completas da *tarefa* enquanto um clone dela está migrando do *servidor* local para outro *servidor*.

A Tabela 6.2 mostra o tempo médio de execução para a *tarefa* executando isoladamente no *servidor* (I), o tempo médio de execução dela enquanto um clone seu migra para outro *servidor* (M), o desvio-padrão σ dessas medições, bem como a razão entre esses tempos (M/I).

Inteiros	I	$\sigma(I)$	M	$\sigma(M)$	Razão (M/I)
100.000	18.833,6	868,5	20.729,8	529,56	1,1
200.000	80.637,8	2.340,83	84.219,8	2.802,44	1,04
300.000	204.950,8	6.643	204.557,4	6.161,34	1

Tabela 6.2: Experimentos com o algoritmo de ordenação (tempo em ms)

Nós, podemos ver, neste experimento, que uma *tarefa* em migração diminui a velocidade de processamento da outra *tarefa* em no máximo 10%. Então, para o propósito de nossa simulação, não consideraremos a interferência de uma *tarefa* em migração sobre a execução da *tarefa* clonada.

6.1.4 Hipóteses de Simplificação

Como dito anteriormente, nosso objetivo é simular uma rede de máquina homogêneas e *tarefas* executando nelas, para assim podermos estudar os diferentes graus de *vivacidade*. Para tal, nós precisamos de um modelo para essa simulação. Obviamente, um modelo não intenciona ser um mapeamento perfeito da realidade; se assim fosse, sua implementação poderia tornar-se inviável. Baseados nisso, precisamos de algumas hipóteses de simplificação para ele.

Os experimentos já descritos, encaminharam-nos a duas hipóteses de simplificação para nosso modelo:

1. O custo de clonagem de uma *tarefa* é zero: Essa hipótese é derivada do fato que o custo de clonagem é insignificante comparado ao custo de migração (por volta de 200 vezes mais rápidos em nossos experimentos). Sendo assim, se o número de clonagens que ocorrem em um *servidor* não for muito elevado, o custo de clonagem é irrelevante. Isso pode ser facilmente explicado, já que o processo de migração depende de operações de E/S lentas por natureza, enquanto o processo de clonagem depende

de operações muito rápidas em memória RAM. Além do mais, como em nosso modelo uma clonagem está sempre associada a uma migração, em uma situação normal, o custo de clonagem poderia ser adicionado ao tempo de migração.

2. A migração de uma *tarefa* clone não interfere com a execução da *tarefa* clonada que executa no mesmo *servidor*: Essa simplificação é endossada pelo fato que a migração de uma *tarefa* clone não retarda significativamente a execução da *tarefa* clonada (em nossos experimentos, a sobrecarga foi de no máximo 10%). Assim, se o número de *tarefas* migrantes no *servidor* local não for muito elevado, nós podemos desconsiderar a sobrecarga.

Como já dito, essas hipóteses são válidas se o número de clonagens e migrações que ocorrem no *servidor* local não for muito elevado. Nesses termos, para limitar esses números, impomos duas restrições:

1. O modelo só admite uma *tarefa* por máquina: Essa é uma restrição razoável para nosso modelo, visto que as *tarefas* que executam no **MobiGrid** são *tarefas* com longo tempo de execução, trabalhando de forma independente com os dados nelas contidos; permitir mais de uma *tarefa* por máquina causaria uma perda de desempenho indesejável na vida real.

2. Se a *tarefa* que executa em uma máquina tiver sua clonagem pedida, ela, durante o período de criação e migração do clone, não atenderá outros pedidos de clonagem.

Nesse ponto, talvez possa ser levantada uma questão relevante: Se uma *tarefa* migrante não interfere de forma significativa com a execução de outras *tarefas* executando na mesma máquina, por que ela interferiria com os programas do usuário local, causando uma perda de desempenho? A resposta está na JVM. Mesmo não executando processamento pesado, a JVM faz uso significativo de recursos de memória, o qual pode afetar ao usuário local, violando um dos objetivos do **InteGrade** que é a transparência em termos de desempenho.

6.1.5 Modelo Matemático

Baseados nas hipóteses descritas acima, definimos um modelo matemático para nossa simulação, a qual trabalhará sobre o paradigma de uma simulação

discreta aleatória. Isso implica que o tempo assume somente valores discretos e que os eventos chave são gerados aleatoriamente (pseudo-aleatoriamente para sermos mais precisos). Os elementos principais desse modelo serão agora descritos:

1. *tarefa*: uma *tarefa* é descrita por um vetor $(x, y) \in \mathbb{Z}_+^* \times \mathbb{Z}_+^*$, onde x é o tempo de execução da *tarefa* executando isoladamente em uma máquina e y é o tempo de migração dessa *tarefa*.

2. *máquina*: a *máquina* representa o *servidor* onde a *tarefa* executará. Na simulação, não há distinção entre o *servidor* sendo morto pelo retorno do usuário local, ou por uma falha. Todas as *máquinas* tem probabilidades $(p_1, p_2) \in S \times S : S = \{x : x \in \mathbb{R}, 0 \leq x \leq 1\}$, onde p_1 é a probabilidade de uma *máquina* viva ser morta e p_2 é a probabilidade de uma máquina morta ser ligada de novo. Nós temos $m \in \mathbb{Z}_+^*$ *máquinas* disponíveis no começo da simulação. As *máquinas* são homogêneas

Para apresentar o algoritmo de simulação, utilizaremos o conceito de *grupo de tarefas*. Uma *grupo de tarefas* descreve as *tarefas* gêmeas. Então, um *grupo de tarefas* é um meio de agrupar as *tarefas* que são *gêmeas*. O algoritmo simplificado da simulação é:

Algoritmo 1 Algoritmo de simulação

```
    G ← {grupo de tarefas}
    D ← {} /* conjunto dos grupos de tarefas que tiveram todas suas tarefas mortas */
    F ← {} /*conjunto dos grupos de tarefas que terminaram a execução */
    para ∀e ∈ G faça
 5:   aloque uma máquina para a primeira tarefa de e
    num ← |G|, time ← 0
    enquanto |D| + |F| < num faça
      time++
      para cada máquina m faça
10:     gere um número pseudo-aleatório n ∈ [0; 1]
        se m está viva então
          se n ≤ p1 então
            mate m
        senão se n ≤ p2 então
15:       ligue m
      para ∀d ∈ G faça
        para cada tarefa t de d que não necessita ser clonada faça
          um passo de execução
        se todas tarefas de d necessitam ser clonadas então
20:       G ← G\{d}, D ← D ∪ {d}
          libere as máquinas das tarefas de d
        senão se d tem uma tarefa que terminou a execução então
          G ← G\{d}, F ← F ∪ {d}
          libere as máquinas das tarefas de d
25:     senão se time = 0 (mod b) então
          inicie a clonagem e migração das tarefas cujas máquinas morreram
        para cada tarefa migrante t faça
          se a máquina de origem ou destino t está morta então
            aloque outra máquina para t e reinicie o processo de migração
30:       senão
            execute um passo de migração para as tarefas migrantes.
```

Note que, na linha 25, iniciamos a operação de clonagem somente se for o instante de verificação por *tarefas* mortas. A procura por *tarefas* mortas

ocorre a cada b unidades de tempo. Esse parâmetro é definido antes do início da simulação. Na linha 26, podemos ver o uso de nossa hipótese 1, visto que que a operação de clonagem não acarreta incremento ao tempo. Na linha 31, podemos ver a nossa hipótese 2, pois o passo de migração é executado concorrentemente com o passo de execução da linha 18.

6.1.6 Implementação da Simulação

Nossa simulação foi implementada em **Java** versão 1.4.2. Daremos agora uma visão mais detalhada da implementação, mostrando sua estrutura de classes:

1. `Simulator`: Esta é a *façade* [GHJV00] de nosso simulador. O programador deve chamar o construtor desta classe fornecendo a lista de `Descriptors`, o número de `máquinas` para a simulação, as probabilidades p_1 e p_2 e o tempo de verificação b, bem como uma semente para o gerador de números pseudo-aleatórios. Para iniciar a simulação, uma invocação do método `start()` deve ser feita. Esse método devolve um objeto `Result`;

2. `Descriptor`: Esta classe representa o conceito de *grupos de tarefas* do modelo. Para criar um `Descriptor`, o programador deve fornecer ao construtor o *grau de vivacidade*, o tempo de execução e o tempo de migração das *tarefas*. O `Descriptor` é responsável por criar as `Tasks`. Ele possui um método chamado `time()`, o qual é usado no fim da simulação para informar o tempo em que uma das `Tasks` do `Descriptor` terminou sua execução, ou o tempo em que todas `Tasks` foram mortas;

3. `Task`: Representa o conceito de *tarefa* do modelo. Ela possui uma referência para a `Machine` que a hospeda;

4. `Machine`: Esta classe representa o conceito de *máquina* do modelo. Ela possui um método `kill()`, invocado pelo `Simulator` para matá-la;

5. `Result`: Esta classe é responsável por encapsular o resultados finais da simulação. Ela possui métodos `getTime()`, `getDeadMachines()`, `getDeadDescriptors()`, `getFinishedDescriptors()`, os quais, respectivamente, devolvem o tempo em que a simulação terminou, o número de *máquinas* que foram mortas, uma lista de `Descriptors` que tiveram

suas `Tasks` mortas, uma lista de `Descriptors` que tiveram pelo menos uma `Tarefa` que terminou sua execução;

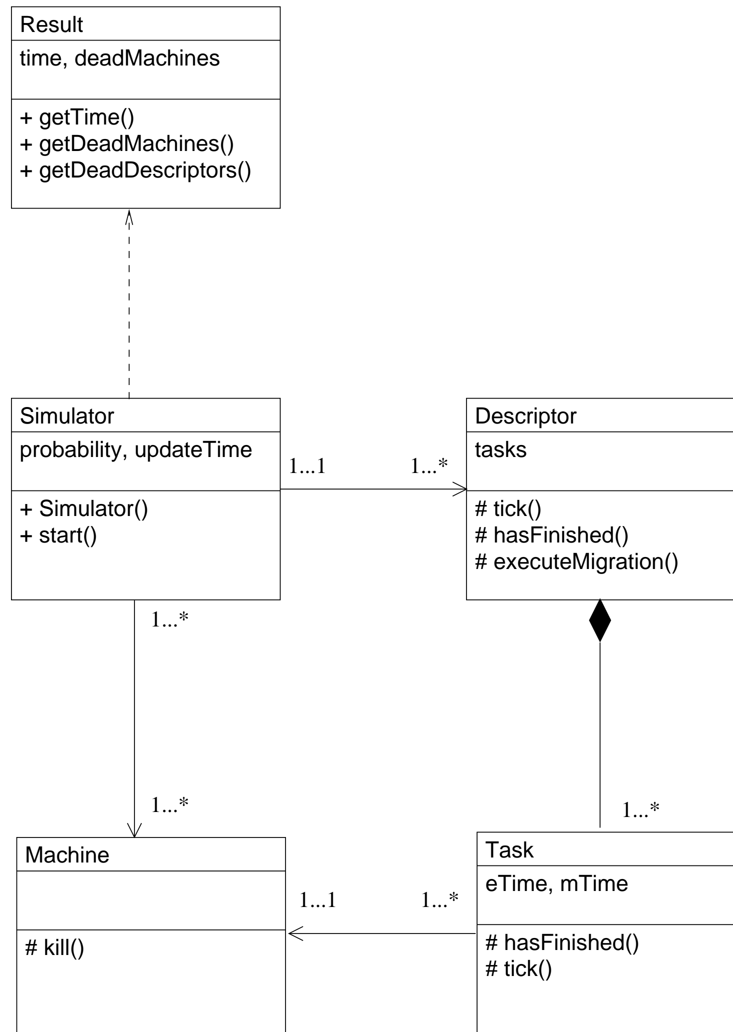

Figura 6.1: Diagrama de Classes UML da Implementação do Simulador

Na Figura 6.1, podemos ver melhor a relação entre as classes.

6.2 Experimentos com o Simulador

Definido o modelo matemático e com o código necessário à simulação pronto, realizamos alguns experimentos.

Para nossos experimentos, dividimos os parâmetros da simulação em dois grupos a saber:

Parâmetro	Valores Default
l	2
TM	5
TE	50
p_1	0,05
p_2	0,1
m	500

Tabela 6.3: Valores Default para os Parâmetros

1. parâmetros de *tarefas*: Definimos como parâmetros de *tarefas* aqueles que definem características que dependem unicamente das *tarefas*, dado que as características do ambiente são homogêneas e imutáveis. Em nossa simulação, são parâmetros de *tarefas*:

   (a) *vivacidade* (l);

   (b) o tempo de migração (TM);

   (c) o tempo de execução (TE).

2. parâmetros do ambiente: São parâmetros inerentes ao ambiente. Em nossa simulação, são parâmetros de ambiente:

   (a) probabilidade de uma dada máquina morrer por unidade de tempo (p_1);

   (b) probabilidade de uma dada *máquina* morta, ser ligada novamente por unidade de tempo (p_2);

   (c) número de *máquinas* (m).

Devido ao grande número de parâmetros, usamos a sistemática de lançar 100 *grupos de tarefas*, variando um parâmetro e fixando todos os demais em valores *default*. Nosso valores *default* são definidos na Tabela 6.3.

Para cada combinação de valores dos parâmetros, realizamos 1000 execuções de simulação. Registramos, então, o número médio do *grupos de tarefas* que conseguiram ter alguma *tarefa* que terminou sua execução (*grupos* terminados). Para esses *grupos* registramos o tempo médio de execução, ou seja, a média de quantas unidades de tempo a primeira *tarefa* que terminou sua execução utilizou para atingir tal estado.

6.2.1 Estudos dos Parâmetros de Tarefas

Estudo da Variação do Parâmetro l

Na Tabela 6.4 podemos visualizar os resultados de nosso experimentos.

l	tempo de execução	número de *grupos* que terminaram
1	50	7,69
2	74,47	26,75
3	78,29	37,92
4	85,48	46,91
5	86,45	51,12
6	88,8	53,99
7	88,62	51,28
8	86,81	49,57
9	86,05	47,59
10	86,53	45,86

Tabela 6.4: Experimentos com a Variação de l

Na Figura 6.2 representamos o gráfico com o tempo de execução dos *grupos de tarefas* que terminaram sua execução. Nesse gráfico, podemos observar que para $1 \leq l \leq 6$ há um significativo incremento no tempo de execução. Creditamos isso ao fato de que esses valores de l aumentaram também o número de *grupos de tarefas* que conseguiram terminar sua execução, conforme pode ser constatado no gráfico da Figura 6.3. Todavia, esse aumento dos *grupos* terminados teve como efeito colateral um número maior de migrações, as quais tiveram seus tempos refletidos no tempo total de execução.

Outro fato interessante a ser notado é que para $l > 6$, o número de *grupos* finalizados começa a diminuir de forma significativa. Acreditamos que esse fenômeno ocorra, pois com 10 grupos, cada um com 6 *tarefas*, teremos mais *tarefas* do que *máquinas*. Isso causa uma concorrência por *máquinas* livres entre as *tarefas* mais adiantadas e as mais atrasadas, o que aumenta a chance de que uma *tarefa* adiantada seja morta e não termine assim sua execução.

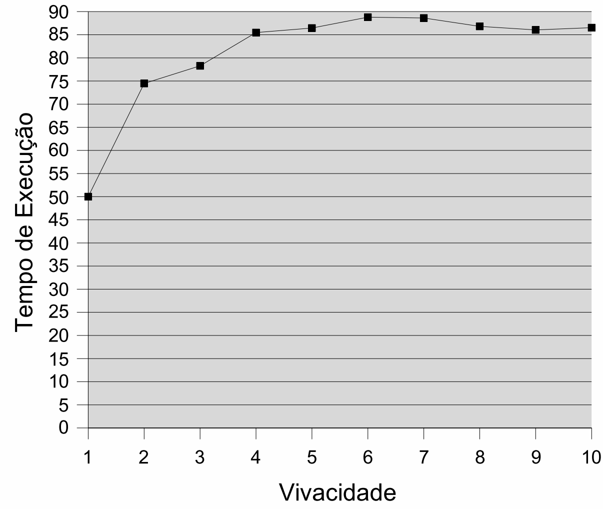

Figura 6.2: Variação de l e o tempo de execução

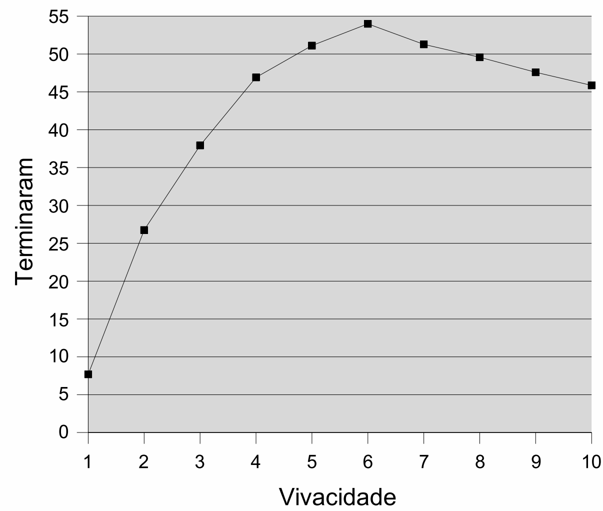

Figura 6.3: Variação de l e o número de *grupos de tarefas* que terminaram

Estudo da Variação do Parâmetro TM

Na Tabela 6.5 podemos visualizar os resultados de nosso experimentos com a variação de TM.

TM	tempo de execução	número de tarefas que terminaram
1	57,36	62,22
2	62,95	50,57
3	67,55	38,73
4	71,14	32,5
5	74,68	26,72
6	77,53	23,18
7	79,52	19,25
8	78,2	17,57
9	80,21	15,96
10	80,4	14,65
11	78,84	12,77
12	79,62	13,23
13	77,43	11,62
14	76,12	10,41
15	74,92	10,6

Tabela 6.5: Experimentos com a Variação de TM

Podemos observar no gráfico da Figura 6.4 que, para $1 \leq TM \leq 7$, o aumento do tempo de migração provoca também um aumento no tempo de execução dos *grupos*. Isso é facilmente explicável, já que durante a migração, a *tarefa* não está executando, o que ocasiona um atraso. No entanto, um fenômeno interessante acontece para $TM > 12$. A partir desse valor, o tempo de execução começa a diminuir. Esse fato é atribuído à metodologia do nosso experimento. De fato, estamos medindo somente o tempo de execução dos *grupos* que terminaram. Podemos observar no gráfico da Figura 6.5 que o incremento no tempo de migração provoca uma queda significativa no número de *grupos* que terminam de executar. Juntando essas duas informações, chegamos à conclusão que somente *grupos* muito "sortudos" conseguiram terminar de executar para valores de TM muito elevados. Para $TM = 13$, a média de *grupos* que terminaram é 11,62. Esses *grupos* tiveram "sorte" e foram poucos afetados pela morte de *máquinas*, necessitando realizar poucas migrações, o que diminui o impacto no tempo de execução.

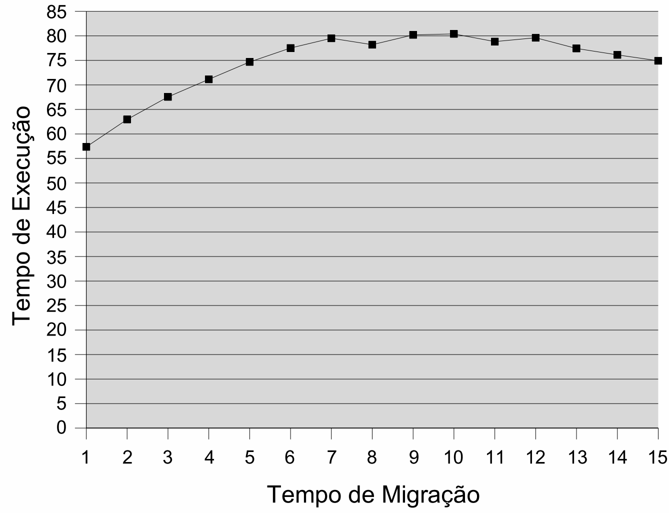

Figura 6.4: Variação de TM e o tempo de execução

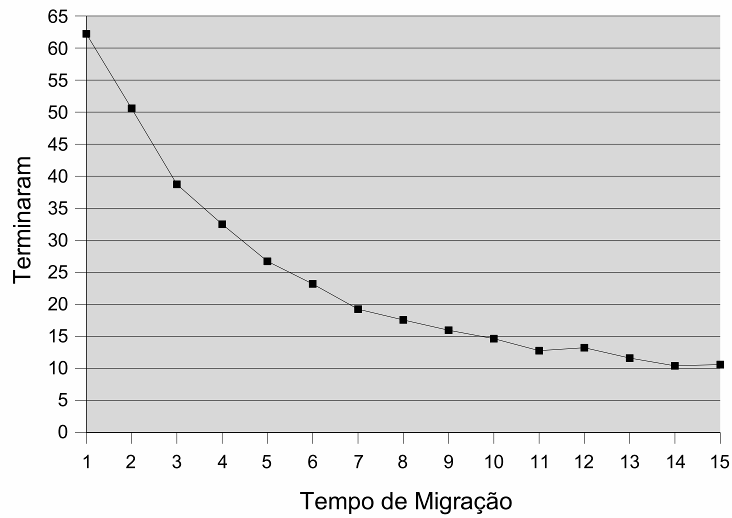

Figura 6.5: Variação de TM e o número de *grupos de tarefas* que terminaram

Estudo da Variação do Parâmetro TE

Na Tabela 6.6, podemos verificar os resultados de nossos experimentos com a variação do tempo de execução das *tarefas*.

TE	tempo de execução	número de tarefas que terminaram
10	16,57	66,92
20	33,64	53,35
30	49,61	42,09
40	62,87	33,53
50	74,32	26,29
60	86,5	21,39
70	100,17	17
80	109,78	13,63
90	122,22	10,72
100	130,05	7,93
110	142,18	7,06
120	152,06	5,5
130	165,7	4,33
140	171,21	3,6
150	182,53	2,48

Tabela 6.6: Experimentos com a Variação de TE

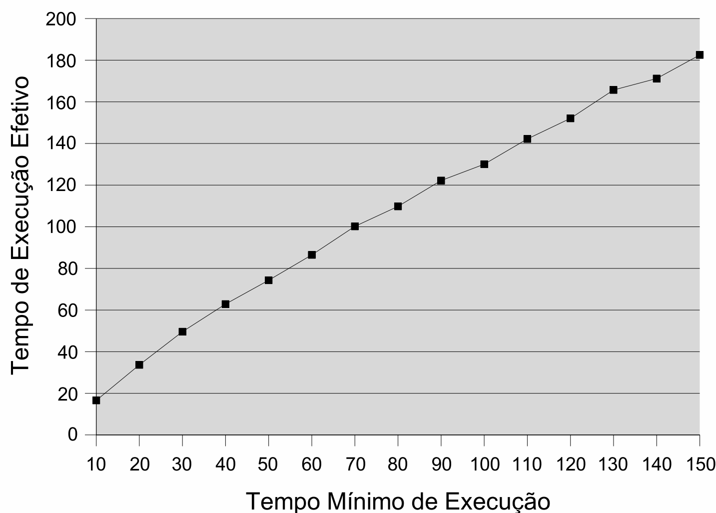

Figura 6.6: Variação de TE e o tempo de execução

O gráfico da Figura 6.6 mostra um crescimento linear no tempo de execução dos *grupos* com o aumento do tempo de execução das *tarefas*. O fato mais

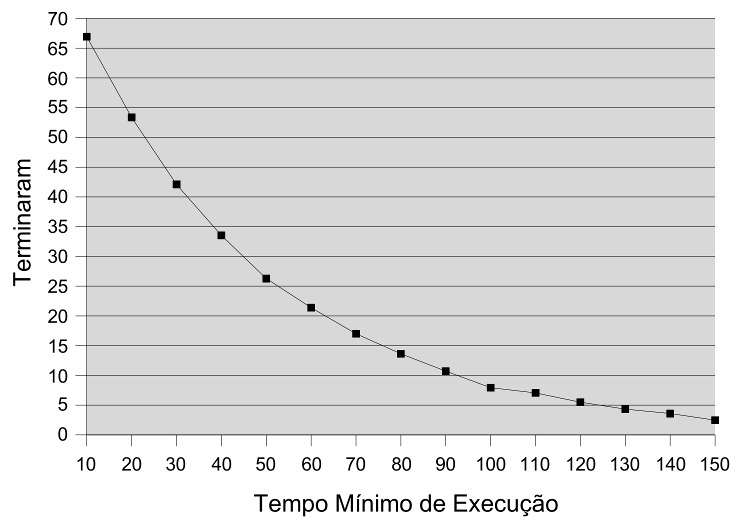

Figura 6.7: Variação de TE e o número de *grupos de tarefas* que terminaram

interessante fica por conta do gráfico da Figura 6.7. Podemos verificar um decrescimento exponencial no número de *grupos* terminados, com o aumento do tempo de execução. Isso é facilmente entendido, pois com o aumento do tempo de execução das *tarefas*, temos uma maior chance que todas as *tarefas* do *grupos* sejam mortas.

6.2.2 Estudos dos Parâmetros de Ambiente

Estudo da Variação de p_1

A Tabela 6.7 apresenta o resultado de nossos experimentos para a variação de p_1.

$P_1 = p_1.10^2$	tempo de execução	número de tarefas que terminaram
0	50	100
1	63,86	89,18
2	68,39	73,94
3	72,24	55,88
4	74,08	41,23
5	74,39	26,57
6	73,54	16,97
7	72,47	10,24
8	68,24	5,34
9	65,36	3,16
10	62,20	1,88

Tabela 6.7: Experimentos com a Variação de p_1

No gráfico da Figura 6.8, podemos observar que, para $0 \leq P_1 \leq 5$, há um aumento no tempo de execução. Esse fenômeno é fácil de ser explicado, pois à medida em que aumenta o valor de p_1, temos mais máquinas mortas. Conseqüentemente, há a necessidade de mais migrações para preservação da *vivacidade* dos *grupos*. Esse tempo de migração acaba causando um atraso na execução das *tarefas*, o que aumenta o tempo de execução do *grupo*. Estranhamente, observamos que para $P_1 > 5$, o tempo de execução começa a diminuir. Aqui acontece novamente o fenômeno já citado no estudo da variação do parâmetro TM. Valores muito altos de p_1 causam uma elevada mortandade de *tarefas*, como pode ser verificado no gráfico da Figura 6.9. Sendo assim, somente poucos grupos terminarão. Esses *grupos* são os mais "sortudos", sendo pouco afetados pela morte de máquinas, necessitando, assim, de poucas migrações.

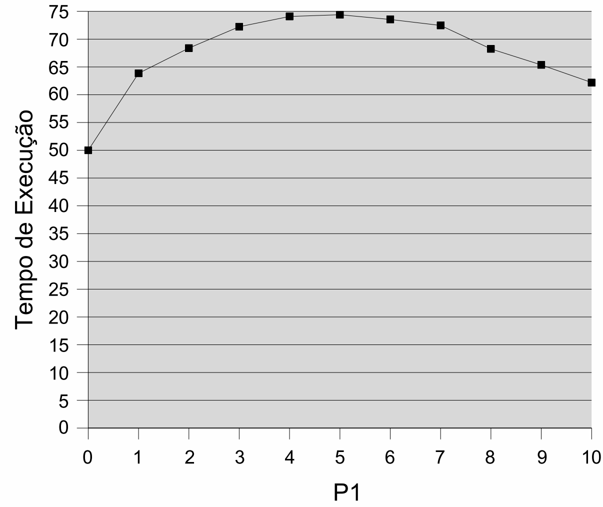

Figura 6.8: Variação de p_1 e o tempo de execução

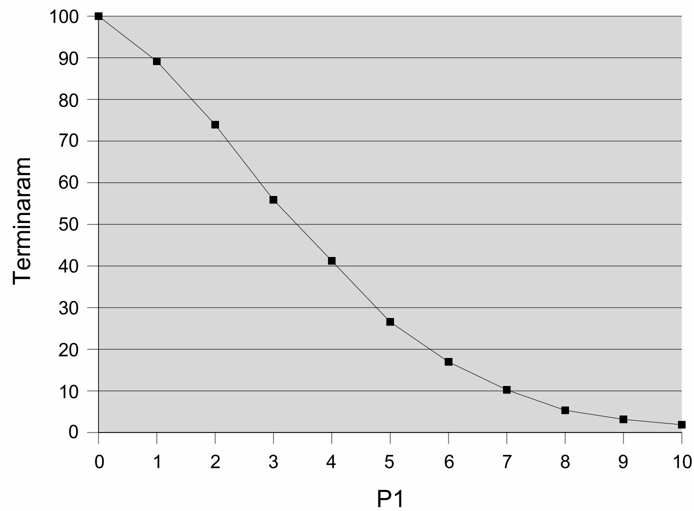

Figura 6.9: Variação de p_1 e o número de *grupos de tarefas* que terminaram

Estudo da Variação de p_2

Na Tabela 6.8 vemos os resultados dos experimentos para a variação do parâmetro p_2.

$P_2 = p_2.10^2$	tempo de execução	número de tarefas que terminaram
1	75,53	26,05
2	74,37	26,69
3	73,52	27,08
4	75,23	26,69
5	74,28	26,56
6	73,91	27,06
7	75,14	26,4
8	73,87	26,85
9	74,51	26,09
10	75,81	27,08

Tabela 6.8: Experimentos com a Variação de p_2

Podemos observar no gráfico da Figura 6.10 que o aumento de p_2 não influi de forma significativa no tempo de execução. Isso é esperado de certa forma, pois o número de *máquinas* disponíveis (500) é maior que o número total de máquinas utilizadas pelos *grupos* (200). Nesse contexto, maiores valores de p_2 só garantem um maior religamento, mantendo elevado o número de *máquinas* disponíveis. Sendo assim, altos valores de p_2 não estão contribuindo para a migração das *tarefas*; estão apenas mantendo as condições favoráveis. Com o mesmo argumento, podemos explicar também o gráfico da Figura 6.11. Esse gráfico nos mostra que maiores valores para p_2 não aumentaram o número de *grupos* finalizados.

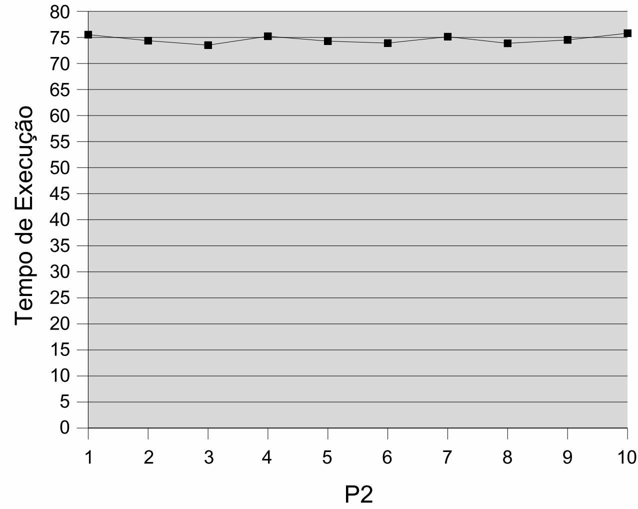

Figura 6.10: Variação de p_2 e o tempo de execução

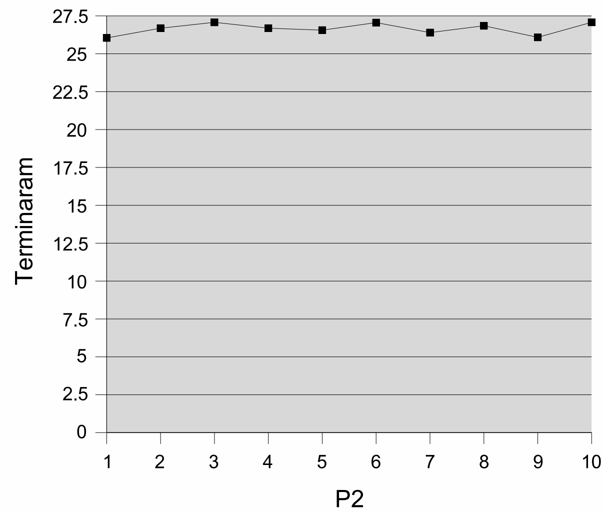

Figura 6.11: Variação de p_2 e o número de *grupos de tarefas* que terminaram

Estudo da Variação de m

A Tabela 6.9 mostra os resultados dos experimentos com a variação do parâmetro m.

m	tempo de execução	número de tarefas que terminaram
100	68,26	17,11
200	75,07	26,47
300	74,47	27,21
400	75,42	27,01
500	75,03	27,34
600	74,41	27,03

Tabela 6.9: Experimentos com a Variação de m

Podemos observar no gráfico da Figura 6.12 que, para $m = 100$ temos um tempo de execução ligeiramente inferior ao tempo de execução para valores $m > 100$. Mais uma vez, a explicação do *grupos* "sortudos" descrita anteriormente vale. Podemos ver no gráfico da Figura 6.13 que, para $m > 100$, há um significativo aumento dos *grupos* terminados (por volta de 50% a mais). Sendo assim, os *grupos* que conseguiram terminar para $m = 100$ são os menos afetados pelas mortes de *máquinas*. Portanto, realizaram poucas migrações, o que impactou pouco no tempo de execução.

Outro fenômeno interessante a se notar é o incremento de *grupos* terminados quando m vai de 100 para 200. Realmente, 100 é um número insuficiente de *máquinas* para todas as *tarefas*, já que possuímos 100 *grupos* cada um com 2 *tarefas*. Observamos também que, para $m > 200$ não há uma variação significativa no número de *grupos* terminados.

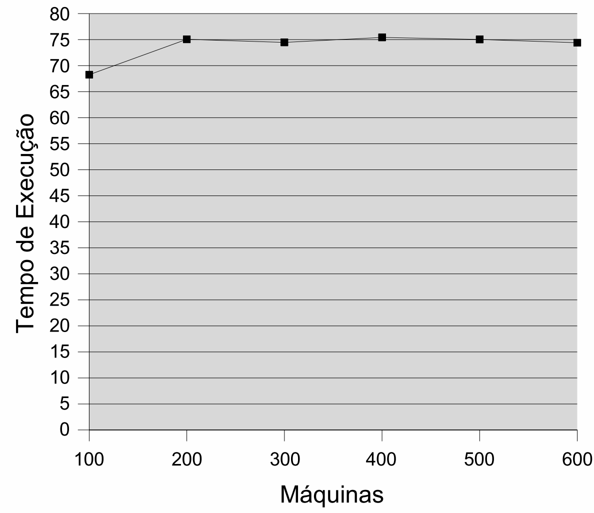

Figura 6.12: Variação de m e o tempo de execução

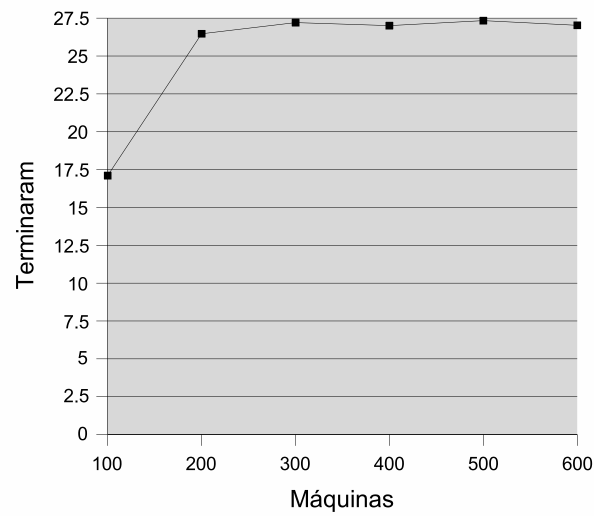

Figura 6.13: Variação de m e o número de *grupos de tarefas* que terminaram

6.3 Resumo dos Resultados da Simulação

Neste capítulo, definimos um modelo matemático para embasar uma simulação de uma grade computacional com grande quantidade de nós executando uma grande quantidade de *tarefas* submetidas a ela. Utilizamos a simulação para verificar os efeitos da variação de diversos parâmetros sobre o tempo de execução das *tarefas* e sobre número de *tarefas* terminadas. Para variarmos os parâmetros sistematicamente, nós o dividimos em duas categoria: Os parâmetros de *tarefas* e os parâmetros de ambiente.

Entre os parâmetros de *tarefas*, observamos que a escolha do valor de l deve ser cuidadosa, pois valores muito elevados podem piorar significativamente o número de *tarefas* terminadas, bem como o tempo de execução. De fato, uma regra simples a ser utilizada é: O valor de l não deve acarretar um número maior de *tarefas* que o número de *máquinas* disponíveis. Para os parâmetros TM e TE, podemos observar que altos valores contribuem fortemente para a diminuição do número de *tarefas* terminadas; quanto ao tempo de execução, como esperado, a variação de TE tem um impacto bastante importante, enquanto a variação de TM também aumenta o tempo de execução, visto que, durante a migração, a *tarefa* não executa.

Entre os parâmetros de ambiente, observamos que p_1 teve uma influência muito grande no tempo de execução e no número de *tarefas* terminadas. Em contrapartida, a influência de p_2 sobre essas medições foi praticamente nula. Quanto ao parâmetro m, podemos verificar que é importante que exista na *grade* mais *máquinas* do que *tarefas* submetidas, visto que um número reduzido de *máquinas* reduz de forma significativa o número de *tarefas* terminadas.

Capítulo 7

Trabalhos Futuros e Conclusão

7.1 Agentes Móveis *versus* Objetos Distribuídos

Desde os artigos mais pioneiros na área de agentes móveis [CHK95], discute-se a utilidade de tal paradigma. Inicialmente, tal discussão permanecia em torno da dicotomia agentes móveis e arquitetura cliente/servidor. Recentemente, a discussão recaiu sobre agentes móveis *versus* objetos distribuídos. De fato, com o avanço da tecnologia de objetos distribuídos, qual a utilidade de um agente móvel, quando um programa pode acessar recursos remotamente via **CORBA**, por exemplo? Para ficar mais claro, uma abordagem alternativa a agentes móveis poderia ser a utilização de objetos que acessariam os recursos localmente e os disponibilizariam para acesso remoto através de uma interface bem definida, com uso de tecnologias de objetos distribuídos. Não consideramos que as duas tecnologias sejam mutuamente excludentes, mas sim complementares, havendo situações em que uma tecnologia é mais indicada que a outra. Em nosso estudo, constatamos três situações importantes em que os agentes móveis são mais indicados do que objetos distribuídos:

1. Quando há a necessidade de uma resposta muito rápida, que não pode ser afetada pela latência da rede: Essa é uma situação bastante heterodoxa que pode acontecer em pesquisas científicas com auxílio de computação [FW03]. Um exemplo interessante seria a disponibilização remota de algum sensor - um telescópio - por exemplo. Com a utilização de agentes móveis, poderíamos enviar um agente a esses sensores programado para procurar por determinados padrões, bem como apto a realizar ajustes rapidamente. A utilização de agentes móveis em situações desse tipo

garante um alto desempenho, além de grande flexibilidade na utilização e controle do recurso.

2. Quando os recursos disponibilizados remotamente são tão dinâmicos que é impossível determinar uma interface que atenda todas as possíveis necessidades de um cliente: Um problema interessante é lidar com a imensa quantidade de dados gerados por um acelerador de partículas (ordem de centenas de terabytes) [FW03]. Nesses termos, o agente móvel tem a imensa vantagem de poder adaptar-se ao ambiente em que executa (diferenças nos dados coletados, bem como na forma de organizá-los e disponibilizá-los, por exemplo), podendo carregar consigo toda uma lógica de como alterar seu comportamento, ao mesmo tempo em que livra completamente o usuário que o despachou de ter de arcar com o custo de sua execução. Um mesmo agente móvel suficientemente flexível poderia migrar entre diversos servidores que armazenam dados de aceleradores, realizando algum tipo de pesquisa que necessite de diversas bases, por exemplo.

3. Utilização de poder computacional ocioso ou de melhor qualidade: Talvez seja a melhor situação para a utilização de agentes móveis frente aos objetos distribuídos. A utilização de agentes móveis permite que o programa seja encapsulado e levado em direção aos recursos computacionais, carregando consigo o estado de execução. Essa característica tem a imensa vantagem de permitir que a execução de um determinado programa seja realizada utilizando recursos de computadores diferentes, de uma forma oportunística que visa a aproveitar até pequenas fatias de tempo ocioso dessas máquinas. Essa foi a abordagem utilizada em nosso trabalho, visto que essa característica adequou-se perfeitamente ao ambiente de grade.

7.2 Agentes Móveis em Ambiente de Grade

Ao longo deste texto, apresentamos a nossa idéia de utilizarmos agentes móveis para encapsular tarefas seqüencias longas (*tarefas*) e despachá-las para nós ociosos da grade. Os agentes móveis foram utilizados para atender a dois requisitos do InteGrade:

1. A presença transparente do sistema;

2. Utilização melhorada de recursos ociosos.

A capacidade de migração do agente móvel atende ao primeiro requisito, pois permite ao sistema liberar a máquina para o usuário local, sem perder o processamento realizado. Além do mais, essa mesma capacidade permite ao agente poder fazer uso de fatias pequenas de tempo, de uma maneira mais oportunística, atendendo ao segundo requisito.

No início de nossa pesquisa, pensávamos em utilizar a capacidade de migração somente para o despacho das *tarefas* quando do retorno do usuário local à sua máquina. No entanto, percebemos que essa estratégia poderia causar algum impacto, pois a migração poderia tomar algum tempo e, durante esse período, a **JVM** permaneceria ligada, roubando recursos que deveriam estar disponíveis exclusivamente ao usuário. Constatamos, então, que a aliança da migração, clonagem e redundância poderia ser um recurso mais poderoso, visto que permite a morte imediata do agente móvel quando da volta do usuário local, com a conseqüente clonagem e migração de um agente gêmeo executando em outra máquina.

Como na época de nosso trabalho não possuíamos uma grade com grande quantidade de nós para efetuarmos experimentos, modelamos matematicamente nossa abordagem de uso de agentes móveis em grade e realizamos uma simulação, cujos resultados apresentamos em 6.2. Devido a nosso mudança de foco para a simulação, infelizmente o nosso código ainda não foi integrado ao InteGrade.

7.3 MobiGrid e MAG

Dentro do projeto InteGrade, conforme já citado em 2.1.2, além do **MobiGrid**, outra linha de pesquisa em agentes móveis foi criada e culminou com a criação do **MAG**. Comparando-se as duas plataformas, observamos algumas diferenças significativas. Ao passo em que o **MobiGrid** abriu mão de mecanismos de *checkpointing* para evitar sobrecarga na rede, tentando preservar o estado de execução de seus agentes através de redundância, o **MAG** utiliza um repositório central onde os agentes móveis salvam, de tempos em tempos, seus estados atuais de execução. Além disso, **MAG** utiliza o mecanismo de migração forte, ao passo que **MobiGrid** só possui migração fraca, tentando conseguir com isso mais rapidez na migração e evitando a sobrecarga que a manipulação de *bytecode* utilizada no **MAG** acarreta. Fundamentalmente,

MobiGrid optou por uma arquitetura mais simples, porém mais leve e liberal. Em contrapartida, **MAG** optou por uma arquitetura mais robusta que sofre com as sobrecargas da migração forte e do *checkpointing*.

Atualmente, o projeto **MAG** trabalha na implementação de um mecanismo de redundância similar ao do **MobiGrid**, visando, com isso, a oferecer mais flexibilidade ao usuário na recuperação de falhas.

7.4 Situação Atual e Trabalhos Futuros

Em nosso trabalho implementamos uma parte funcional do arcabouço. Nós temos implementadas as classes `AgletTask` e `AgletState`. Sendo assim, o usuário pode implementar *tarefas* que serão submetidas ao arcabouço. A preservação do estado atual da *tarefa* nos moldes descritos foi implementada, o que facilita muito a criação de `AgletTasks`. Temos também um `AgletManager` que cuida da migração e da vivacidade das *tarefas*, bem como um *servidor* que já implementa a funcionalidade de despacho.

Apesar de o código presente estar funcional, há muitos detalhes a serem implementados, bem como recursos que merecem atenção especial. Tivemos alguns problemas que derivaram da implementação e da documentação do **Aglets**. Alguns recursos presentes na documentação não estão, na verdade, implementados. Um problema que tivemos na implementação da *vivacidade* ocorreu graças ao fato de que o método `clone()` de `AgletProxy` não devolve um `AgletProxy` para o `Aglet` clonado, como afirma a documentação. Houve uma perda de tempo na detecção desse problema, bem como na descoberta de uma maneira de contorná-lo.

O código de nossa simulação, embora bastante flexível, ainda tem seu comportamento bastante amarrado no conceito de que, no caso de morte de todos os gêmeos, não há o relançamento de *tarefas*, contabilizando-se aquele *grupo* como morto simplesmente. Em nosso trabalho, surgiu a questão de se poder relançar as *tarefas* do *grupo* até que uma delas consiga terminar efetivamente seu trabalho, contabilizando, assim, somente o tempo de execução, abandonando-se o conceito de *grupos* não finalizados.

7.5 Resumo das Contribuições

Dentre as contribuições deste projeto de mestrado podemos citar:

- Criação de um arcabouço flexível para uso de agentes móveis em ambiente de grade;
- Criação de estratégias para preservação do estado de execução das *tarefas*;
- Definição do conceito de *vivacidade*, além de estratégias para sua preservação;
- Determinação de um modelo matemático para simulações de agentes móveis executando em ambiente de grade;
- Publicação como *full paper* de um artigo intitulado *MobiGrid - Framework for Mobile Agents on Computer Grid Environments* [BG04] na conferência internacional MATA 2004 - *International Workshop on Mobility Aware Technologies and Applications* - que ocorreu em outubro de 2004 em Florianópolis;
- Publicação como *short paper* de um artigo intitulado *A Study of Mobile Agents Liveness Properties on MobiGrid* [BGK05] na conferência MATA 2005, que ocorreu, desta vez, em Montreal, Canadá;
- Exibição de um pôster sobre o MobiGrid na conferência nacional WCGA 2005 [WCG07] - Workshop de Computação em Grid e Aplicações - que ocorreu em Petrópolis, no Laboratório Nacional de Computação Científica.

7.6 Conclusão

A idéia de utilizar agentes móveis no InteGrade é muito instigante e interessante. Nosso arcabouço, no futuro, pode ser usado para implementar muitos tipos de aplicações, desde aplicações embaraçosamente paralelizáveis, como o **SETI@home**, até aplicações que necessitem de muitos tipos de recursos não disponíveis em um único nó do InteGrade. Dada a característica oportunística dos agentes móveis, InteGrade caminhar rumo ao objetivo de ociosidade zero em seus nós, o que é impossível atualmente. Outro ponto interessante é nossa preocupação em construir um arcabouço que seja transparente ao usuário local da máquina, de forma que ele não enfrente uma queda de desempenho. Outro recurso forte do arcabouço é sua portabilidade, já que ele está sendo

escrito em **Java** em quase sua totalidade, exceção feita ao *daemon*, que deve ser implementado para cada plataforma que pretendamos utilizar.

No futuro, nosso arcabouço pode ser estendido para servir a uma infra-estrutura de grade que gerencia diferentes tipos de recursos disponíveis em várias máquinas. Dessa forma, as *tarefas* poderiam consultar o *gerente* procurando uma máquina ociosa que possuísse um recurso específico. O *gerente* obteria essa informação da infra-estrutura de grade e a repassaria à *tarefa*. Isso seria uma capacidade poderosa e útil. Como exemplo de recurso, tome bancos de dados, sensores específicos etc. Isso levaria a uma generalização da utilização de recursos, já que, por enquanto, nosso arcabouço está focado somente em recursos computacionais.

As possibilidades dos agentes móveis são inúmeras, e nosso arcabouço pretende fornecer uma camada básica sobre a qual um uso mais complexo dos agentes móveis pode ser alcançado.

Referências Bibliográficas

[AG04] IKV++ Technologies AG. Sítio da IKV++ Technologies, 2004. `http://www.ikv.de/`. Última visita em julho de 2004.

[AGK+01] Josef Altmann, Franz Gruber, Ludwig Klug, Wolfgang Stockner, and Edgar Weippl. Using Mobile Agents in Real World: A Survey and Evaluation of Agents Platforms. In *Proceedings of the 2nd International Workshop on Infrastructure for Agents, MAS, and Scalable MAS at the 5th International Conference on Autonomous Agents*, maio 2001. `http://www.umcs.maine.edu/~wagner/workshop/05_altmann_et_al.pdf` Última visita em maio de 2004.

[Agl04] Aglets. Site do Projeto Aglets, 2004. `http://aglets.sourceforge.net`. Última visita em fevereiro de 2004.

[AMMoV01] Rocco Aversa, Beniamino Di Martino, Nicola Mazzocca, and Salvat ore Venticinque. Mobile agents for distributed and dynamically balanced optimization a pplications. In *Proceedings of the 9th International Conference on High-Performan ce Computing and Networking*, pages 161–172. Springer-Verlag, 2001.

[BA04] BSP, BSP* and CGM Algorithms. Sítio com Coleção de Artigos sobre BSP/CGM, 2004. `http://www.scs.carleton.ca/~bsp/`. Última visita em setembro de 2004.

[BCPR05] Fabio Bellifemine, Giovanni Caire, Agostino Poggi, and Giovanni Rimassa. JADE - A White Paper. Technical report, Java Agent Development Framework, setembro 2005. `http://jade.tilab.com/papers/WhitePaperJADEEXP.pdf`. Última visita em maio de 2005.

[Bez06] Germano Capistrano Bezerra. *Análise de Conglomerados Aplicada ao Reconhecimento de Padrões de Uso de Recursos Computacionais - Dissertação de mestrado*. Universidade de São Paulo. Orientador: Marcelo Finger, 2006.

[BG04] Rodrigo M. Barbosa and Alfredo Goldman. MobiGrid - Framework for Mobile Agents on Computer Grid Environments. In *Mobility Aware Technologies and Applications (MATA 2004)*, pages 147–157. Springer-Verlag, 2004.

[BGK05] Rodrigo M. Barbosa, Alfredo Goldman, and Fabio Kon. A Study of Mobile Agents Liveness Properties on MobiGrid. In *Mobility Aware Technologies and Applications (MATA 2005)*, 2005. `http://www.congresbcu.com/mata2005/en/program.htm` e `http://integrade.incubadora.fapesp.br/portal/publications/files/papers/%barbosa-mata05.pdf`. Última visita em novembro de 2006.

[BOI04] BOINC. Sítio do Projeto BOINC, 2004. `http://boinc.berkeley.edu`. Última visita em fevereiro de 2004.

[Bou01] Sara Bouchenak. Making Java Applications Mobile or Persistent. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems*, january 2001. `http://citeseer.ist.psu.edu/bouchenak01making.html`. Última visita em maio de 2004.

[BTBCS04] Beowulf.org: The Beowulf Cluster Site. Sítio do Projeto Beowulf, 2004. `http://www.beowulf.org/`. Última visita em julho de 2004.

[CHK95] David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile Agents: Are They a Good Idea? Technical report, T.J. Watson Research Center, 1995.

[CMS04] Edson Cáceres, Henrique Mongelli, and Siang Song. BSP/CGM Algorithms (apesentação). In *Workshop Integrade*, janeiro 2004. `gsd.ime.usp.br/InteGrade/workshops/ws04/presentations/ws04BSPCGM.pdf`. Última visita em julho de 2004.

[Con04] Condor. Sítio do Projeto Condor, 2004. http://www.cs.wisc.edu/condor/. Última visita em fevereiro de 2004.

[Ecl06] Eclipse.org home. Sítio do Projeto Eclipse, 2006. http://www.eclipse.org/. Última visita em dezembro de 2006.

[Fer04] Luca Ferrari. *The Aglets 2.0.2 User's Manual*, outubro 2004. http://aglets.sourceforge.net/usermanual.html. Última visita em agosto de 2005.

[FK97] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputing Applications*, 2(11):115–128, 1997.

[FK99] Ian Foster and Karl Kesselman. *The Grid: Blueprint for a new Computing Structure*, chapter 2,3,5,11. Morgan Kaufmann Publishers, 1999.

[FW03] Geoffrey Fox and David Walker. e-Science Gap Analysis. Technical report, Indiana Univesity and Cardiff University, junho 2003.

[GHJV00] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Padrões de Projeto*. Bookman, 2000.

[GKG+04] Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger, and Germano Bezerra. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice and Experience*, 16:449–459, março 2004.

[Glo04] Globus. Sítio do Projeto Globus, 2004. http://www.globus.org. Última visita em fevereiro de 2004.

[Gmb01] IKV++ GmbH. *Grasshopper Basics And Concepts*, março 2001. http://grasshopper.ikv.de/cgi-bin/lnkinlte.cgi?l=FIKV200010110144. Última visita em julho de 2004.

[GQKG04] Andrei Goldchleger, Carlos Queiroz, Fabio Kon, and Alfredo Goldman. Running Highly-Coupled Parallel Applications in a

Comput ational Grid. In *Proceedings of 22th Brazilian Symposium on Computer Networks (SB RC'2004)*, 2004. http://gsd.ime.usp.br/publications/InteGradeBSPSBRC04.pdf. Última vis ita em agosto de 2004.

[GWF+94] Andrew S. Grimshaw, Willian A. Wulf, James C. French, Alfred C. Weaver, and Paul F. Reynolds Jr. A Synopsis of the Legion Project. Technical report, University of Virginia Computer Science Department, june 1994.

[Hom07] Home Page Universitá di Modena e Reggio Emilia. Sítio da Universidade de Modena e Reggio Emilia, 2007. http://www.eclipse.org/. Última visita em janeiro de 2007.

[IBM04] IBM. Sítio do Projeto Aglets na IBM, 2004. http://www.research.ibm.com/trl/aglets. Última visita em fevereiro de 2004.

[Int07] InteGrade. Sítio do Projeto InteGrade, 2007. http://integrade.incubadora.fapesp.br. Última visita em janeiro de 2007.

[IWKK00] Torsten Illmann, Michael Weber, Frank Kargl, and Tilmann Krüger. Migration in java: Problems, classification and solutions. In *Proceedings of International ICSC Symposium on Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce (MAMA'00)*, 2000. http://cia.informatik.uni-ulm.de/papers/mama00.pdf. Última visita em janeiro de 2005.

[JW83] Richard A. Johnson and Dean Wichern. *Applied Multivariate Statistical Analysis*. Prentice-Hall, 1983.

[Leg04] Legion. Sítio do Projeto Legion, 2004. http://www.cs.virginia.edu/~legion/. Última visita em fevereiro de 2004.

[LGPL05] GNU Lesser General Public License. Sítio do LGPL, 2005. http://www.gnu.org/copyleft/lesser.html. Última visita em agosto de 2005.

[LLM88] Michael Litzkow, Miron Livny, and Matt Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages pages 104–111, June 1988.

[LM95] Pat Langley and Michael B. Morgan, editors. *Elements of Machine Learning*. Morgan Kaufmann, 1995.

[Lop06] Rafael Fernandes Lopes. *MAG: uma grade computacional baseada em agentes móveis - Dissertação de mestrado*. Universidade Federal do Maranhão. Orientador: Francisco José da Silva e Silva, 2006.

[MBB+98] Dejan Milojicic, Markus Breugst, Ingo Busse, John Campbell, Stefan Covaci, Barry Friedman, Kazuya Kosaka, Danny Lange, Kouichi Ono, Mitsuru Oshima, Cynthia Tham, Sankar Virdhagriswaran, and Jim White. MASIF The OMG Mobile Agent System Interoperability Facility. In *Proceedings of the International Workshop on Mobile Agents (MA'98)*, setembro 1998. http://www.hpl.hp.com/personal/Dejan_Milojicic/ma4.pdf. Última visita em julho de 2005.

[MFHV98] Paul T. Murray, Roger A. Fleming, Paul D. Harry, and Paul A. Vickers. Somersault: Enabling Fault-Tolerant Distributed Software Systems. Technical report, HP Laboratories Bristol, April 1998. http://www.hpl.hp.com/techreports/98/HPL-98-81.pdf. Last visit on September, 2005.

[Mic04] Sun Microsystems. Sítio do Java Technology, 2004. http://java.sun.com. Última visita em agosto de 2004.

[Mic05] Sun Microsystems. Sítio da Sun Microsystems, 2005. http://www.sun.com. Última visita em julho de 2005.

[Mit97] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.

[MPIs04] The Message Passing Interface standard. Sítio do Padrão MPI, 2004. http://www-unix.mcs.anl.gov/mpi/. Última visita em setembro de 2004.

[OK97] Mitsuro Oshima and Guenter Karjoth. Aglets Specification 1.0. Technical report, IBM, maio 1997. http://www.research.ibm.com/trl/aglets/spec10.htm.

[OMG98] OMG. MASIF Revision. Technical report, Object Management Group, março 1998. http://www.omg.org/cgi-bin/doc?orbos/98-03-09.pdf. Última visita em julho de 2005.

[OMG04] OMG. Common Object Request Broker Architecture: Core Specification. Technical report, Object Management Group, março 2004. http://www.omg.org/technology/documents/formal/corba_iiop.htm. Última visita em julho de 2005.

[OMG05] Object Management Goup. Sítio da OMG, 2005. http://www.omg.org. Última visita em julho de 2005.

[OP04] Oxford Paralell. Sítio do Oxford BSP Toolset, 2004. http://www.bsp-worldwide.org/implmnts/oxtool/. Última visita em fevereiro de 2004.

[PC04] LAM/MPI Paralell Computing. Sítio do LAM/MPI, 2004. http://www.lam-mpi.org/. Última visita em julho de 2004.

[PMI04] MPICH A Portable MPI Implementation. Sítio do MPICH, 2004. http://www-unix.mcs.anl.gov/mpi/mpich/. Última visita em julho de 2004.

[Pro05] Log4J Project. Sítio do Log4J, 2005. http://logging.apache.org/log4j/docs/. Última visita em agosto de 2005.

[PVM04] PVM: Parallel Virtual Machine. Sítio do PVM, 2004. www.csm.ornl.gov/pvm/pvm_home.html. Última visita em julho de 2004.

[SET04] SETI@home. Sítio do Projeto SETI@home, 2004. http://setiathome.ssl.berkeley.edu/. Última visita em fevereiro de 2004.

[Sou05] SourceForge.net. Sítio do SourceForge.net, 2005. http://sourceforge.net/. Última visita em agosto de 2005.

[SSBS99] Thomas L. Sterling, John Salmon, Donald J. Becker, and Daniel F. Savarese. *How to build a Beowulf: a guide to the implementation and application of PC clusters.* MIT Press, 1999.

[TFfIPA05] The Foundation for Intelligent Physical Agents. Sítio do FIPA, 2005. `http://www.fipa.org/`. Última visita em agosto de 2005.

[TILHPI05] Telecom Italia Lab Home Page ITA. Sítio do TILAB, 2005. `http://www.tilab.com`. Última visita em agosto de 2005.

[WCG07] WCGA 2005. Sítio do III Workshop de Computação em Grid e Aplicações, 2007. `http://wcga05.lncc.br/`. Última visita em janeiro de 2007.

[Wor04] BSP Worldwide. BSP site, 2004. `http://www.bsp-worldwide.org/`. Última visita em fevereiro de 2004.

Nota:[1]

[1]Muitas das referências a sítios da Internet possuem datas até o final de 2004 como última visita. Isso ocorre, pois boa parte da pesquisa em Internet foi realizada nesse período.