Universidade Federal do Maranhão
Centro de Ciências Exatas e Tecnologia
Programa de Pós-graduação em Engenharia de Eletricidade

Rafael Fernandes Lopes

MAG: uma grade computacional baseada em agentes móveis

São Luís
2006

Rafael Fernandes Lopes

MAG: uma grade computacional baseada em agentes móveis

Dissertação apresentada ao Programa de Pós-graduação em Engenharia de Eletricidade da Universidade Federal do Maranhão, como requisito parcial para a obtenção do grau de MESTRE em Engenharia de Eletricidade.

Orientador: Francisco José da Silva e Silva

Doutor em Ciência da Computação – UFMA

São Luís

2006

Lopes, Rafael Fernandes

MAG: uma grade computacional baseada em agentes móveis / Rafael Fernandes Lopes. – São Luís, 2006.

124 f.

Dissertação (Mestrado) – Universidade Federal do Maranhão – Programa de Pós-graduação em Engenharia de Eletricidade.

Orientador: Francisco José da Silva e Silva.

1. Sistemas de Computação. 2. Grades Computacionais. 3. Agentes Móveis. I. Título.

CDU 004.75

Rafael Fernandes Lopes

MAG: uma grade computacional baseada em agentes móveis

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por Rafael Fernandes Lopes e aprovada pela comissão examinadora.

Aprovada em 13 de janeiro de 2006

BANCA EXAMINADORA

Francisco José da Silva e Silva (orientador)

Doutor em Ciência da Computação – UFMA

Marcelo Finger

Doutor em Ciência da Computação – IME/USP

Zair Abdelouahab

Doutor em Ciência da Computação – UFMA

À minha esposa Alionália e
ao meu filho Andrei

Resumo

Nos últimos anos, a computação em grade têm emergido como uma promissora alternativa para a integração e compartilhamento de recursos multi-institucionais. Entretanto, a construção de um *middleware* de grade é uma tarefa complexa. Desenvolvedores devem lidar com vários desafios de projeto e implementação, como: gerenciamento e alocação eficiente de recursos distribuídos, escalonamento dinâmico de tarefas, alta escalabilidade e heterogeneidade, tolerância a falhas, mecanismos eficientes para a comunicação colaborativa entre nós da grade e aspectos de segurança.

O MAG (Mobile Agents for Grid Computing Environments) explora a tecnologia de agentes móveis como uma forma de superar vários destes desafios. O *middleware* MAG executa as aplicações da grade carregando dinamicamente o código da aplicação no agente móvel. O agente do MAG pode ser realocado dinamicamente entre nós da grade através de um mecanismo de migração transparente chamado MAG/Brakes, como uma forma de prover balanceamento de carga e suporte para nós não dedicados. O *middleware* MAG também inclui mecanismos para prover tolerância a falhas de aplicações, uma característica essencial para ambientes de grade. O paradigma de agentes foi extensivamente utilizado para projetar e implementar os componentes do MAG, formando uma infraestrutura multiagente para grades computacionais. Esta dissertação de mestrado descreve a arquitetura, implementação e aspectos de desempenho do MAG e do MAG/Brakes.

Palavras-chaves: Grades computacionais. Agentes móveis. Migração forte. Tolerância a falhas.

Abstract

In recent years, Grid computing has emerged as a promising alternative to the integration and sharing of multi-institutional resources. However, constructing a Grid middleware is a complex task. Developers must address several design and implementation challenges, such as: efficient management and allocation of distributed resources, dynamic task scheduling, high scalability and heterogeneity, fault tolerance, efficient mechanisms for collaborative communication among Grid nodes, and security issues.

MAG (Mobile Agents for Grid Computing Environments) explores the mobile agent technology as a way to overcome several of these challenges. MAG middleware executes Grid applications by dynamically loading the application code into a mobile agent. The MAG agent can be dynamically reallocated among Grid nodes through a transparent migration mechanism called MAG/Brakes, as a way to provide load balancing and support for non-dedicated nodes. MAG middleware also includes mechanisms for providing application fault tolerance, an essential characteristic for Grid environments. We make extensive use of the agent paradigm to design and implement MAG components, forming a multi-agent infrastructure for computational Grids. This master thesis describes MAG and MAG/Brakes architecture, implementation and performance issues.

Key-words: Computational grids. Mobile agents. Strong migration. Fault-tolerance.

Agradecimentos

À Deus, que me guia e me ilumina. Aos meus pais, Edmilson e Engracia, pela confiança e carinho. À minha irmã e meus familiares que sempre torcem por mim, especialmente aos meus padrinhos Rafael Fernandes (*in memorian*) e Alda Maria Fernandes.

À minha esposa Alionália e ao meu filho Andrei, em reconhecimento aos muitos momentos roubados do nosso convívio durante a realização deste trabalho. Muito obrigado pela compreensão e apoio.

Ao meu orientador Francisco pela dedicação na orientação deste trabalho. Obrigado pelas experiências proporcionadas e pelas lições aprendidas: elas serão de extrema importância por toda a minha vida.

Aos professores Marcelo Finger e Zair Abdelouahab por terem aceito o convite para participação na minha banca de mestrado.

Aos membros do Laboratório de Sistemas Distribuídos: Bysmarck, Stanley, Gilberto, Eduardo e Estevão. Agradeço pelo incentivo e apoio prestados durante todo este trabalho. Também agradeço a todos os colegas do mestrado, em especial a Alisson, Emerson, Emanoel Claudino, Lindonete e Simone. O caminho foi árduo mas valeu a pena.

Aos amigos do Departamento de Informática: Carlos De Salles, Maria Auxiliadora e Mário Meireles. Obrigado por terem me ouvido a cada momento de desânimo e angústia durante esta jornada.

Ao professor Aristófanes Corrêa e à sua equipe por fornecerem a aplicação utilizada como referência nos testes de avaliação de desempenho realizados nesta dissertação.

Aos colegas do Núcleo de Tecnologia da Informação da UFMA pela compreensão e por acreditarem neste trabalho.

Ao CNPq pelo financiamento aos projetos Flexigrid e Integrade2, o que permitiu a existência da infraestrutura necessária para a realização deste trabalho. Por fim, a todos que direta ou indiretamente contribuiram para a elaboração deste trabalho.

"Há homens que lutam um dia e são bons.
Há outros que lutam um ano e são melhores.
Há os que lutam muitos anos e são muito bons.
Porém, há os que lutam toda a vida.
Estes são os imprescindíveis."

Bertolt Brech

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

As redes de computadores existentes em instituições tanto públicas quanto privadas formam hoje um enorme parque computacional interconectado principalmente por tecnologias ligadas à Internet. No entanto, apesar de permitir comunicação e troca de informações entre computadores, as tecnologias que compõem a Internet atual não disponibilizam abordagens integradas que permitam o uso coordenado de recursos pertencentes a várias instituições na realização de computações. Uma nova abordagem, denominada computação em grade (*grid computing*) [FK99, Buy02], tem sido desenvolvida com o objetivo de superar esta limitação.

Grade, em um nível conceitual, é um tipo de sistema paralelo e distribuído que possibilita o compartilhamento, seleção, e agregação de recursos autônomos geograficamente distribuídos em tempo de execução, dependendo de sua disponibilidade, capacidade, desempenho, custo, e requisitos de qualidade de serviço de usuários [Buy02].

A computação em grade permite a integração e o compartilhamento de computadores e recursos computacionais, como software, dados e periféricos, em redes corporativas e entre estas redes, estimulando a cooperação entre usuários e organizações, criando ambientes dinâmicos e multi-institucionais, fornecendo e utilizando os recursos de maneira a atingir objetivos comuns e individuais [BBL00, FKT01]. Existem atualmente muitas aplicações para a computação em grade. Ian Foster e Carl Kesselman [FK99] identificaram as cinco principais classes de aplicações para grades de computadores: supercomputação distribuída, alto rendimento, computação sob demanda, computação intensiva de dados e computação colaborativa. Estas classes de aplicações são apresentadas na Tabela 1.1.

Krauter et al. [KBM02] criaram uma taxonomia dos possíveis tipos de grades e descreveram quais classes de aplicações são suportadas por cada tipo. Esta taxonomia é ilustrada na Figura 1.1.

Grades computacionais são aquelas cujo foco principal é agregar capacidade computacional em um único sistema. As classes de aplicações cobertas por este tipo de grade são as de supercomputação distribuída e de alto rendimento. *Grades de dados* têm por objetivo prover uma infra-estrutura para sintetizar informações que estão distribuídas

Categoria	Características	Exemplos
Supercomputação distribuída	Grandes problemas com intensiva necessidade de CPU, memória, etc.	Simulações interativas distribuídas, cosmologia, modelagem climática
Alto rendimento	Agregar recursos ociosos para aumentar a capacidade de processamento. A grade é utilizada para executar uma grande quantidade de tarefas independentes ou fracamente acopladas.	Projeto de chips, problemas de criptografia, simulações moleculares
Computação sob demanda	Recursos remotos integrados em computações locais, muitas vezes por um período de tempo limitado	Instrumentação médica, processamento de imagens microscópicas
Computação intensiva de dados	Síntese de novas informações de muitas ou grandes origens de dados	Experimentos de alta energia, modernos sistemas meteorológicos
Computação colaborativa	Suporte à comunicação ou a trabalhos colaborativos entre vários participantes	Educação, projetos colaborativos

Tabela 1.1: Cinco maiores classes de aplicações de grades

Figura 1.1: Taxonomia dos sistemas de grade

em uma rede de computadores. Elas disponibilizam mecanismos para o armazenamento e gerenciamento das informações, além de serviços como a mineração de dados. Estão englobadas nesta categoria as grades que tratam de aplicações de computação intensiva de dados. *Grades de serviços* foram criadas para prover serviços que não poderiam ser fornecidos por uma única máquina. Esta classificação é dada às grades que oferecem serviços de computação sob demanda e computação colaborativa.

1.1 Requisitos

Para que a computação em grade se torne realidade, um *middleware* que atenda a diversos requisitos deve ser desenvolvido, entre os quais se destacam:

- Gerenciamento e alocação eficiente de recursos distribuídos;
- Escalonamento dinâmico de tarefas de acordo com a disponibilidade de recursos distribuídos. A alocação de tarefas não deve ser estática, dado que deve ser levada em consideração a natureza dinâmica dos ambientes de grade. Isto requer mobilidade transparente de código, como uma forma de promover o balanceamento de carga e o suporte para nós não dedicados;
- O escalonador de tarefas pode levar em consideração o histórico de uso dos nós da grade, como uma forma de prever a disponibilidade de recursos, melhorando a alocação de recursos e minimizando o custo de realocação das aplicações [GKGF03];
- Mecanismos de tolerância a falhas. Nós da grade são instáveis, dado que em geral eles não são dedicados à execução de aplicações da grade e não compreendem um ambiente controlado, ao contrário do que ocorre com máquinas paralelas e *clusters* de computadores. A probabilidade da ocorrência de falhas é aumentada pelo fato de que muitas aplicações da grade executam longas tarefas que podem requerer vários dias de computação;
- Suporte a alta escalabilidade e grande heterogeneidade de componentes de hardware e software;
- Mecanismos eficientes para comunicação colaborativa entre os nós da grade;
- A disponibilização de mecanismos de proteção e segurança. O *middleware* de grade deve autenticar o código das aplicações baixados para os nós da grade, prover canais de comunicação seguros entre os componentes do *middleware* e fornecer mecanismos de controle de acesso aos recursos distribuídos. Algumas aplicações podem também requisitar a criação de *logs* e a auditoria de uso dos recursos. [Sta02]

1.2 Objetivo

Agentes móveis exibem grande adequação para o desenvolvimento de infraestruturas de grade, por causa de algumas de suas características intrínsecas, como capacidade de cooperação, autonomia, heterogeneidade, reatividade e mobilidade. Estas e outras características podem ser exploradas por projetistas e implementadores de *middlewares* de grade para transpor os diversos desafios relativos ao seu desenvolvimento.

Esta dissertação de mestrado tem como principal objetivo explorar o uso da tecnologia de agentes móveis para o desenvolvimento de um *middleware* denominado MAG (*Mobile Agents Technology for Grid Computing Environments*) através do qual seja possível a resolução de problemas computacionalmente intensivos em grades de computadores.

1.3 Estrutura da Dissertação

Esta dissertação descreve a arquitetura do MAG, sua implementação e avaliação de desempenho. Ela está organizada como a seguir: o capítulo 2 descreve a arquitetura geral do *middleware* MAG. O capítulo 3 apresenta o mecanismo de migração transparente de aplicações desenvolvido para o MAG, enquanto que o capítulo 4 mostra alguns resultados obtidos através da análise de desempenho do *middleware*. O capítulo 5 discute relevantes trabalhos relacionados e o capítulo 6 apresenta conclusões obtidas a partir deste trabalho além de descrever os trabalhos futuros que podem ser desenvolvidos a partir deste esforço inicial.

2 Arquitetura geral do MAG

A tecnologia de agentes móveis exibe grande adequação para o desenvolvimento de infraestruturas de grade, por causa de algumas de suas características intrínsecas, como capacidade de cooperação, autonomia, heterogeneidade, reatividade e mobilidade. Estas e outras características podem ser exploradas por projetistas e implementadores de *middlewares* de grade para transpor os diversos desafios relativos ao seu desenvolvimento.

Este capítulo descreve a arquitetura e a implementação do MAG (*Mobile Agents Technology for Grid Computing Environments*) [MAG, LSS05], uma infraestrutura de software livre[1] baseada na tecnologia de agentes móveis que permite o reaproveitamento de recursos ociosos existentes em instituições para a resolução de problemas computacionalmente intensivos.

2.1 Camadas de Software do *middleware* MAG

O projeto MAG (*Mobile Agents Technology for Grid Computing Environments*) explora a tecnologia de agentes móveis como uma forma de superar os desafios de projeto e implementação relativos ao desenvolvimento de um *middleware* de grade. É um projeto desenvolvido por alunos e professores do Departamento de Informática da Universidade Federal do Maranhão (DEINF/UFMA), cujo principal objetivo é criar uma infraestrutura de software livre baseada em agentes móveis que permita a resolução de problemas computacionalmente intensivos em grades computacionais já existentes. O projeto MAG recebe apoio financeiro do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) como parte do projeto FlexiGrid (aprovado em parceria com a Universidade Federal de Goiás) e do projeto Integrade 2 (aprovado em parceria com a Universidade de São Paulo, a Pontifícia Universidade Católica do Rio de Janeiro, a Universidade Federal do Mato Grosso do Sul e a Universidade Federal de Goiás).

O MAG executa aplicações na grade carregando dinamicamente o código da

[1] O MAG segue a licença GPL (*GNU General Public License*). Esta licença pode ser acessada em: `http://www.gnu.org/copyleft/gpl.html`)

aplicação no agente móvel. Os agentes do MAG podem ser realocados dinamicamente entre os nós da grade através de um mecanismo de migração transparente de aplicações. Hoje este mecanismo é utilizado apenas para prover suporte a nós não dedicados. Além de utilizar as capacidades dos agentes para a execução de aplicações, o MAG também propõe o uso de agentes para a construção dos componentes que formam a infraestrutura da grade.

O uso da tecnologia de agentes móveis se deu pela grande adequação deste paradigma ao desenvolvimento de *middlewares* de grade. Esta adequação deve-se às várias características intrínsecas aos agentes de software, como:

- *Cooperação:* agentes têm a habilidade de interagir e cooperar com outros agentes. Isto pode ser explorado para o desenvolvimento de mecanismos de comunicação complexos entre os nós da grade;
- *Autonomia:* agentes são entidades autônomas, significando que suas execuções ocorrem sem nenhuma, ou com muito pouca intervenção dos clientes que os iniciaram. Este é um modelo adequado para a submissão e a execução de aplicações na grade;
- *Heterogeneidade:* muitas plataformas de agentes móveis podem ser executadas em ambientes heterogêneos, uma característica importante para um melhor uso dos recursos computacionais entre ambientes multi-institucionais;
- *Reatividade:* agentes podem reagir a eventos externos, como variações na disponibilidade de recursos;
- *Mobilidade:* agentes móveis podem migrar de um nó para outro, movendo parte da computação sendo executada e provendo balanceamento de carga entre os nós da grade;
- *Proteção e segurança:* muitas plataformas de agentes oferecem mecanismos de proteção e segurança que podem ser reaproveitados pela infraestrutura da grade. Entre estes mecanismos encontramos os de autenticação, criptografia e controle de acesso.

A arquitetura do MAG está organizada em forma de uma pilha de camadas, conforme pode ser visto na Figura 2.1. Atualmente no MAG é possível executar duas

classes distintas de aplicações: regulares e paramétricas. As aplicações regulares são aplicações seqüenciais compostas de um binário que executa em uma única máquina. Já as aplicações paramétricas são aquelas que executam múltiplas cópias do mesmo binário em máquinas distintas com entradas diferentes. Esta classe de aplicações (também chamada de BoT ou *Bag-of-Tasks*) divide as tarefas em sub-tarefas menores que executam de forma independente, sem haver comunicação entre elas. Várias aplicações enquadram-se nesta categoria, como as de processamento de imagens, simulação e mineração de dados. Aplicações escritas na linguagem Java são executadas pelo MAG, enquanto que aplicações nativas do sistema operacional são executadas diretamente pelo *middleware* Integrade.

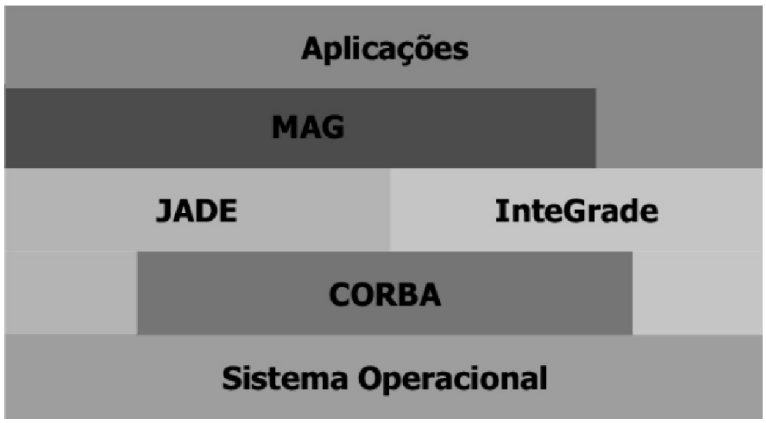


Figura 2.1: Arquitetura em camadas do *middleware* MAG

O MAG utiliza o *middleware* Integrade [GKGF03, GKG+04, CGKG04, Gol04] como fundação para sua implementação, evitando assim, a duplicação de esforços no desenvolvimento de uma série de componentes. O MAG adiciona ao Integrade um novo mecanismo de execução de aplicações, através do uso da tecnologia de agentes móveis, para permitir a execução de aplicações Java, não suportadas nativamente pelo Integrade. Além disso, o MAG propõe também a criação de mecanismos de tolerância a falhas de aplicações, migração transparente (para prover suporte a nós não dedicados), além de permitir o acesso de clientes móveis e nômades à grade (através de dispositivos de computação portátil e da web).

A camada JADE (*Java Agent Development Framework*) é um arcabouço utilizado para a construção de sistemas multiagentes. Ela provê ao MAG várias funcionalidades como facilidades de comunicação, controle do ciclo de vida e monitoração da execução dos agentes móveis. O JADE é uma plataforma portável (pois foi desenvolvida utilizando tecnologia Java) e aderente à especificação FIPA[2] [FIP].

[2] *Foundation for Intelligent Physical Agents* – organização sem fins lucrativos que objetiva a produção

As camadas superiores utilizam a tecnologia de objetos distribuídos CORBA para promover a comunicação entre seus componentes. Além disso, esta tecnologia fornece diversos serviços que podem ser utilizados pela infraestrutura do MAG como, por exemplo, o serviço de negócios (*trading*) [Obj00]. Por último, a camada de sistema operacional pode ser variável, dado que o MAG é independente de plataforma (pois utiliza tecnologia Java).

2.2 Integrade

Integrade é um *middleware* de grade desenvolvido pelo Instituto de Matemática e Estatística da Universidade de São Paulo (IME/USP), o Departamento de Computação e Estatística da Universidade Federal de Mato Grosso do Sul (DCT/UFMS) e o Departamento de Informática da Pontifícia Universidade Católica do Rio de Janeiro (DI/PUC-Rio). Os principais objetivos do projeto Integrade são [GKG+04]:

- Utilizar o poder computacional ocioso de máquinas disponíveis em instituições para resolver problemas computacionais;
- Permitir o acesso remoto a hardware e software de outras máquinas, como processadores, discos e compiladores;
- Permitir o compartilhamento de recursos computacionais, evitando a necessidade, por exemplo, de adquirir novo hardware para utilizar o sistema;
- Ser escalável e facilmente gerenciado através de uma estrutura hierárquica.

O Integrade é estruturado em aglomerados, ou seja, unidades autônomas dentro da grade que contém todos os componentes necessários para funcionar independentemente. Estes aglomerados podem estar organizados de uma forma hierárquica ou conectados através de uma rede ponto-a-ponto (*Peer-to-Peer* ou P2P) [Gol04]. A arquitetura de um aglomerado Integrade pode ser visto na Figura 2.2.

O nó "Gerenciador do Aglomerado" representa o nó responsável por gerenciar o aglomerado e por comunicar-se com os outros gerenciadores de aglomerados. Um "Nó de Usuário" é um nó pertencente a um usuário que submete aplicações para execução na grade. Um "Nó Compartilhado" é tipicamente um PC ou estação de trabalho em um

de padrões de interoperabilidade entre agentes de software heterogêneos.

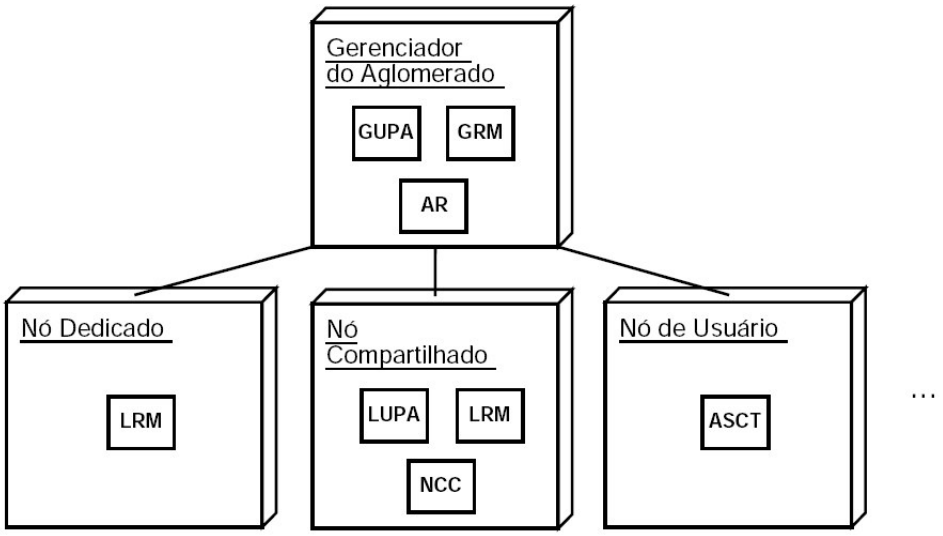


Figura 2.2: Arquitetura de um aglomerado Integrade

laboratório compartilhado que exporta parte de seus recursos, tornando-os disponíveis aos usuários da grade. Um "Nó Dedicado" é aquele reservado à execução de computações da grade. Estas categorias podem sobrepor-se: um nó pode ser ao mesmo tempo um "Nó de Usuário" e um "Nó Compartilhado". Como também pode ser visto na Figura 2.2, varias entidades de software compõem a arquitetura mostrada. Elas executam nos nós componentes do aglomerado de forma a prover as funcionalidades necessárias a cada categoria de nó. Estas entidades serão descritas a seguir:

- *GRM (Global Resource Manager):* gerenciador de recursos do aglomerado. Mantêm informações a respeito do estado de disponibilidade dos recursos presentes nas máquinas que compõem o aglomerado, efetuando o escalonamento de tarefas nos nós de acordo com estas informações e efetuando negociação e reserva de recursos. Executa no nó gerenciador do aglomerado;

- *GUPA (Global Usage Pattern Analyser):* executa em cooperação com o GRM, gerenciando informações sobre os padrões de uso de recursos dos nós componentes do aglomerado. Fornece estas informações ao GRM de forma a colaborar para que ele tome melhores decisões de escalonamento;

- *AR (ApplicationRepository):* armazena as aplicações que podem executar na grade. Toda aplicação submetida à execução deve ser primeiramente registrada neste repositório;

- *LRM (Local Resource Manager):* é executado em cada nó que fornece recursos ao

aglomerado. Coleta informações sobre o estado do nó como memória, CPU, disco e utilização da rede. Os LRMs enviam periodicamente informações ao GRM de seus aglomerados, através de um "protocolo de atualização de informações";

- *LUPA (Local Usage Pattern Analyser):* executa junto ao LRM e coleta localmente informações de padrões de uso dos usuários do nó. Baseia-se em longas séries de dados derivadas de padrões semanais de uso do nó. Periodicamente transmite as informações locais ao GUPA;

- *NCC (Node Control Center):* este componente permite aos usuários de máquinas provedoras de recursos definir condições para o compartilhamento dos mesmos. Parâmetros como períodos de tempo em que ele não deseje compartilhar seus recursos, a quantidade de recursos a serem utilizadas por aplicações da grade (por exemplo, 30% de CPU e 50% de memória física) ou outras definições relativas à máquina podem ser feitas através desta ferramenta;

- *ASCT (Application Submission and Control Tool):* ferramenta para submissão aplicações para execução na grade. Os usuários podem especificar pré-requisitos de execução, como a plataforma de hardware e software nas quais a aplicação deve ser executada, requisitos de memória necessários à execução da aplicação, entre outros. Também é possível monitorar o progresso da aplicação em execução.

O Integrade adota o modelo BSP (*Bulk Synchronous Parallelism*) [Val90] para a execução de computações paralelas, usando a biblioteca BSPLib [HMS+98] da Universidade de Oxford como referência.

2.3 Arquitetura

Por utilizar a tecnologia de agentes em sua concepção, a modelagem do *middleware* MAG não poderia utilizar as técnicas tradicionais de desenvolvimento de software, dado que as mesmas não fornecem recursos suficientes para a representação de diversos aspectos relativos a sistemas baseados em agentes como, por exemplo, a modelagem de ontologias, necessária para definir o conhecimento dos agentes a respeito do domínio da aplicação.

Após analisar algumas metodologias para o desenvolvimento de sistemas multiagentes, optou-se por utilizar a *Agil PASSI* [CCSS04]. A *Agil PASSI* é uma adaptação da metodologia PASSI (*Process for Agents Societies Specification and Implementation*) [CSSC03, CSS03] original. Ela foi concebida para permitir o desenvolvimento de sistemas multiagentes através de um processo ágil [AGIa, AGIb].

Para a modelagem do MAG, entretanto, foi necessário adaptar alguns aspectos da *Agil PASSI*, dado que esta metodologia foi concebida para a modelagem de sistemas baseados integralmente em agentes de software, o que não acontece no projeto MAG, que reutiliza componentes de software da arquitetura do Integrade.

A metodologia *Agil PASSI* é composta fundamentalmente por 4 modelos [CCSS04]:

- *Requisitos:* modelo dos requisitos envolvidos no desenvolvimento do sistema;
- *Sociedade de agentes:* uma visão dos agentes envolvidos na solução, suas interações e seu conhecimento sobre o sistema;
- *Codificação:* modelos que descrevem como o sistema deverá ser codificado;
- *Testes:* planejados antes da fase de codificação e executados logo após ela.

2.3.1 Modelo de Requisitos do MAG

Este modelo é composto por duas atividades: *planejamento* e *descrição de requisitos de domínio* (*Domain Requirements Description* ou DRD). A etapa de planejamento é utilizada para efetuar o planejamento do desenvolvimento do projeto. É nesta atividade que são efetuadas a análise de riscos, a divisão das tarefas em sub-tarefas e o planejamento das interações entre os desenvolvedores durante a implementação do sistema. Já na atividade de descrição de requisitos de domínio são utilizados diagramas UML de casos de uso para representar a descrição funcional do sistema. Neste diagrama, o relacionamento entre os casos de uso pode seguir basicamente três estereótipos: (a) *include*, que indica que o comportamento de um caso de uso está inserido dentro de outro, evitando a criação de casos de uso que contenham funcionalidades semelhantes; (b) *extend*, que indica que o comportamento de um caso de uso é estendido por outro que lhe adiciona novas funcionalidades; e (c) *communicate* que é utilizado na relação entre os atores e o sistema, e na comunicação entre funcionalidades distintas da aplicação.

O MAG é um sistema complexo. Para facilitar o entendimento deste diagrama, ele foi dividido em 6 cenários: *registro de aplicações*, *execução de aplicações*, *liberação de nós*, *tolerância a falhas*, *atualização de informações de nós* e *visualização de aplicações em execução na grade*. Os casos de uso de um cenário não são exclusivos, podendo aparecer em diversos diagramas.

Registro de aplicações

No MAG, para permitir que uma aplicação seja executada na grade, é necessário que a mesma esteja previamente armazenada em um repositório de aplicações. Este processo pode ser efetuado por qualquer usuário com permissão para utilizar a grade. A Figura 2.3 mostra os casos de uso realizados neste processo.

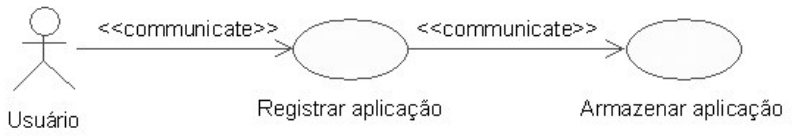


Figura 2.3: Cenário "Registro de aplicações"

Este cenário é muito simples e de fácil entendimento: quando o usuário da grade requisita o registro de uma aplicação (*Registrar aplicação*), o seu binário é transmitido através da rede para ser efetivamente armazenado no repositório de aplicações (*Armazenar aplicação*).

Execução de aplicações

Este cenário, apresentado na Figura 2.4, é o mais complexo de todos, dado que engloba diversos casos de uso. Neste processo o usuário requisita, à grade, a execução de uma aplicação (*Solicitar execução de aplicação*). Para que esta solicitação seja atendida pela grade, um nó deve ser escalonado para executar a solicitação. Este escalonamento deve levar em consideração a disponibilidade de recursos dos nós da grade (*Gerenciar aglomerado*, *Escalonar aplicações* e *Consultar informações de nós*). Uma vez alocado um nó que execute a solicitação do usuário (*Requisitar a execução de aplicação em um nó* e *Instanciar processo que executa aplicações*), a aplicação deve, então, ser preparada para executar. Esta preparação consiste basicamente de dois aspectos: baixar o

código executável da aplicação (que deverá estar previamente armazenado no repositório de aplicações) para o nó que executará a solicitação, e baixar os arquivos utilizados pela aplicação como entrada de dados (*Baixar aplicação*, *Obter aplicação*, *Baixar arquivos de entrada* e *Obter arquivos de entrada*).

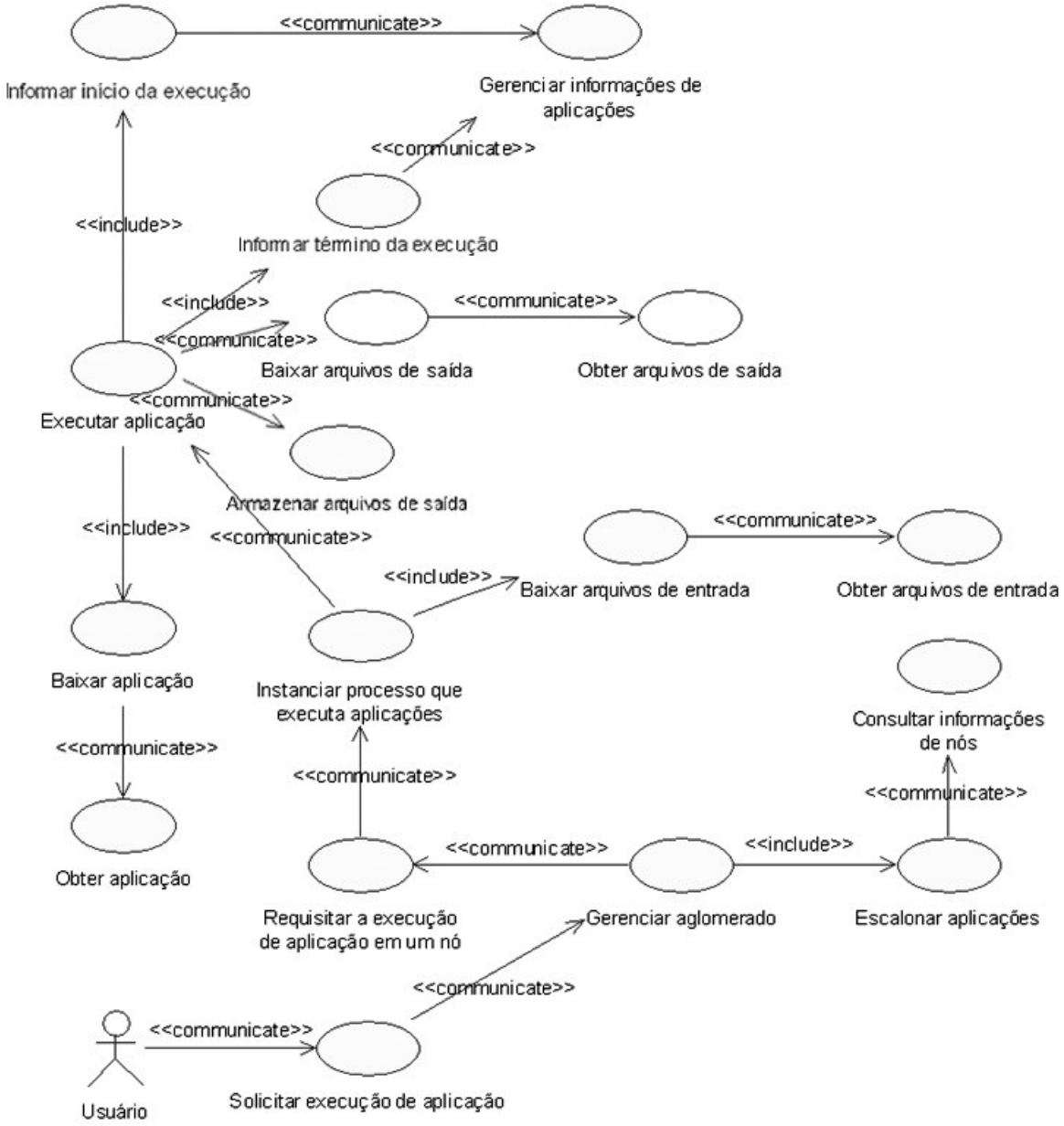


Figura 2.4: Cenário "Execução de aplicações"

A última tarefa da grade antes de iniciar efetivamente a execução da aplicação, é a de armazenar as informações relativas à execução em um repositório de informações de execução (*Informar início da execução* e *Gerenciar informações de aplicações*). Entre estas informações estão um identificador único de execução, o nó da grade que executa atualmente a solicitação, os argumentos da aplicação e a lista dos arquivos de entrada. A

aplicação é, então, instanciada e passa a executar (*Executar aplicação*).

A grade passa a monitorar a execução da aplicação, esperando por seu término. Assim, no término da execução da aplicação, a grade estará ciente deste fato, e assim poderá retornar os resultados da computação ao usuário que a solicitou. Os resultados da computação no MAG tomam a forma de arquivos de saída, gerados durante a execução da aplicação (*Armazenar arquivos de saída*, *Baixar arquivos de saída* e *Obter arquivos de saída*). Por fim, as informações relativas à execução devem ser removidas do repositório de informações de execução e armazenadas em arquivos de *log* para que seja mantido um histórico destes dados (*Informar término da execução* e *Gerenciar informações de aplicações*).

Liberação de nós

O MAG utiliza o poder computacional ocioso de máquinas existentes em redes computacionais para executar aplicações. Isto implica dizer que as máquinas que compõem a grade não são, necessariamente, dedicadas à grade, podendo pertencer a outros usuários. Dessa forma, é fundamental que o MAG disponibilize um mecanismo através do qual os usuários possam solicitar o uso exclusivo de seus recursos.

Quando um usuário requisita o uso de sua máquina, a grade deve tornar-se ciente de que novas computações não deverão ser alocadas naquele nó e que as computações que lá estejam executando sejam migradas para outros nós. Assim, dois mecanismos devem ser fornecidos aos usuários do MAG para que este processo seja possível: (a) um mecanismo através do qual o usuário possa informar à grade de sua intenção de utilizar o nó de maneira exclusiva, e (b) um mecanismo que permita a migração de aplicações Java entre os nós da grade. Este último mecanismo é detalhado no capítulo 3. O cenário da Figura 2.5 traz os casos de uso relativos ao processo de liberação de nós da grade.

Ao solicitar a liberação do nó (*Liberar nó*), a máquina do usuário deve ser desregistrada do aglomerado, para que novas computações não sejam escalonadas para executar no mesmo (*Desregistrar nó*, *Gerenciar aglomerado* e *Armazenar informações de nós*). Além disso, um novo nó deverá ser escalonado para abrigar a computação após sua migração (*Migrar aplicações em um nó*, *Escalonar aplicações*, *Consultar informações de nós* e *Migrar aplicação*). Após efetuar a migração da aplicação para uma outra localidade, a grade deverá modificar, em seu repositório de informações de execução, a informação re-

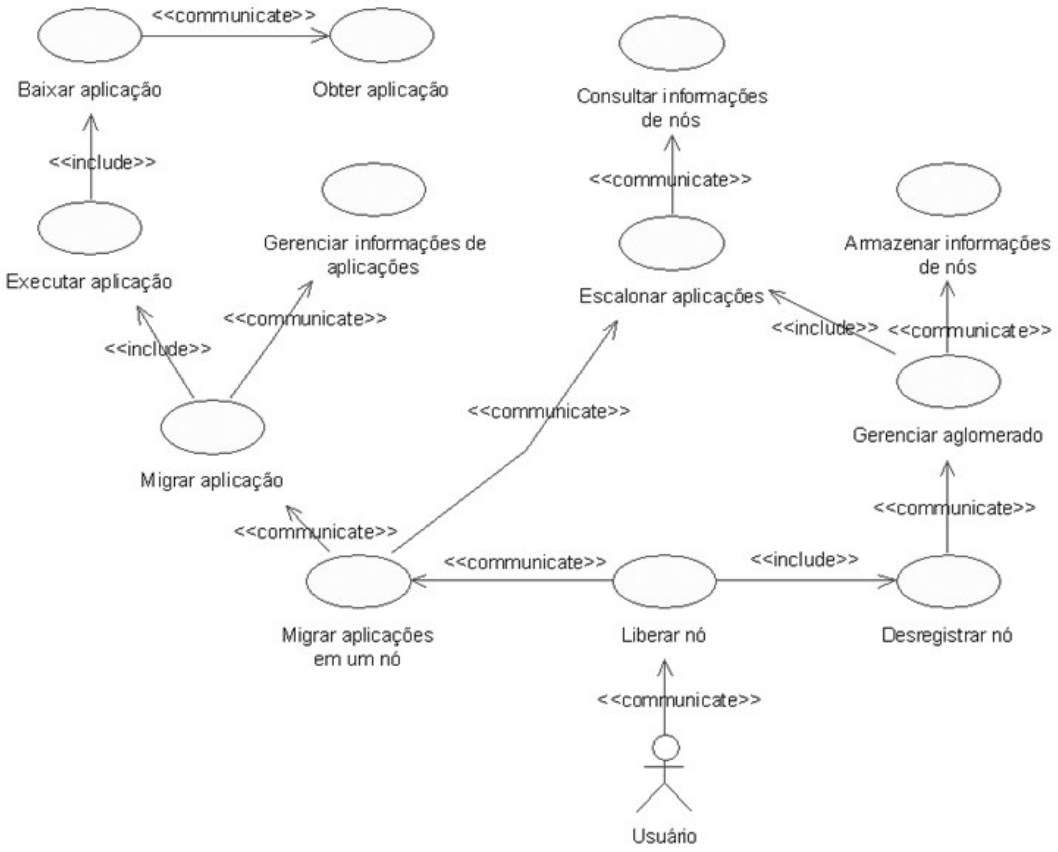


Figura 2.5: Cenário "Liberação de nós"

lativa à localização atual da aplicação em execução (*Gerenciar informações de execução*), para então poder retomar sua execução (*Executar aplicação*, *Baixar aplicação* e *Obter aplicação*).

Recuperação de nós

Tolerância a falhas é um aspecto fundamental a ser tratado em grades computacionais. Isto se deve ao fato dos nós não serem dedicados, e não se encontrarem em um ambiente controlado, como ocorre em máquinas paralelas e *clusters* de computadores (e.g. *clusters* Beowulf [BEO]). O cenário composto pelos casos de uso relacionados ao processo de recuperação de nós da grade pode ser visto na Figura 2.6.

Para garantir a recuperação de aplicações em caso de falha, o MAG propõe o uso da abordagem de *checkpointing*. Nesta abordagem, durante a execução das aplicações, seus estados de execução (*checkpoints*) são periodicamente salvos em um repositório estável, que deve sobreviver a eventuais falhas que ocorram no sistema (*Capturar checkpoint* e *Salvar checkpoint*).

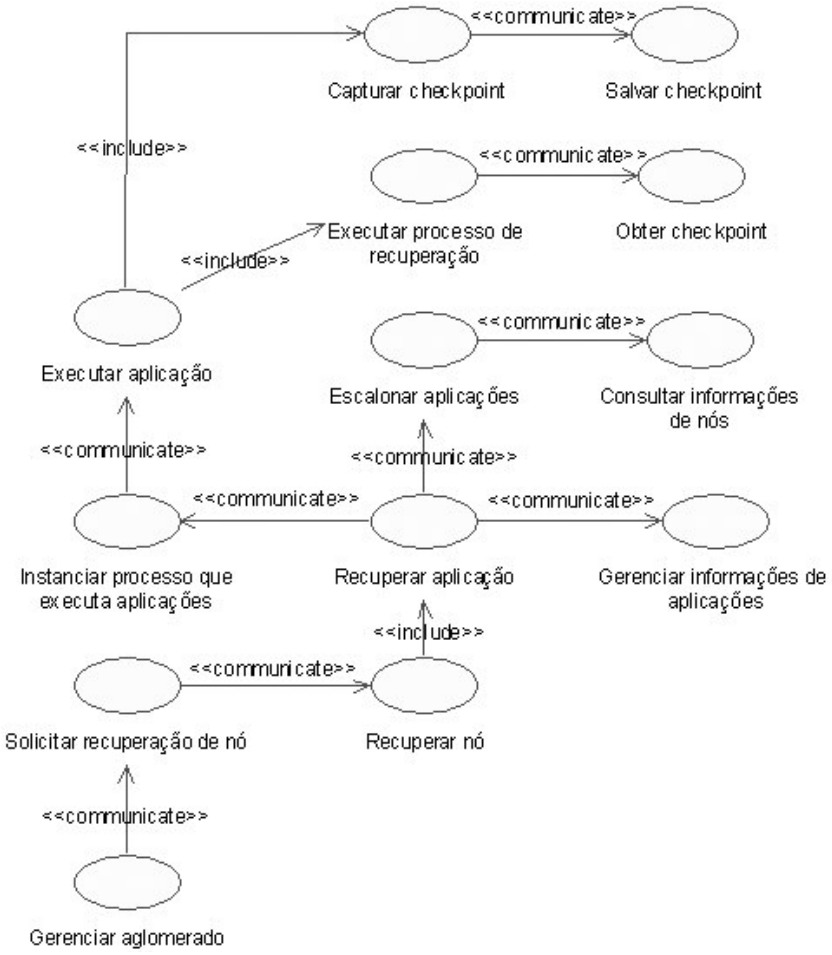


Figura 2.6: Cenário "Recuperação de nós"

Caso a falha de um nó seja detectada, o processo de recuperação das aplicações que lá executavam (*Gerenciar aglomerado*, *Solicitar recuperação de nó*, *Recuperar nó* e *Recuperar aplicação*) inicia-se com a pesquisa de quais aplicações executavam no nó que falhou (*Gerenciar informações de aplicações*). Um novo nó é então escalonado para cada aplicação (*Escalonar aplicações* e *Consultar informações de nós*), de maneira que as mesmas possam continuar sua execução a partir de seu último estado pré-falha. Para tanto, a grade deve tornar a aplicação ciente de que ela deverá recuperar as computações que falharam, passando a elas o último *checkpoint* salvo antes da ocorrência da falha (*Instanciar processo que executa aplicações*, *Executar aplicação*, *Executar processo de recuperação* e *Obter checkpoint*).

Atualização de informações de nós

Alguns dos cenários mostrados anteriormente dependem fortemente do processo de escalonamento de aplicações. Para que este escalonamento possa ser efetuado, é necessário que a grade conheça os tipos e quantidades de recursos disponíveis na grade. Assim, passa a ser necessário um processo de atualização de informações de nós, no qual os nós informam à grade sua disponibilidade de recursos, ou a ausência deles. Os casos de uso envolvidos neste processo podem ser vistos na Figura 2.7.

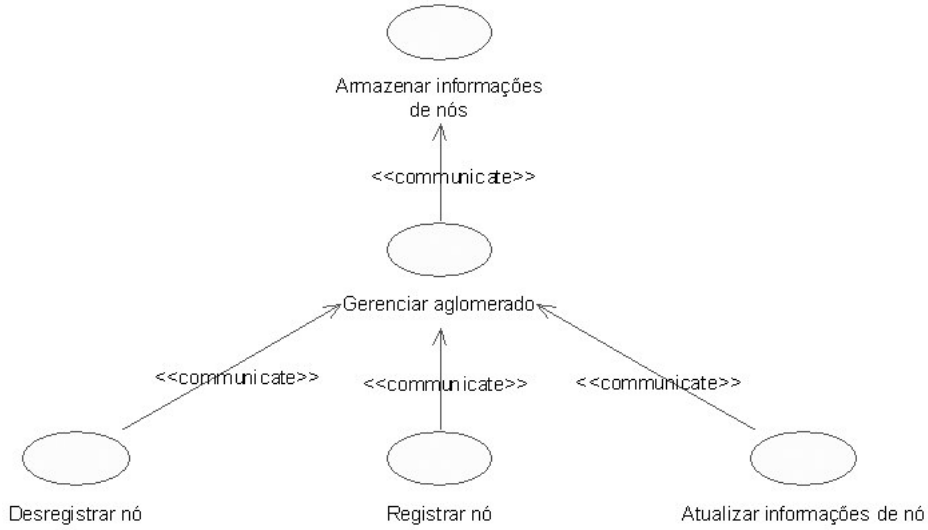


Figura 2.7: Cenário "Atualização de informações de nós"

Ao ser agregado à grade, um nó deve ser registrado (*Registrar nó*), de maneira a informar sua disponibilidade de recursos inicial e também informações estáticas sobre seus recursos (freqüência do processador, quantidade total de memória, etc). Estes dados são armazenados pela grade (*Gerenciar aglomerado* e *Armazenar informações de nós*) e periodicamente atualizados, através de notificações vindas dos próprios nós (*Atualizar informações de nó*). Caso o nó deixe de fazer parte da grade (seu usuário requisitou seu uso exclusivo ou simplesmente a máquina foi desligada) a grade deve então remover estas informações de suas bases de dados (*Desregistrar nó*).

Visualização de aplicações em execução na grade

Este último cenário é composto apenas por 2 casos de uso, e está mostrado na Figura 2.8. Neste cenário, o usuário têm acesso às informações armazenadas no repositório de informações de execução, podendo visualizar quais aplicações executam na grade e em

quais nós (*Visualizar grade* e *Gerenciar informações de aplicações*).

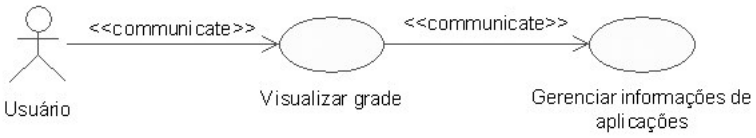


Figura 2.8: Cenário "Visualização de aplicações em execução na grade"

2.3.2 Modelo da Sociedade de Agentes

Os diagramas de *identificação dos agentes* (*Agent Identification* ou AId) e de *descrição de ontologia do domínio* (*Domain Ontology Description* ou DOD) compõem este modelo. Cada um destes diagramas será explorado no contexto do MAG, no decorrer desta seção.

Identificação dos agentes (AId)

De acordo com a *Agil PASSI*, um agente pode ser definido como um caso de uso ou um conjunto de casos de uso. Assim, o diagrama AId é produzido a partir do diagrama de descrição de requisitos de domínio, agrupando-se os casos de uso encontrados em pacotes, representando os diferentes agentes que compõem a infraestrutura do sistema. As comunicações entre os casos de uso do diagrama de descrição de requisitos de domínio são mantidos neste diagrama, passando a representar agora, a comunicação existente entre os agentes da arquitetura.

O diagrama AId, entretanto, precisou ser adaptado para a realidade do MAG. Isto se deve ao fato deste projeto não ser formado apenas por agentes, mas também por componentes de software, uma vez que o mesmo reutiliza a infraestrutura do Integrade. A representação dos componentes de software mostrados neste diagrama é a de um pacote com o estereótipo *Component*. A Figura 2.9 mostra o diagrama de identificação de agentes do MAG.

Cada agente e componente desta arquitetura recebeu uma denominação relativa ao papel que desempenha. Os componentes de software desta arquitetura que foram reutilizados do Integrade têm os seus nomes originais apresentados entre parênteses. Estes

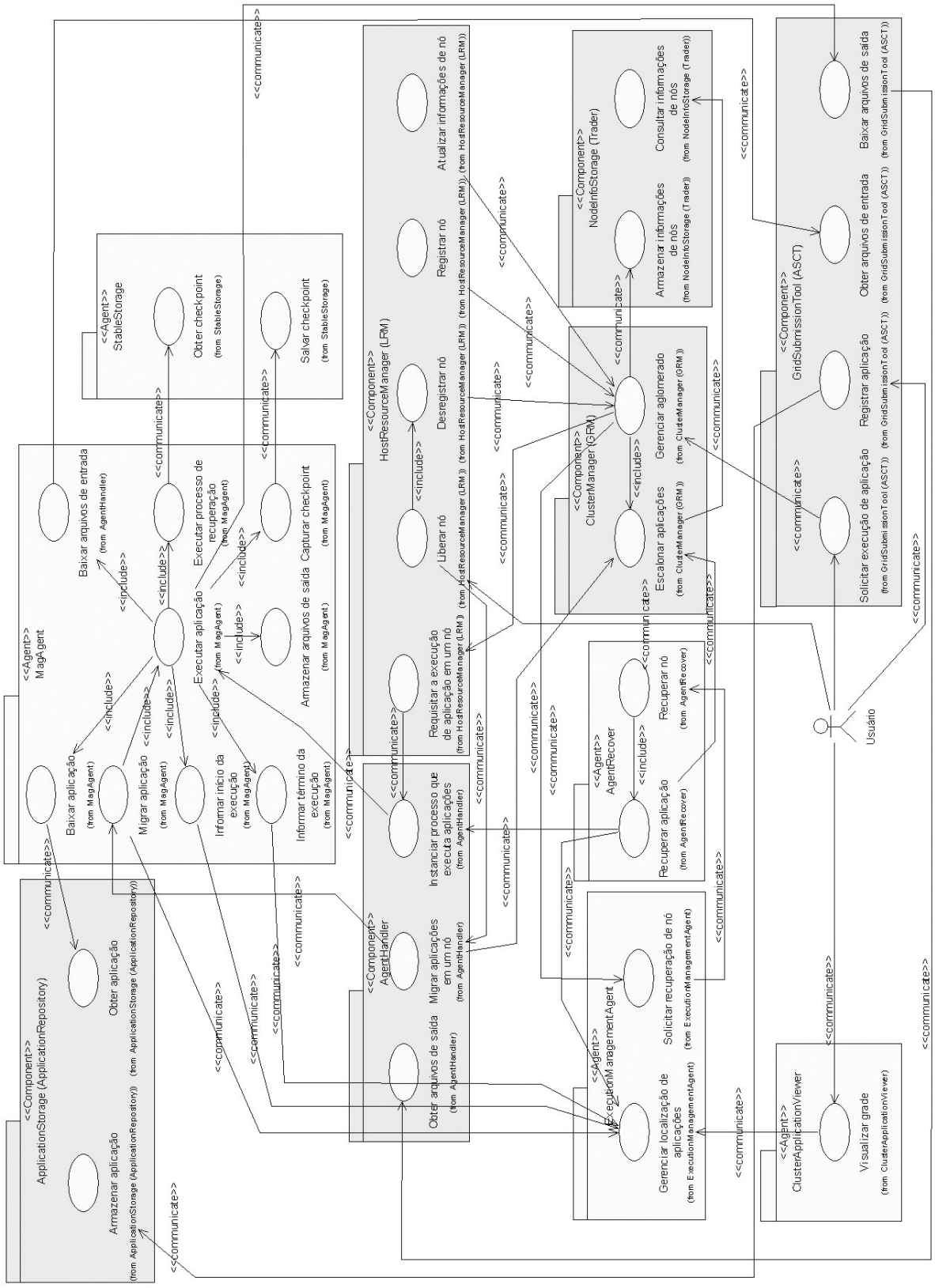


Figura 2.9: Diagrama de identificação de agentes do MAG

componentes tiveram que ser ligeiramente modificados para funcionar em conjunto com a arquitetura do MAG. Os agentes / componentes identificados neste diagrama foram:

- *GridSubmissionTool (ASCT):* componente utilizado pelo usuário para a submissão e controle da execução de aplicações. Através dele é possível registrar aplicações, requisitar suas execuções e coletar seus resultados. Foi reutilizada a ferramenta ASCT do Integrade;

- *ClusterManager (GRM):* componente responsável por efetuar tarefas de gerenciamento do aglomerado tais como o registro, desregistro e atualização de informações de nós conectados à grade, detecção de falhas de nós e escalonamento de tarefas. O componente GRM do Integrade efetua este papel;

- *NodeInfoStorage (Trader):* este componente é um armazém de dados de informações de nós, permitindo o armazenamento e a recuperação destas informações. Este componente foi reutilizado do projeto Integrade;

- *ApplicationStorage (ApplicationRepository):* componente que permite o armazenamento e a recuperação de aplicações para execução na grade. Toda aplicação a ser submetida para execução deverá ser previamente armazenada neste repositório. O MAG utiliza o *ApplicationRepository* do Integrade para efetuar esta tarefa;

- *HostResourceManager (LRM):* este componente deve executar em nós que compartilham recursos com a grade. Ele é responsável por registrar e desregistrar o nó junto à grade, informando que ele faz parte ou não de um aglomerado. Além disso, monitora os recursos da máquina em que executa, enviando periodicamente estas informações ao *ClusterManager*, fazendo com que a grade sempre conheça a disponibilidade de seus recursos. Ele é também responsável por requisitar a execução de aplicações no nó em que executa, e por iniciar o processo de liberação deste mesmo nó, caso isto tenha sido requerido por um usuário. O componente LRM do Integrade foi utilizado para exercer estas funcionalidades;

- *AgentHandler:* componente que mantêm a plataforma de agentes que executa em nós da grade. É responsável pela instanciação de agentes para a execução de aplicações, notificação da necessidade de migração para localidades remotas e retorno dos arquivos de saída da aplicação ao usuário que originou a requisição;

- *ClusterApplicationViewer:* o *ClusterApplicationViewer* (ou CAV) [GCLS05] é uma ferramenta gráfica utilizada pelos administradores da grade para visualizar as aplicações que executam na mesma. Através dele é possível saber em que nós as aplicações da

grade executam, além de permitir a consulta a informações detalhadas sobre cada uma destas execuções;

- *ExecutionManagementAgent:* este agente armazena dados relativos às aplicações em execução na grade, como o identificador único da execução, o nó em que a aplicação executa atualmente, a lista dos arquivos de entrada e saída, o nome do usuário que requisitou a computação, etc. Estes dados são utilizados pelo processo de recuperação de execuções que tenham falhado, além de permitir a consulta de informações sobre as execuções em curso na grade através do *ClusterApplicationViewer*;

- *StableStorage:* é um repositório estável de *checkpoints* de aplicações. É dito estável porque deve sobreviver a eventuais falhas que ocorram no sistema;

- *MagAgent:* principal agente da arquitetura do MAG. Ele é responsável por instanciar e executar as requisições feitas à grade. Ele também monitora a execução destas aplicações, aguardando por seu término, para poder notificar o usuário deste fato, além de migrá-las caso isto seja requerido. Também salva o *checkpoint* das aplicações periodicamente no *StableStorage*, de forma a minimizar a perda de computações já executadas, caso ocorram falhas durante a execução das aplicações. Cada *MagAgent* é responsável por manter os dados de execução de sua aplicação atualizados junto ao *ExecutionManagementAgent*;

- *AgentRecover:* este agente é criado com o objetivo de tratar uma falha que tenha ocorrido em um nó da grade. Consulta o *ExecutionManagementAgent* para obter a informação de quais aplicações executavam no nó que falhou, solicita o escalonamento de novos nós para receber estas aplicações e recria os agentes que executarão as mesmas, informando a eles que deverão recuperar suas computações a partir de seus últimos *checkpoints*.

A partir do diagrama de identificação de agentes, o sistema passa a ser definido em função dos agentes que o compõem e das relações entre eles. Este modelo será refinado nos diagramas a seguir, de maneira que seja possível projetar a estrutura interna dos agentes do sistema e definir como será realizada a comunicação entre eles.

Descrição de ontologia de domínio (DOD)

O objetivo do diagrama de descrição de ontologia de domínio (ou DOD) é modelar as ontologias presentes no sistema, ou seja, representar o conhecimento dos agentes que compõem o sistema. É este conhecimento do sistema que possibilita que estes agentes possam interagir, trocando informações para atuar de maneira cooperativa. A ontologia do domínio precisa ser bem definida para evitar ambigüidades na interpretação das mensagens trocadas entre os agentes do sistema.

O diagrama DOD do MAG pode ser visto na Figura 2.10. Este diagrama ilustra a ontologia relativa à arquitetura do MAG. Esta ontologia foi apenas parcialmente implementada, dado que alguns componentes comunicam-se através da tecnologia CORBA de objetos distribuídos, não utilizando a comunicação padrão de agentes baseada em FIPA-ACL[3] [Fou02a].

Ontologias são descritas em termos de três diferentes tipos de elementos: conceitos, predicados e ações. Na metodologia *Agil PASSI*, estes elementos são representados através de um diagrama de classes UML com modificações em seus estereótipos.

Os conceitos representam as entidades e categorias presentes no domínio do sistema. Eles aparecem no diagrama DOD com o estereótipo *Concept*. Os conceitos podem ser vistos informalmente como "as entidades sobre as quais o assunto da comunicação gira". No diagrama DOD do MAG podem ser vistos vários conceitos como, por exemplo, o *Checkpoint*, utilizado na comunicação entre o *MagAgent* e o *StableStorage* para efetuar a salva e a recuperação do *checkpoint* das aplicações. O acordo entre os agentes para a realização destas operações não seria possível caso eles desconhecessem a existência deste conceito no sistema.

Os predicados aparecem na forma de inferências sobre os conceitos do domínio. Apresentam o estereótipo *Predicate* no diagrama DOD. Estão relacionados às propriedades dos conceitos e geralmente apresentam-se como questionamentos sobre estas propriedades. Por exemplo, pode ser necessário ao *MagAgent* descobrir se um dado *checkpoint* é o último (ou seja, o mais recente) de uma aplicação. Para tanto, o *MagAgent* conta com o predicado *IsLastCheckpoint* para consultar junto ao *StableStorage* sobre a validade desta inferência.

[3] *Agent Communication Language*. Linguagem formal utilizada por agentes de software para que estes possam comunicar-se. Esta linguagem segue uma especificação da FIPA [FIP] para a definição da estrutura de mensagens de comunicação de agentes

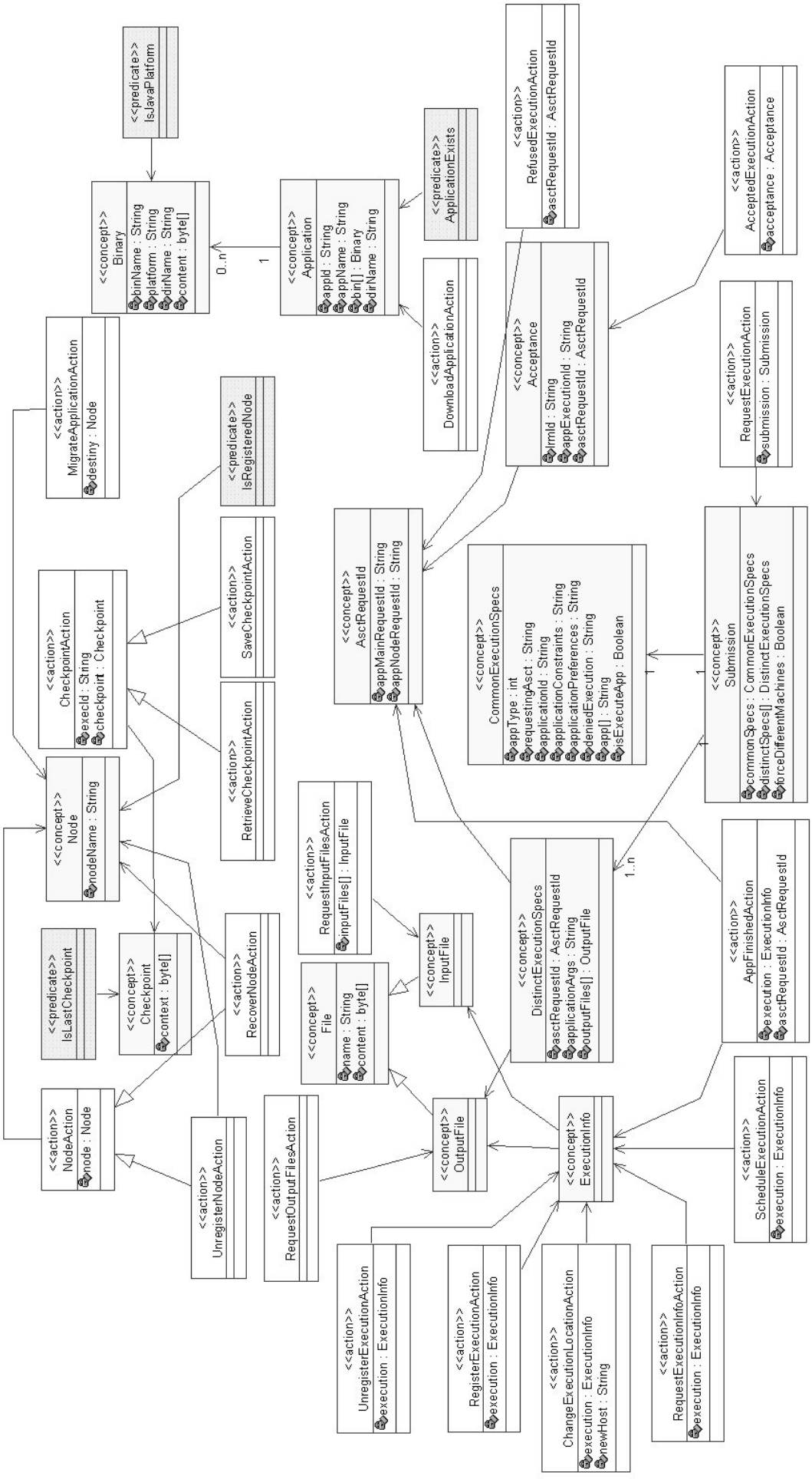


Figura 2.10: Diagrama de descrição de ontologia de domínio do MAG

Por último, as ações representam tarefas relacionadas aos conceitos da ontologia. No diagrama DOD aparecem com o estereótipo *Action*. As ações podem ser entendidas como requisições de trabalho trocadas entre agentes. Na ontologia do MAG, pode ser visto que o conceito *Checkpoint* tem duas ações associadas: *RetrieveCheckpointAction* e *SaveCheckpointAction*, sendo que estas ações representam, respectivamente, as requisições de recuperação e salva do *checkpoint* da aplicação.

Uma vez definida a ontologia do sistema, passa a ser possível identificar as mensagens trocadas entre os agentes. A ontologia é importante no sentido de permitir que os agentes distinguam a semântica das mensagens por eles recebidas.

2.4 Implementação

O objetivo do modelo de código é definir como o sistema será efetivamente implementado. É neste nível que a modelagem da estrutura dos agentes e de seus comportamentos são definidos. Para cumprir este objetivo, quatro diagramas são definidos neste modelo: diagrama de ontologia de comunicação (*Communication Ontology Diagram* ou COD), diagrama de definição de estrutura multiagente (*Multi-Agent Structure Definition* ou MASD), diagrama de definição de estrutura de agente (*Single-Agent Structure Definition* ou SASD) e diagrama de descrição de comportamento multiagente (*Multi-Agent Behaviour Description* ou MABD). Cada um destes diagramas descreve um aspecto de implementação da arquitetura do MAG.

O MAG foi desenvolvido utilizando as linguagens Java e C++. O código C++ se fez necessário para que o MAG pudesse utilizar o Integrade como base de sua implementação, tendo sido inserido no módulo LRM, com dois objetivos: (1) permitir que o LRM repasse ao MAG as requisições encaminhadas a este *middleware*, e (2) inserir o mecanismo que permite a liberação de nós da grade. O código-fonte do MAG é constituído por 8278 linhas de código, sendo destas 874 C++ e 7404 Java. As 874 linhas de código C++ foram adicionadas ao LRM, enquanto que das 7404 linhas de código Java, 98 foram adicionadas ao ASCT, 27 ao repositório de aplicações, e 527 ao GRM. As 6752 linhas de código restantes representam o código que compõe infraestrutura do MAG.

Os componentes do Integrade (GRM, LRM, ApplicationRepository e ASCT) não estão cobertos pelo escopo deste trabalho. Maiores informações sobre estes compo-

nentes podem ser encontradas em [Gol04].

2.4.1 Diagrama de ontologia de comunicação (COD)

Este diagrama é expresso como um diagrama de classes da UML e seu principal objetivo é associar o conhecimento dos agentes (modelado através de sua ontologia) às comunicações que ocorrem entre os mesmos. No diagrama COD, dois tipos de elementos são possíveis: agente e comunicação, representados pelos estereótipos *Agent* e *Communication*, respectivamente. Neste diagrama os agentes podem apresentar atributos que sejam relevantes ao processo de comunicação. Já as comunicações são expressas em termos de três atributos, explicados a seguir:

a) *Ontologia:* a comunicação deve estar associada a um dos elementos da ontologia já modelada no diagrama DOD, seja um conceito, um predicado ou uma ação. É a ontologia que define a semântica da comunicação, fazendo com que os agentes possam entender o significado das mensagens trocadas: se é apenas a transmissão de dados relativos a um conceito, uma consulta utilizando um predicado ou uma requisição de execução de tarefa através de uma ação;

b) *Linguagem de conteúdo:* enquanto a ontologia define o significado da comunicação, a linguagem define as notações léxicas e sintáticas que podem ser utilizadas na comunicação. Dois exemplos de linguagem são a RDF [Fou01] (*Resource Description Framework*), baseada em XML e a linguagem SL [Fou02d] (*Semantic Language*), criada pela FIPA com o intuito de torná-la a linguagem padrão para a comunicação entre agentes;

c) *Protocolo de interação:* este protocolo define como os agentes devem interagir durante sua comunicação. A FIPA define uma série de protocolos de interação que podem ser utilizados na comunicação entre agentes. Exemplos destes protocolos são o FIPA-Request [Fou03], FIPA-Subscribe [Fou02e] e FIPA-Contract-Net [Fou02b].

Para representar as comunicações do MAG, foi necessário que este diagrama sofresse adaptações, dado que as comunicações que envolvem componentes de software não fazem uso de ontologias e são baseados em CORBA IIOP. Assim, este diagrama foi modificado em três pontos: (1) além de agentes, agora os componentes também aparecem

neste diagrama (classes com o estereótipo *Component*); (2) as comunicações que envolvem componentes não mostram a ontologia envolvida, além de terem seus protocolos de interação substituídos pelo protocolo de transporte IIOP; e (3) o nome atribuído à comunicação passa a representar o nome do método CORBA invocado. Já as comunicações que se apresentam sem o campo de protocolo representam invocações a métodos locais. Estas comunicações são mostradas no diagrama com o intuito de facilitar o entendimento do leitor sobre a comunicação na ocorrência de determinados eventos, como a migração de agentes. O diagrama COD do MAG pode ser visto na Figura 2.11.

Todas as comunicações entre agentes utilizam elementos da ontologia definida no diagrama DOD. Por exemplo, entre o *MagAgent* e o *StableStorage* existem duas comunicações: *SaveCheckpoint* e *QueryCheckpoint*, que representam as solicitações de salva e recuperação de um *checkpoint* junto ao *StableStorage*. As informações transmitidas nestas comunicações seguem especificações definidas por elementos da ontologia do MAG, neste caso, as ações *SaveCheckpointAction* e *RetrieveCheckpointAction*. Dessa forma, ao receber uma mensagem, o *StableStorage* é capaz de identificar seu significado através da semântica definida pela ontologia. A linguagem utilizada na comunicação entre os agentes do MAG é a SL [Fou02d] (*Semantic Language*), padrão da plataforma JADE.

Dois protocolos de interação são utilizados na comunicação entre agentes no MAG: os protocolos FIPA-Request [Fou03] e FIPA-Query [Fou02c]. O protocolo FIPA-Request permite que um agente solicite a outro a realização de uma operação. Este protocolo está ilustrado no diagrama de seqüência AUML [OPB00] da Figura 2.12(a): um agente iniciador (*Initiator*) efetua o procedimento de requisição (*request*) de um serviço a um agente participante (*Participant*). Este, por sua vez, pode aceitar ou não a requisição recebida (*agree* ou *refuse*). Em caso positivo, o agente participante efetuará o processamento da requisição e responderá ao agente iniciador com um dos três tipos de mensagens: (1) uma mensagem *failure* caso ocorra uma falha; (2) uma mensagem informando que a operação foi efetuada com sucesso pelo agente participante (*inform-done*); (3) uma mensagem contendo o resultado do processamento (*inform-result*). No MAG o protocolo FIPA-Request é utilizada nas comunicações *SaveCheckpoint*, *RegisterAgent*, *ChangeAgentHost*, *UnregisterAgent* e *RecoverNode*.

O protocolo FIPA-Query é utilizado para solicitar consultas a um agente. A Figura 2.12(b) mostra o diagrama de seqüência AUML deste protocolo. Utilizando o protocolo FIPA-Query é possível efetuar dois diferentes tipos de consultas a agentes. Para

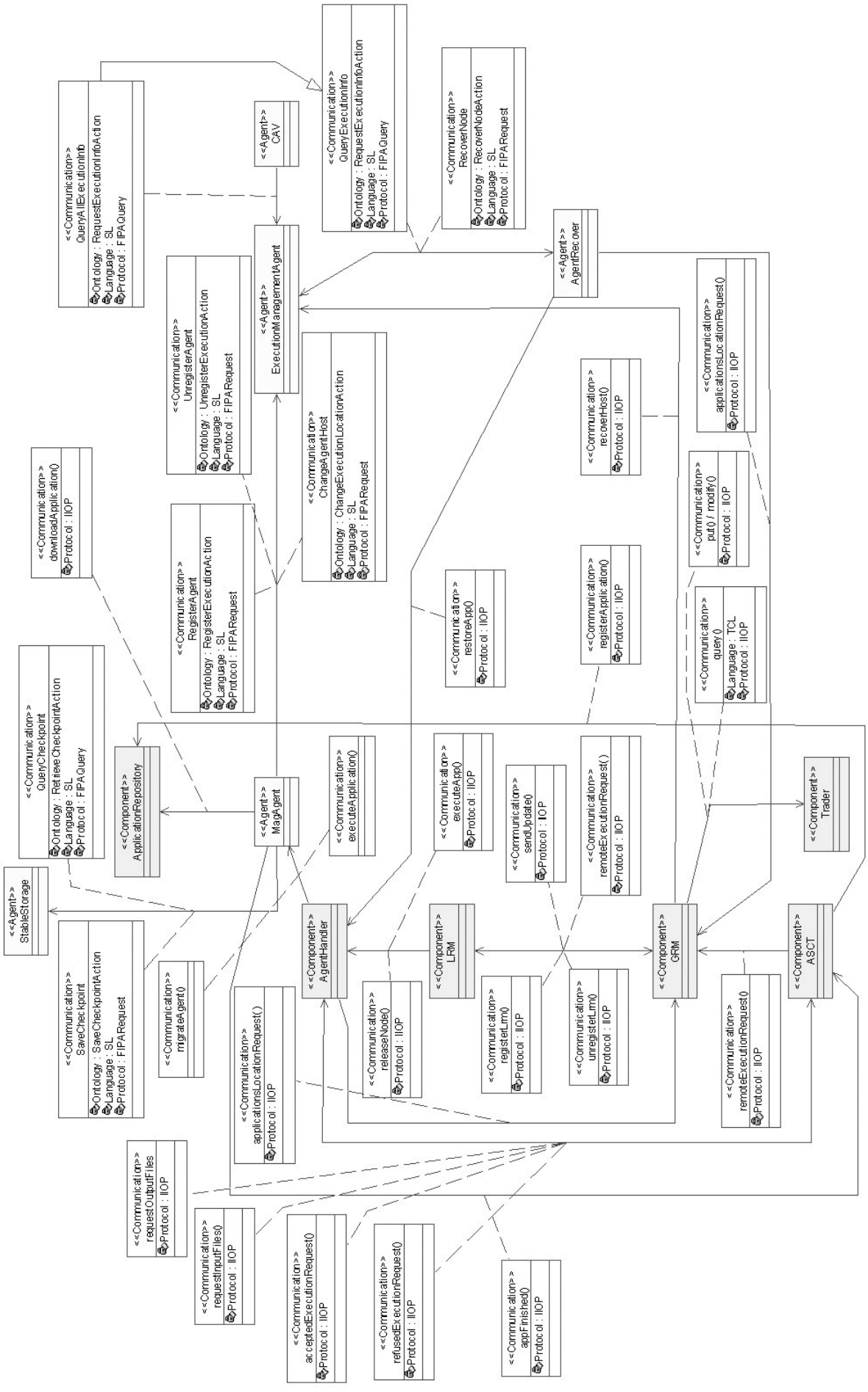
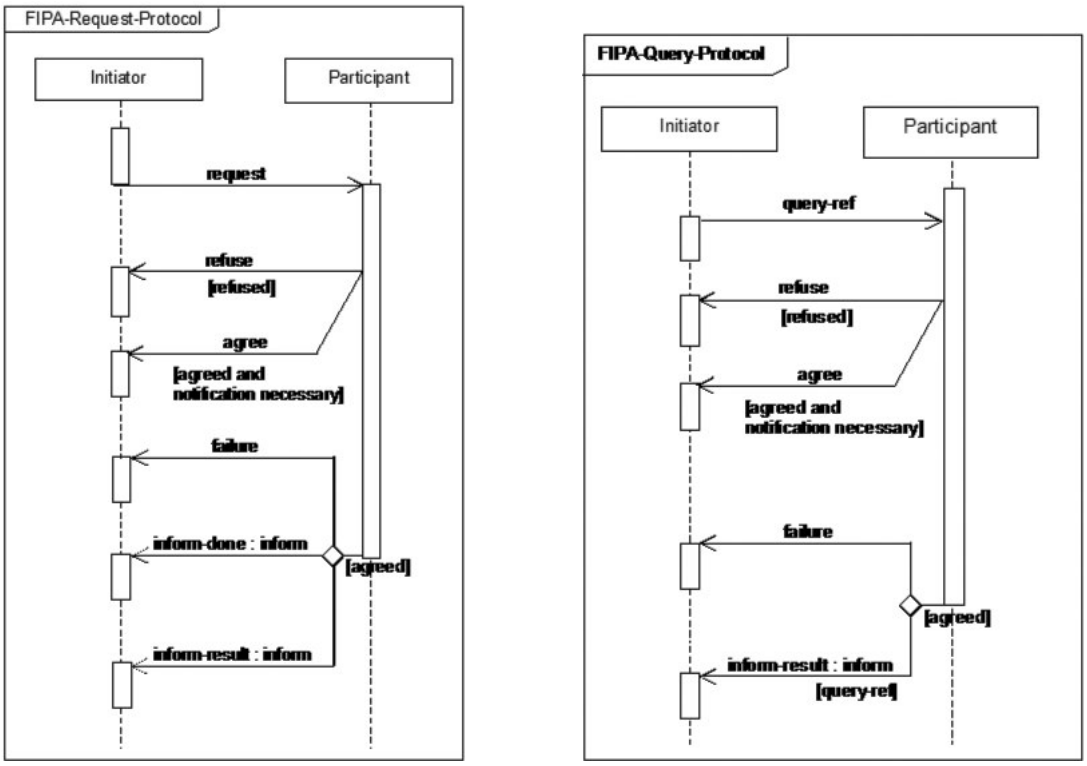


Figura 2.11: Diagrama de ontologia de comunicação do MAG



(a) Protocolo FIPA-Request (b) Protocolo FIPA-Query

Figura 2.12: Protocolos FIPA

tanto, duas diferentes performativas são disponibilizadas: *query-if* e *query-ref*. A performativa *query-if* é utilizada para consultar um agente a respeito de uma dada proposição, cujo resultado é booleano. Este tipo de consulta não é utilizada na infraestrutura do MAG e, portanto, não será explanado. Já as consultas baseadas na performativa *query-ref* são utilizadas quando um agente deseja consultar outro sobre um determinado objeto identificado (e.g. uma consulta a um determinado *checkpoint* junto ao *StableStorage*). Como pode ser visto na Figura 2.12(b), a interação inicia-se com uma solicitação de consulta (*query-ref*) de um agente iniciador (*Initiator*) a respeito de algum objeto junto a um agente participante (*Participant*). Este por sua vez pode aceitar ou não a solicitação de consulta (*agree* ou *refuse*). Caso o agente participante aceite a solicitação, ele então efetua o processamento da consulta e retorna seu resultado ao agente iniciador (*inform-result*). Caso uma falha ocorra no processamento da consulta, uma mensagem de falha deve ser retornada ao agente iniciador (*failure*). No MAG as comunicações *QueryCheckpoint*, *QueryExecutionInfo* e *QueryAllExecutionInfo* utilizam este protocolo.

Na arquitetura do Integrade o componente de software Trader foi substituído pelo serviço de negócios do CORBA (*trading*) [Obj00] e posteriormente reutilizado pela

arquitetura do MAG. A comunicação entre o GRM e o Trader é baseada na linguagem TCL (*Trader Constraint Language*), utilizada para efetuar consultas a referências de objetos CORBA que atendam a determinados requisitos.

2.4.2 Diagrama de definição de estrutura multiagente (MASD)

Através do diagrama de definição de estrutura multiagente é possível ter uma visão completa da sociedade de agentes e dos atores que compõem o sistema. Este diagrama é representado através de um diagrama de classes, com as classes simbolizando os agentes identificados através do diagrama AId (seção 2.3.2), e com os relacionamentos entre classes representando a comunicação entre os mesmos. Os métodos dos agentes representam as tarefas ou comportamentos dos mesmos. O diagrama MASD do MAG é mostrado na Figura 2.13.

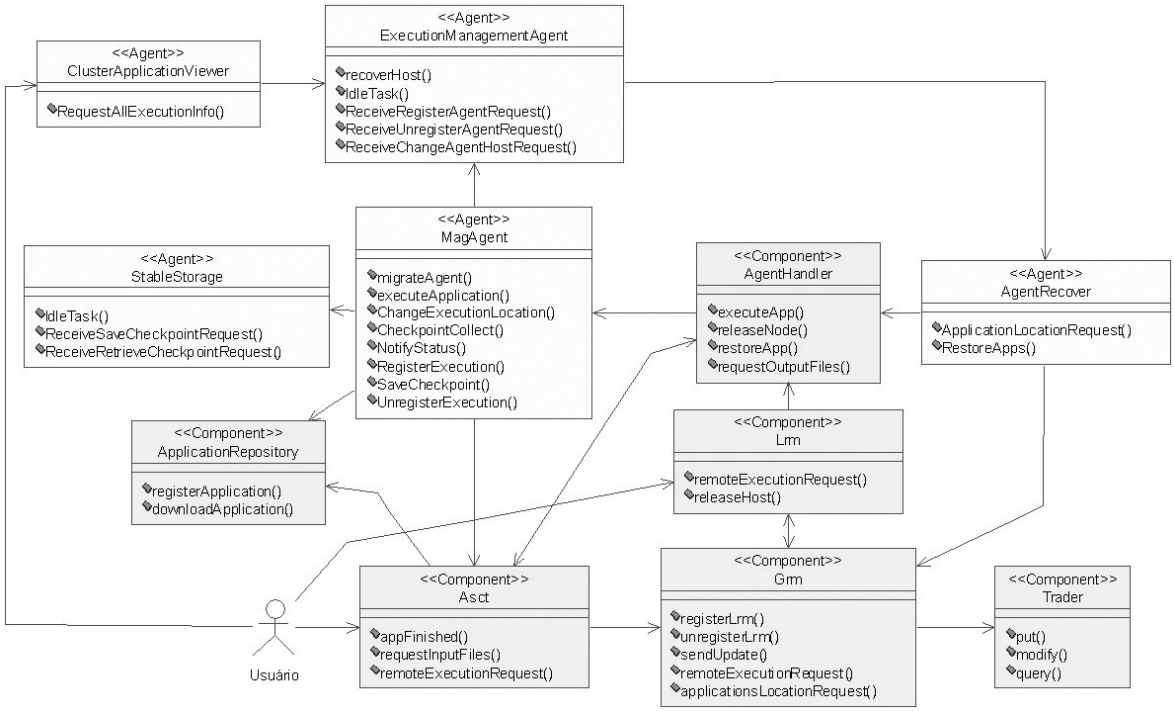


Figura 2.13: Diagrama de definição de estrutura multiagente do MAG

Além dos agentes, este diagrama também apresenta os componentes de software que compõem a infraestrutura do MAG. Estes componentes contêm métodos que podem ser invocados por outros componentes ou por agentes, através de invocações CORBA. No diagrama da Figura 2.13, os métodos que aparecem com a inicial maiúscula representam tarefas dos agentes, enquanto aqueles cuja inicial é minúscula indica que foram implementados como métodos (locais ou remotos).

2.4.3 Diagrama de definição de estrutura de agente (SASD)

Para cada agente do diagrama MASD deve ser gerado um diagrama de definição de estrutura de agente (SASD), que corresponde à representação da estrutura interna do mesmo. Neste diagrama são apresentadas as tarefas (ou comportamentos) dos agentes, bem como suas interfaces. A exemplo do diagrama MASD, o SASD também utiliza diagramas de classes como sua forma de apresentação.

Nesta seção serão apresentados os agentes e o componente de software *AgentHandler*, que formam a infraestrutura do *middleware* MAG. Dado que a metodologia *Agil PASSI* não prevê o uso do diagrama SASD para representar componentes de software, a estrutura do *AgentHandler* será explicada através da descrição de sua interface.

MagAgent

O *MagAgent* é o principal agente da arquitetura do MAG. Ele é responsável por instanciar e executar as aplicações na grade, além de incorporar a capacidade de migrar aplicações entre nós da grade, utilizando para isto o arcabouço MAG/Brakes que será explicado em detalhes no capítulo 3.

O diagrama SASD do *MagAgent* é apresentado na Figura 2.14. O *MagAgent* é composto por sete classes, sendo uma delas a classe principal do agente e as outras classes de comportamentos que podem ser executados pelo mesmo. Os comportamentos que estendem a classe *OneShotBehaviour* são executados pelo agente apenas uma vez, sendo destruídos após o término de sua execução. Já os comportamentos que estendem a classe *TickerBehaviour* executam periodicamente com intervalos de tempo fixos entre duas execuções consecutivas do código do comportamento.

As classes que compõem o diagrama SASD do *MagAgent* são explicadas a seguir:

- *MagAgent:* nesta classe encontram-se os métodos que controlam o ciclo de vida do agente, como os métodos de inicialização (`setup()`) e destruição (`takeDown()`) do agente. Esta é a classe principal do agente, responsável por adicionar novos comportamentos ao agente à medida que isto for sendo necessário. Contém os métodos `executeApplication()` e `migrateAgent()`, ambos invocados pelo componente *AgentHandler* quando solicitados a execução e a migração de uma aplicação,

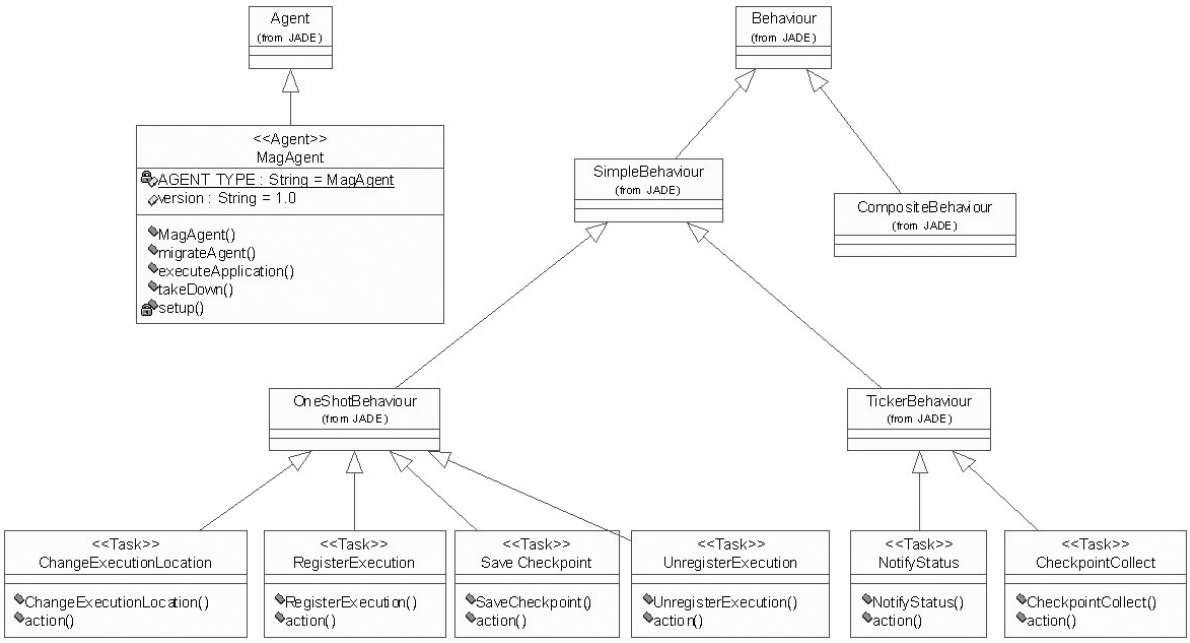


Figura 2.14: Diagrama de definição de estrutura do agente *MagAgent*

respectivamente;

- *DownloadApplication:* comportamento que acessa o repositório de aplicações (*ApplicationRepository*) e baixa o *bytecode* da aplicação para sua posterior instanciação e execução;

- *NotifyStatus:* comportamento que monitora a execução da aplicação, aguardando por seu término. Ao descobrir que a execução terminou notifica o *MagAgent* sobre este fato;

- *NotifyApplicationFinish:* através deste comportamento o ASCT do usuário que requisitou a execução é notificado do término da execução de sua aplicação, devendo o ASCT, a partir desta notificação, requisitar os resultados da computação à grade;

- *RequestInputFiles:* normalmente as aplicações que executam na grade necessitam ler arquivos de entrada de dados. Para tanto, é necessário que o *MagAgent* disponibilize estes arquivos à aplicação. O comportamento *RequestInputFiles* é responsável por solicitar ao ASCT que originou a requisição, os arquivos de entrada da aplicação, salvando-os no sistema de arquivos local;

- *RegisterAgent* e *UnregisterAgent:* estes dois comportamentos são utilizados para registrar e desregistrar os dados relativos à execução de uma aplicação junto ao *Execu-*

tionManagementAgent, responsável pelo gerenciamento das execuções de aplicação na grade, explicado a seguir;

- *ChangeAgentHost:* após a ocorrência de uma migração, o agente deve mudar a informação de localização da aplicação junto ao *ExecutionManagementAgent*. Esta classe de comportamento implementa esta funcionalidade.

ExecutionManagementAgent

O *ExecutionManagementAgent* é responsável por armazenar e disponibilizar informações de aplicações que executam na grade. As informações disponibilizadas por este agente são utilizadas em dois contextos: (1) colaborar com o mecanismo de tolerância a falhas da grade, já que através destas informações, é possível descobrir quais aplicações executavam em um nó que tenha falhado, além de obter os dados relativos à execução pré-falha destas aplicações; (2) permitir que administradores da grade tenham acesso aos dados da execução e às informações de localização das aplicações que executam na grade. A análise dos dados relativos à execução de aplicações em grades de computadores permite aos desenvolvedores da infra-estrutura de grade e aos administradores das mesmas a tomada de decisões relativas ao projeto do *middleware* de grade e a configuração do ambiente de execução, respectivamente.

Este agente está estruturado como mostrado na Figura 2.15. Como pode ser visto, este agente é composto por cinco classes: *ExecutionManagementAgent*, *IdleTask*, *ReceiveRegisterAgentRequest*, *ReceiveUnregisterAgentRequest*, *ReceiveChangeAgentHostRequest* e *ReceiveQueryExecutionInfo*.

- *ExecutionManagementAgent:* classe principal do agente. Além dos métodos de controle de ciclo de vida do agente (`setup()` e `takeDown()`), possui o método `recoverHost()`, invocado pelo GRM quando uma falha de nó é detectada. Quando uma requisição deste tipo é recebida o *ExecutionManagementAgent* instancia um novo agente de recuperação (*AgentRecover*) e repassa a ele todas as informações sobre as execuções que estavam alocadas ao nó que falhou;

- *IdleTask:* comportamento cíclico[4] responsável pelo recebimento de mensagens ori-

[4]Um comportamento cíclico é aquele que executa indefinidamente até que sua destruição seja explicitamente solicitada. Os comportamentos deste tipo devem herdar da classe *CyclicBehaviour*.

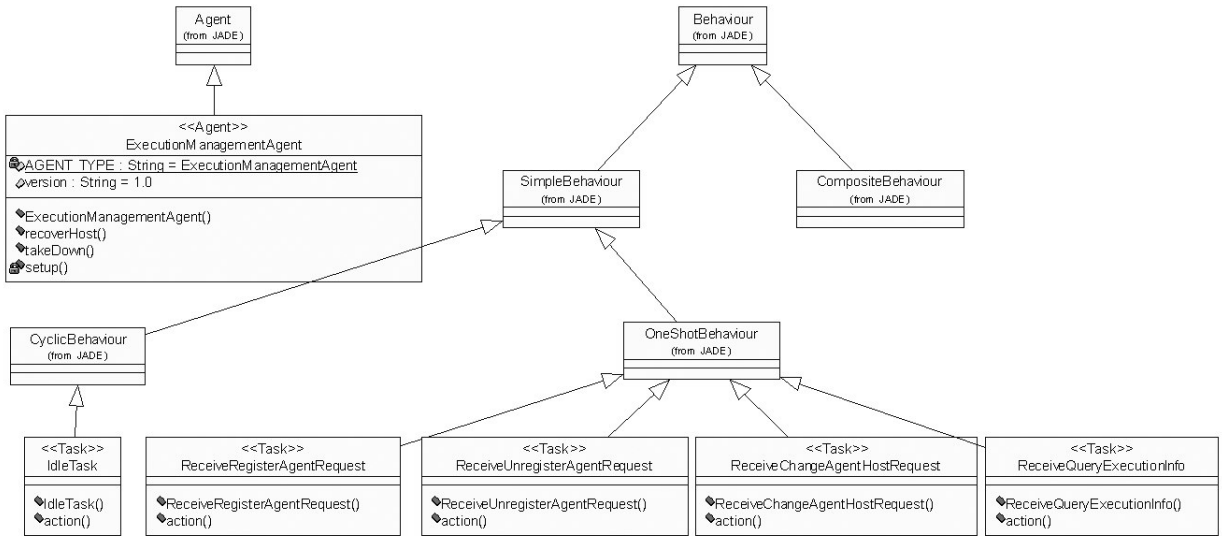


Figura 2.15: Diagrama de definição de estrutura do agente *ExecutionManagementAgent*

ginadas por outros agentes da grade. Ao receber uma mensagem, verifica o tipo de ação ou predicado associado a ela e aciona o comportamento responsável por seu tratamento. Por exemplo, ao receber uma mensagem do tipo *RegisterAgent* (seção 2.4.1), o comportamento *IdleTask* aciona o comportamento *ReceiveRegisterAgentRequest*, repassando a ele os dados contidos na mensagem;

- *ReceiveRegisterAgentRequest*, *ReceiveUnregisterAgentRequest*, *ReceiveChangeAgentHostRequest* e *ReceiveQueryExecutionInfo:* comportamentos responsáveis pelo tratamento de mensagens recebidas pelo agente. São acionados pelo comportamento *IdleTask* para efetuar o tratamento de mensagens. Tratam mensagens de solicitação de registro e desregistro de agentes, de mudança de nó (i.e. migração) e de solicitação de consulta de informações sobre aplicações, respectivamente.

Para efetuar o armazenamento das informações relativas à execução de aplicações da grade, o *ExecutionManagementAgent* utiliza o banco de dados XML Xindice. Cada execução gera um documento XML, cuja estrutura pode ser vista na Figura 2.16.

Este arquivo XML é dividido em duas partes básicas: (1) informações gerais sobre a aplicação que está sendo executada (*applicationInfo*), e (2) informações relativas à execução em andamento (*executionInfo*). Na *applicationInfo* armazena-se: o identificador da aplicação no repositório de aplicações, o nome da aplicação e a plataforma[5] de hardware

[5]Quatro plataformas são atualmente suportadas: Linux nas arquiteturas i686 e x86_64, Macintosh e Java

```
<application source="grid">  <!-- grid / web / mobile -->
        <appExecutionId>10000</appExecutionId>

        <applicationInfo>
                <appReposId>10000</appReposId>
                <binaryName>Fibonacci</binaryName>
                <platformType>Java</platformType> <!-- Linux_i686 / Linux_x86_64 / -->
        </applicationInfo>                        <!-- Macintosh / Java -->

        <executionInfo>
                <executionType>regular</executionType> <!-- regular / bsp / parametric -->
                <userName>rafaelf</userName>
                <executingHost>guaguin</executingHost>
                <appArgs>50</appArgs>
                <appMainRequestId>1</appMainRequestId>
                <appNodeRequestId>0</appNodeRequestId>
                <appConstraints>osName == 'Linux'</appConstraints>
                <appPreferences>max(freeRAM)</appPreferences>
                <executionState>running</executionState> <!-- running / finished -->

                <outputFiles>
                        <file>stdout</file>
                        <file>stderr</file>
                        <file>out.dat</file>
                </outputFiles>

                <inputFiles>
                        <file>in.dat</file>
                </inputFiles>

                <initialTimestamp>99999999</initialTimestamp>
                <finishTimestamp>99999999</finishTimestamp>
        </executionInfo>
</application>
```

Figura 2.16: Exemplo de arquivo XML utilizado pelo *ExecutionManagementAgent*

/ software suportada pela aplicação. Na *executionInfo* são armazenados o tipo de execução (regular, paramétrica ou BSP), o nome do usuário que originou a requisição, o nome da máquina em que a aplicação está executando atualmente, os argumentos passados à aplicação, os identificadores da requisição (*appMainRequestId* e *appNodeRequestId*), as restrições e preferências da aplicação, o seu estado atual de execução (*running* ou *finished*), a lista dos arquivos de entrada e saída e os tempos de início e término da execução.

AgentRecover

Um dos grandes desafios a ser superado no desenvolvimento de um *middleware* de grade, é a disponibilização de mecanismos de tolerância a falhas. Grades são mais suscetíveis à falhas que plataformas computacionais tradicionais pois agregam, potencialmente, milhares de recursos, serviços e aplicações, que necessitam interagir de forma a tornar possível o uso da grade como uma plataforma de execução de aplicações [MCBS03]. No MAG, este problema ainda se agrava pelo fato de os aglomerados serem compostos por nós não dedicados, não estando em um ambiente controlado, a exemplo do que acontece com os *clusters* computacionais.

O *AgentRecover* gerencia o processo de recuperação de nós da grade. Ele é instanciado sob-demanda pelo *ExecutionManagementAgent*, caso este seja informado da ocorrência da falha de um nó. Após recuperar todos os agentes MAG que executavam no nó que falhou, o *AgentRecover* é destruído.

Ao ser instanciado, o *AgentRecover* recebe do *ExecutionManagementAgent* informações relativas às aplicações que executavam no nó que falhou, necessárias ao processo de recuperação. Em seguida o *AgentRecover* solicita ao GRM o reescalonamento destas aplicações e recria os agentes que falharam em outros nós. Estes agentes passam então a conduzir o processo de recuperação das aplicações de maneira autônoma, reiniciando-as a partir de seus últimos *checkpoints* (obtidos junto ao *StableStorage*).

O diagrama SASD que mostra a estrutura do *AgentRecover* é apresentado na Figura 2.17. Cada classe que compõe este diagrama será detalhada em seguida.

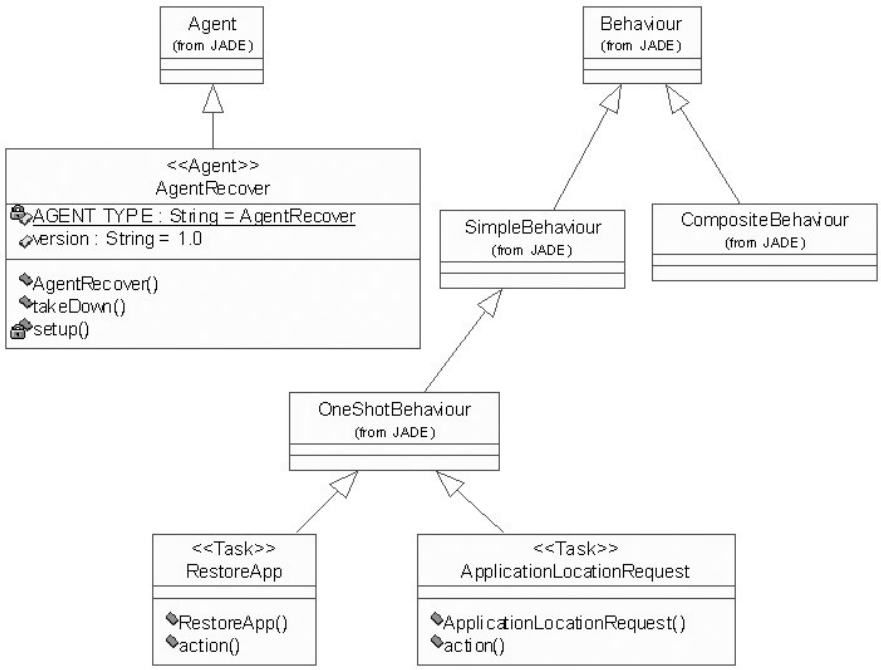


Figura 2.17: Diagrama de definição de estrutura do agente *AgentRecover*

- *AgentRecover:* classe principal do agente. Responsável por inicializar o agente e seus comportamentos;
- *ApplicationLocationRequest:* comportamento que solicita ao GRM, o reescalonamento das aplicações que executavam no nó que falhou;
- *RestoreApp:* uma vez definidos os nós que receberão as aplicações sendo recuperadas, este comportamento passa a solicitar aos *AgentHandlers* localizados nestes nós,

a recriação dos agentes que falharam e de suas computações;

- *QueryExecutionInfo:* comportamento responsável por solicitar ao *ExecutionManagementAgent* a consulta a informações relativas às execuções de aplicações que falharam juntamente com um nó da grade. Estas informações servirão como base para o seu processo de recuperação.

StableStorage

Para prover tolerância a falhas a aplicações, o MAG utiliza a abordagem de *checkpointing*. Esta técnica consiste na salva periódica dos estados de execução das aplicações. A partir destes estados a execução das aplicações podem ser recuperadas, fazendo com que as aplicações continuem suas execuções do ponto onde pararam antes da ocorrência da falha. Assim, passa a ser necessária a existência de um *armazém estável*, ou seja, um repositório de dados que sobrevive às eventuais falhas do sistema. Este repositório é o responsável por armazenar o *checkpoint* das aplicações que executam na grade. No MAG, o agente *StableStorage* é a implementação do armazém estável de *checkpoints*. Para evitar a falha deste agente é necessário que o mesmo execute no nó gerenciador do aglomerado. A implementação do *StableStorage* é apresentada através de seu diagrama SASD, mostrado na Figura 2.18.

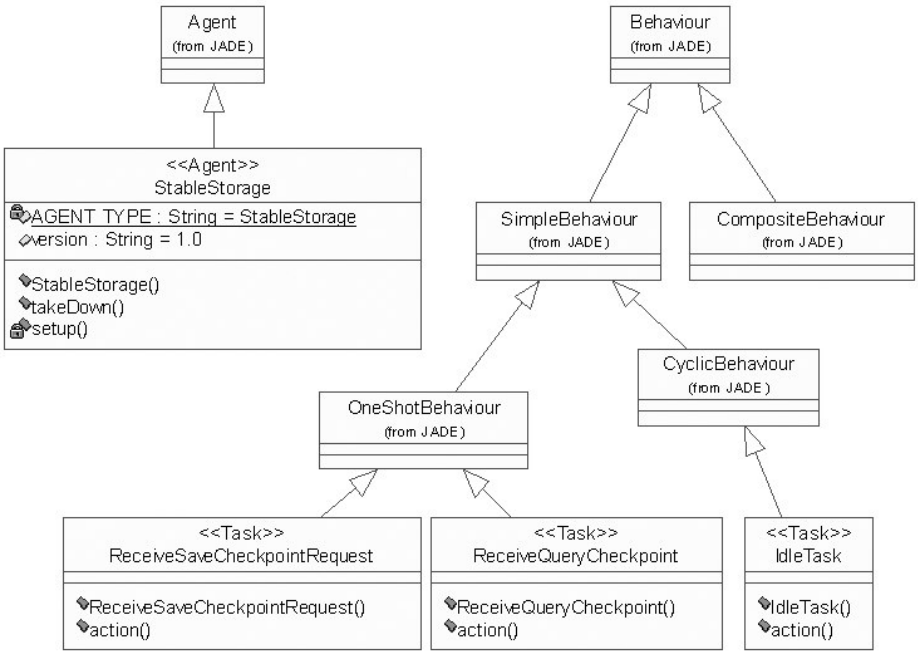


Figura 2.18: Diagrama de definição de estrutura do agente *StableStorage*

- *StableStorage:* classe principal do agente. Responsável por inicializar o agente e seus comportamentos;

- *IdleTask:* comportamento cíclico responsável pelo recebimento de mensagens originadas por outros agentes da grade. Ao receber uma mensagem, verifica o tipo de ação ou predicado associado a ela e aciona o comportamento responsável por seu tratamento.

- *ReceiveSaveCheckpointRequest* e *ReceiveCheckpointQuery:* comportamentos responsáveis por tratar solicitações de salva e recuperação de *checkpoints* armazenados no *StableStorage*;

AgentHandler

O *AgentHandler* é o único componente de software implementado na arquitetura do MAG. Ele executa em todos os nós que fornecem poder computacional à grade, sendo o responsável por manter a plataforma de agentes em execução no nó e por instanciar agentes MAG em resposta às solicitações de execução. Interage com os componentes do Integrade, tornando o paradigma de agentes transparente a ele.

A metodologia *Agil PASSI* não permite através de seus diagramas, representar componentes de software. Assim, esta seção apresenta a interface CORBA disponibilizada pelo *AgentHandler*. A Figura 2.19 apresenta a IDL (*Interface Definition Language*) [Obj02a] do componente *AgentHandler*.

- `executeApp`: este método é invocado pelo LRM da máquina em que executa para repassar uma requisição de execução de aplicação Java ao MAG. As informações necessárias à execução estão divididas em dois objetos distintos, cujas estruturas são definidas na IDL do Integrade [Gol04]: `commonSpecs` contém informações relevantes a todos os nós da aplicação, independente do tipo de execução (regular ou paramétrica), como o identificador da aplicação no repositório de aplicações; já o `distinctSpecs`, contém informações específicas a apenas um nó, como por exemplo, os argumentos que serão utilizados por uma cópia de uma aplicação paramétrica sendo executada em um determinado nó da grade. As tabelas 2.1 e 2.2 descrevem os campos das estruturas de cada um destes objetos;

```
interface AgentHandler {}
   void executeApp (in types::CommonExecutionSpecs commonSpecs,
                    in types::DistinctExecutionSpecs distinctSpecs);
   void releaseNode ();
   void restoreApp (in string appExecutionId, in string applicationName,
                    in types::CommonExecutionSpecs commonSpecs,
                    in types::DistinctExecutionSpecs distinctSpecs);
   types::FileSeq requestOutputFiles (in string appId);
}
```

Figura 2.19: Interface IDL do componente *AgentHandler*

- `releaseNode`: este método é invocado pelo LRM quando recebe uma requisição de liberação de nó. Isto significa que o usuário da máquina deseja ter a posse exclusiva de seus recursos, não mais cedendo-os à grade. Uma vez recebida esta requisição, o *AgentHandler* solicita ao GRM o reescalonamento das aplicações em execução em seu nó e passa a solicitar aos agentes MAG, a migração de suas computações para outras localidades;

- `restoreApp`: este método tem sua assinatura muito semelhante à do método `executeApp`, diferindo apenas no fato de que, além das opções de execução (`commonSpecs` e `distinctSpecs`), são também passados como parâmetros o nome e o identificador de execução da aplicação antes da ocorrência da falha. Sua função é restaurar um agente que tenha falhado. Este método é invocado pelo *AgentRecover* quando está procedendo a recuperação das aplicações que executavam em um nó que tenha falhado. Ao ser invocado, o método `restoreApp` instancia um novo *MagAgent*, e informa a este que ele deverá recuperar uma dada computação que tenha falhado. Este agente do MAG passa então a recuperar sua aplicação de maneira autônoma;

- `requestOutputFiles`: após receber uma notificação de término de execução, o ASCT deve então coletar os arquivos de saída gerados pela aplicação, de forma a devolvê-los ao usuário solicitante. Este procedimento é realizado através deste método.

CommonExecutionSpecs	
Campo	**Descrição**
requestingAsctIor	IOR[6]do ASCT que solicitou a execução
grmIor	IOR do GRM que enviou a requisição
deniedExecution	Lista dos LRMs que já recusaram a requisição de execução
applicationId	Identificador da aplicação no Repositório de Aplicações
applicationConstraints	Expressão em TCL denotando requisitos indispensáveis da aplicação
applicationPreferences	Expressão em TCL denotando requisitos preferenciais da aplicação

Tabela 2.1: Campos da estrutura CommonExecutionSpecs

DistinctExecutionSpecs	
Campo	**Descrição**
asctRequestId	Identificador da requisição atribuído pelo ASCT requisitante
applicationArgs	Argumentos de linha de comando passados à aplicação
outputFiles	Lista de arquivos de saída da aplicação

Tabela 2.2: Campos da estrutura DistinctExecutionSpecs

ClusterApplicationViewer

O *ClusterApplicationViewer* [GCLS05] é uma ferramenta gráfica que disponibiliza informações a respeito das execuções ocorridas na grade. A análise destas informações permite aos desenvolvedores da infra-estrutura de grade e aos administradores das mesmas a tomada de decisões relativas ao projeto do *middleware* da grade e a configuração do ambiente de execução, respectivamente.

Uma das preocupações no desenvolvimento desta ferramenta é que deve ser assegurado ao usuário que a informação exibida através da interface esteja sendo apresentada de forma correta. Para tanto, o padrão arquitetural MVC (*Model-View-Controller*) [Gol90, Bur92] foi aplicado na modelagem desta ferramenta. O uso do padrão MVC colabora no sentido de manter as informações apresentadas ao usuário sincronizadas com os dados das execuções de aplicações na grade.

O *ClusterApplicationViewer* solicita consultas periodicamente ao *ExecutionManagementAgent* sobre as aplicações em execução na grade. A partir destas informações é possível manter a interface com o usuário atualizada. O diagrama SASD do *ClusterApplicationViewer* é apresentado na Figura 2.20.

[6]Uma IOR (*Interoperable Object Reference*) identifica unicamente um objeto CORBA, permitindo a realização de invocações remotas a este objeto.

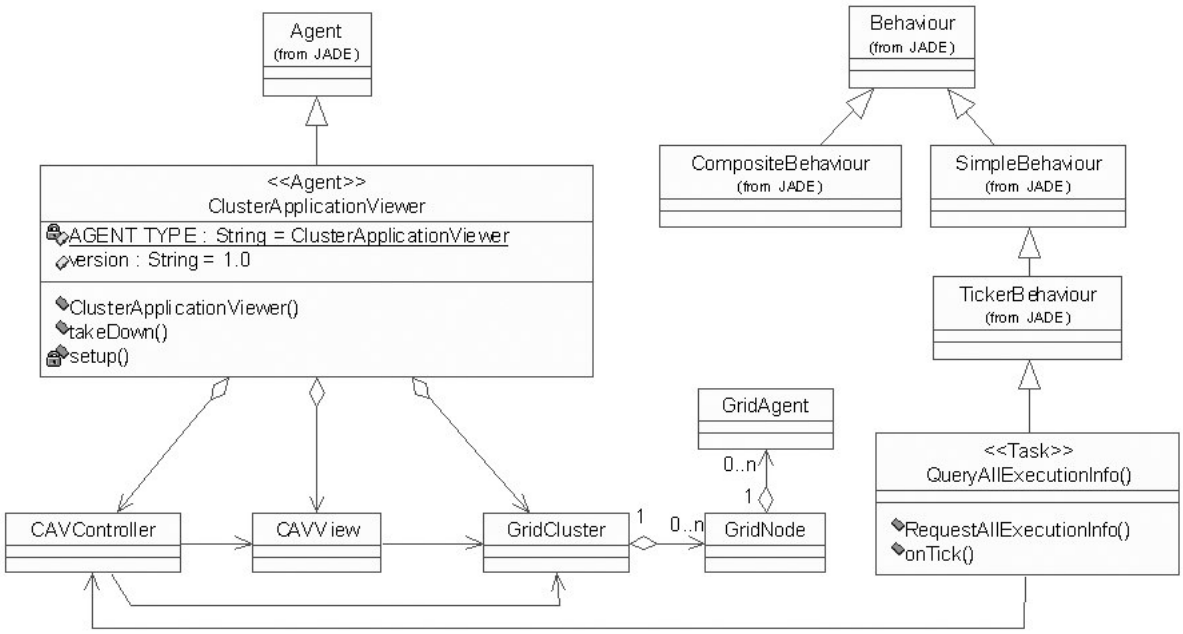


Figura 2.20: Diagrama de definição de estrutura do agente *ClusterApplicationViewer*

- *ClusterApplicationViewer:* classe principal do agente. Estende a classe *GuiAgent* do JADE, que dá ao agente a capacidade de incorporar uma interface gráfica;
- *GridCluster*, *GridNode* e *GridAgent:* classes responsáveis por armazenar informações sobre a grade. Cada uma destas classes guarda informações sobre uma entidade da grade: informações de um aglomerado, de um nó e de um agente, respectivamente;
- *CAVView:* classe de interface gráfica da ferramenta;
- *CAVController:* esta classe trata todos os eventos da interface gráfica e efetua atualizações necessárias nos dados da aplicação, armazenados em objetos das classes *GridCluster*, *GridNode* e *GridAgent*;
- *QueryAllExecutionInfo* comportamento responsável por coletar junto ao *ExecutionManagementAgent* as informações sobre o estado de todas as aplicações que executam na grade.

A Figura 2.21 ilustra a interface gráfica do *ClusterApplicationViewer*, implementada com o *toolkit* gráfico *Swing*, nativo do Java.

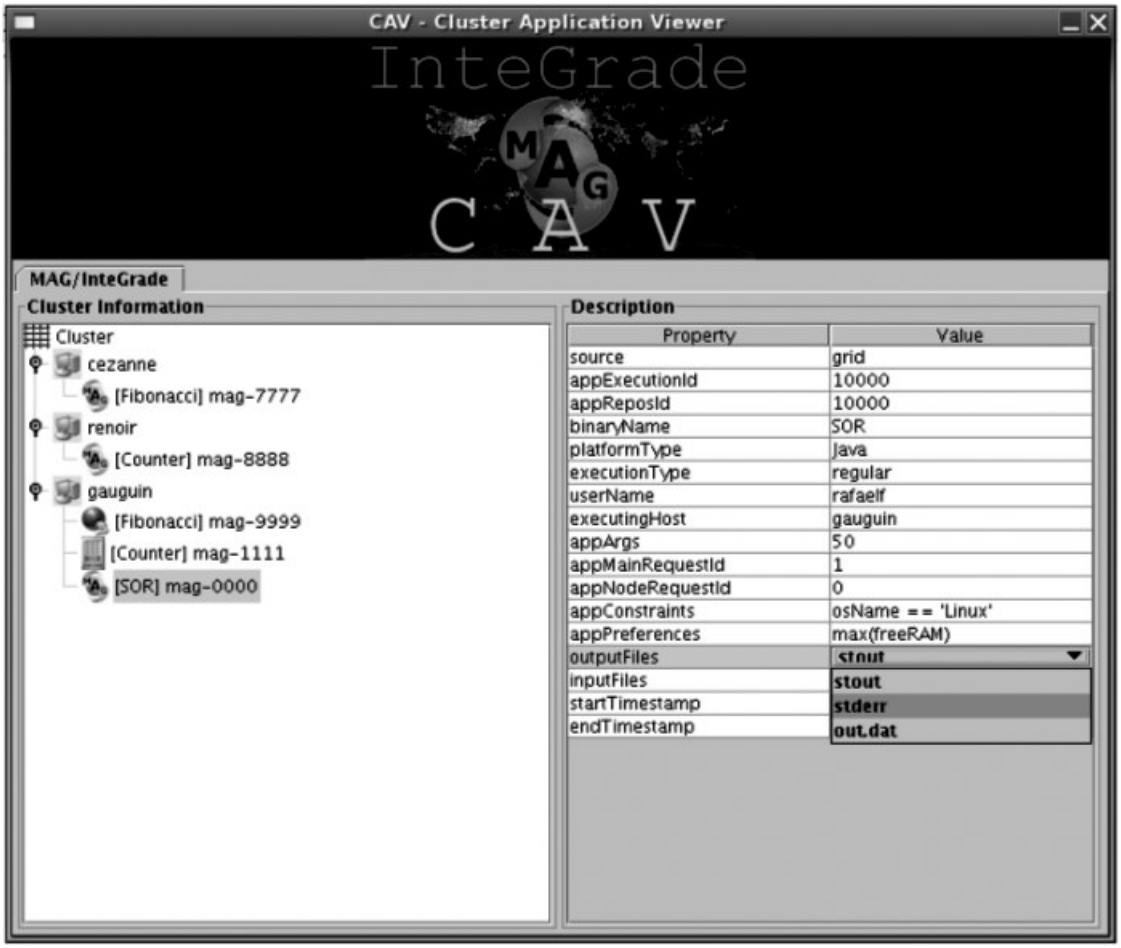


Figura 2.21: Interface gráfica do *ClusterApplicationViewer*

2.4.4 Diagrama de descrição de comportamento multiagente (MABD)

O diagrama de descrição de comportamento multiagente (MABD) expressa o fluxo de execução e comunicação dos agentes do sistema, através da interação entre seus agentes. É representado através de um diagrama de atividades UML. Eestes diagramas são divididos em *swimlanes*, que correspondem aos comportamentos dos agentes, enquanto que as atividades correspondem aos métodos destes comportamentos. As setas entre atividades indicam alguma forma de comunicação. Esta comunicação pode ser de um dos seguintes tipos: `invoke` (invocação de método), `instantiate` (instanciação de agente), `newTask` (instanciação de comportamentos de um agente) e `message` (envio de mensagens entre agentes).

Dado que a arquitetura do MAG é composta não somente por agentes, mas também por componentes de software, algumas modificações neste diagrama passam a

ser necessárias: para representar o fluxo de execução e comunicação de componentes de software, as atividades apresentam-se como métodos, enquanto que as *swimlanes* passam a representar os próprios componentes.

Três cenários serão apresentados através do diagrama MABD: (a) submissão de aplicações para execução na grade, (b) liberação de nós (caso o usuário de uma máquina requisite seu uso exclusivo), e (c) a recuperação de nós na ocorrência de uma falha.

Submissão de execução

Neste cenário um usuário utiliza a interface gráfica disponibilizada pelo componente ASCT para solicitar a execução de uma aplicação à grade. Este diagrama ilustra a cooperação entre os agentes do MAG e os componentes do Integrade para a execução de aplicações. O diagrama MABD deste cenário, devido ao seu grande tamanho, foi dividido em duas partes, apresentadas nas figuras 2.22 e 2.23.

Um usuário requisita a execução de uma aplicação através da interface do ASCT (a aplicação deve ter sido previamente registrada no repositório de aplicações). O ASCT então encaminha esta requisição ao GRM (1), que executa seu algoritmo de escalonamento de forma a selecionar um nó disponível. O GRM então repassa a requisição ao LRM do nó escalonado (2). Entretanto, se nenhum nó atender os requisitos de execução da aplicação, o GRM então responde ao ASCT informando que sua solicitação de execução foi negada (4). Caso a aplicação a ser executada seja nativa do sistema operacional o LRM executa a aplicação sem nenhuma intervenção por parte do MAG. Entretanto, se esta for uma aplicação escrita usando a linguagem Java, a requisição é encaminhada ao *AgentHandler* local (3). Este *AgentHandler* então informa ao ASCT que aceitou o seu pedido de execução (5), repassando a ele sua IOR CORBA. Em seguida o *AgentHandler* instancia um *MagAgent* responsável por executar a aplicação do usuário (7), que prepara o ambiente para a execução da aplicação baixando seus arquivos de entrada de dados (6) e seu *bytecode* (8, 9), salvando-os no sistema de arquivos local.

Em seguida, o *MagAgent* registra os dados relativos à execução de sua aplicação junto ao *ExecutionManagementAgent* (10, 11). Ao receber esta solicitação, o *ExecutionManagementAgent* a trata (12), armazenando os dados relativos à execução da aplicação. O *MagAgent* então instancia a aplicação e inicia sua execução em uma *thread* independente (13). Após o início da execução da computação, o *MagAgent* passa a monitorá-la,

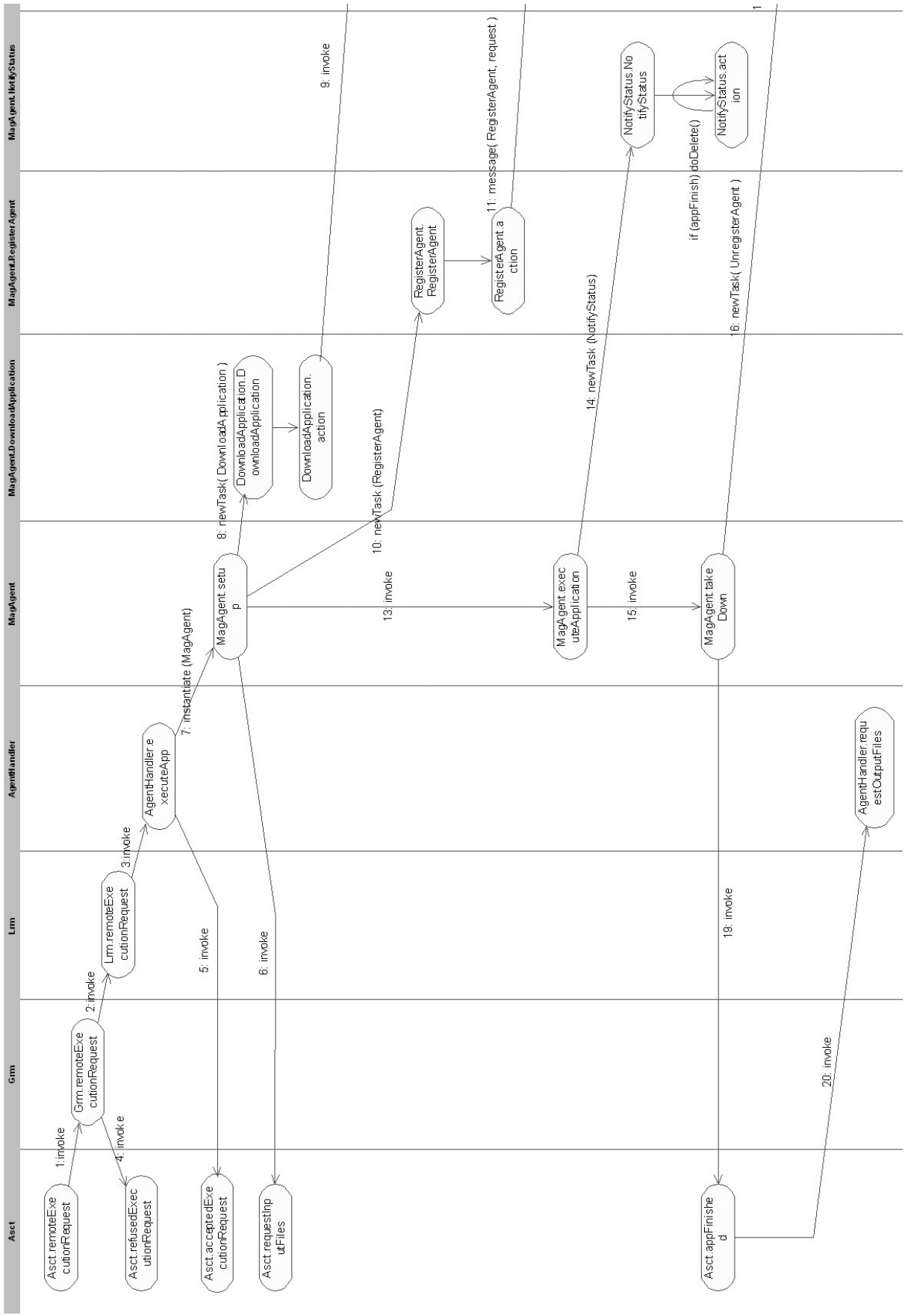


Figura 2.22: Diagrama MABD do MAG (parte 1) – Submissão de Aplicações

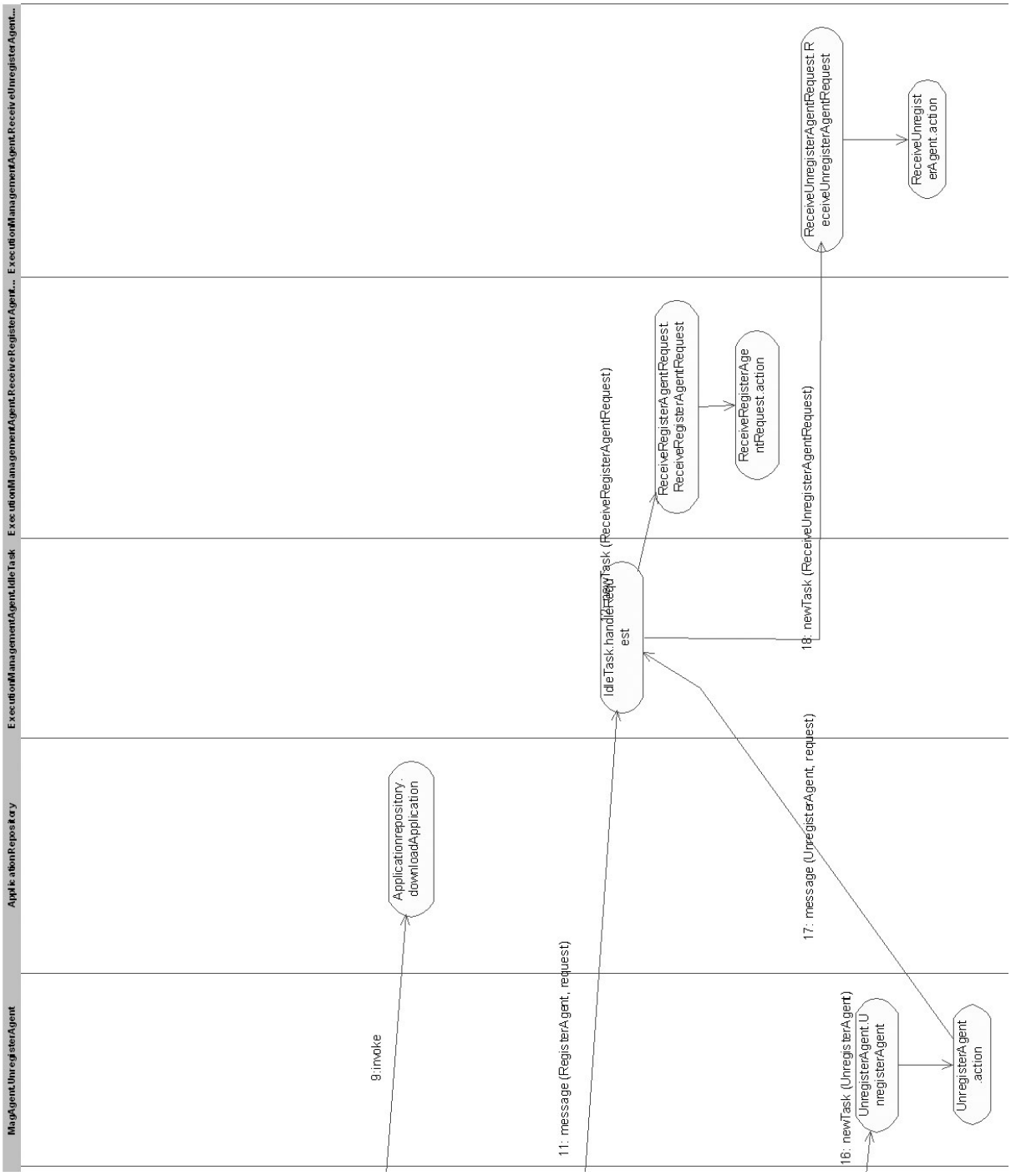


Figura 2.23: Diagrama MABD do MAG (parte 2) – Submissão de Aplicações

aguardando por seu término (14). Ao perceber o fim da execução da aplicação, o *MagAgent* passa a executar suas rotinas de término (15), solicitando a remoção do registro dos dados da execução junto ao *ExecutionManagementAgent* (16, 17, 18) e notificando o ASCT que originou a requisição sobre o término da execução da aplicação solicitada (19). Em seguida, o ASCT requisita os resultados da computação ao *AgentHandler* no qual a execução da aplicação terminou (20), exibindo por fim, estes dados a seu usuário.

Liberação de nós

Um dos objetivos de projeto do MAG é explorar o tempo ocioso de máquinas que compreendam a infraestrutura da grade. Assim, se um usuário requisitar o uso exclusivo de uma destas máquinas, ela deverá ser liberada de forma a não prejudicar a sua utilização. Neste caso, as computações da grade em execução no nó devem ser migradas para outras localidades. A Figura 2.24 ilustra o mecanismo de liberação de nós do MAG.

Após ser informado da intenção do usuário de utilizar sua máquina, o LRM local solicita ao GRM que remova as informações relativas a seus recursos (1). Isto tem o intuito de não permitir que novas computações sejam escalonadas para execução naquele nó. O LRM então notifica o *AgentHandler* local sobre a solicitação do usuário, informando que as computações executadas pelo MAG devem ser migradas (2). Em seguida, o *AgentHandler* requisita ao GRM que escalone outros nós para continuar a execução da aplicações que serão migradas (3). A seleção dos nós é baseada em vários critérios como disponibilidade de recursos (CPU, memória e espaço em disco) e preferências / restrições impostas pelo usuário que solicitou a execução da aplicação. O *AgentHandler* então solicita a migração de cada *MagAgent* em execução na máquina (4). Cada *MagAgent* migra[7] para o nó de destino juntamente com sua aplicação (5). Após o término do processo de migração, o *MagAgent* modifica a informação relativa a sua localização junto ao *ExecutionManagementAgent* (6, 7).

[7]A migração das aplicações envolve a captura do estado de execução da aplicação na origem, e o seu reestabelecimento no destino. Isto é possível através de mecanismos específicos, descritos no capítulo 3

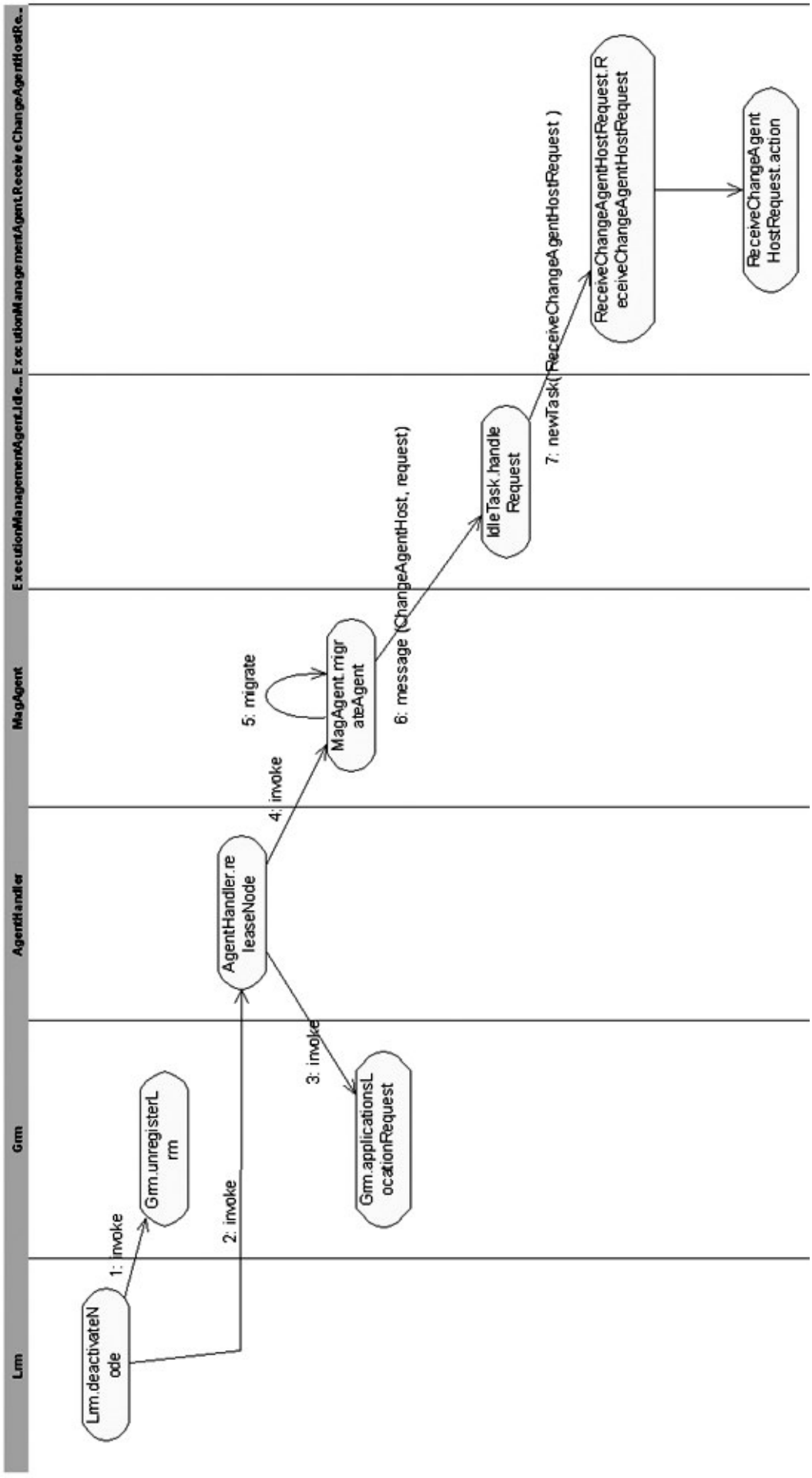


Figura 2.24: Diagrama MABD do MAG – Liberação de nós

Recuperação de nós

A implementação atual do MAG provê uma infraestrutura básica para fornecer tolerância a falhas às aplicações que executam na grade, tratando apenas falhas de colapso ocorram em nós da grade. Futuramente pretende-se estender este modelo para fornecer suporte à outros modelos de falhas.

Cada um dos componentes da infraestrutura de tolerância a falhas do MAG foi desenvolvido como um agente que juntos compõem uma sociedade de agentes responsável por prover suporte a falhas de nós da grade. A Figura 2.25 mostra o diagrama MABD do MAG para o cenário de recuperação da falha de um nó da grade.

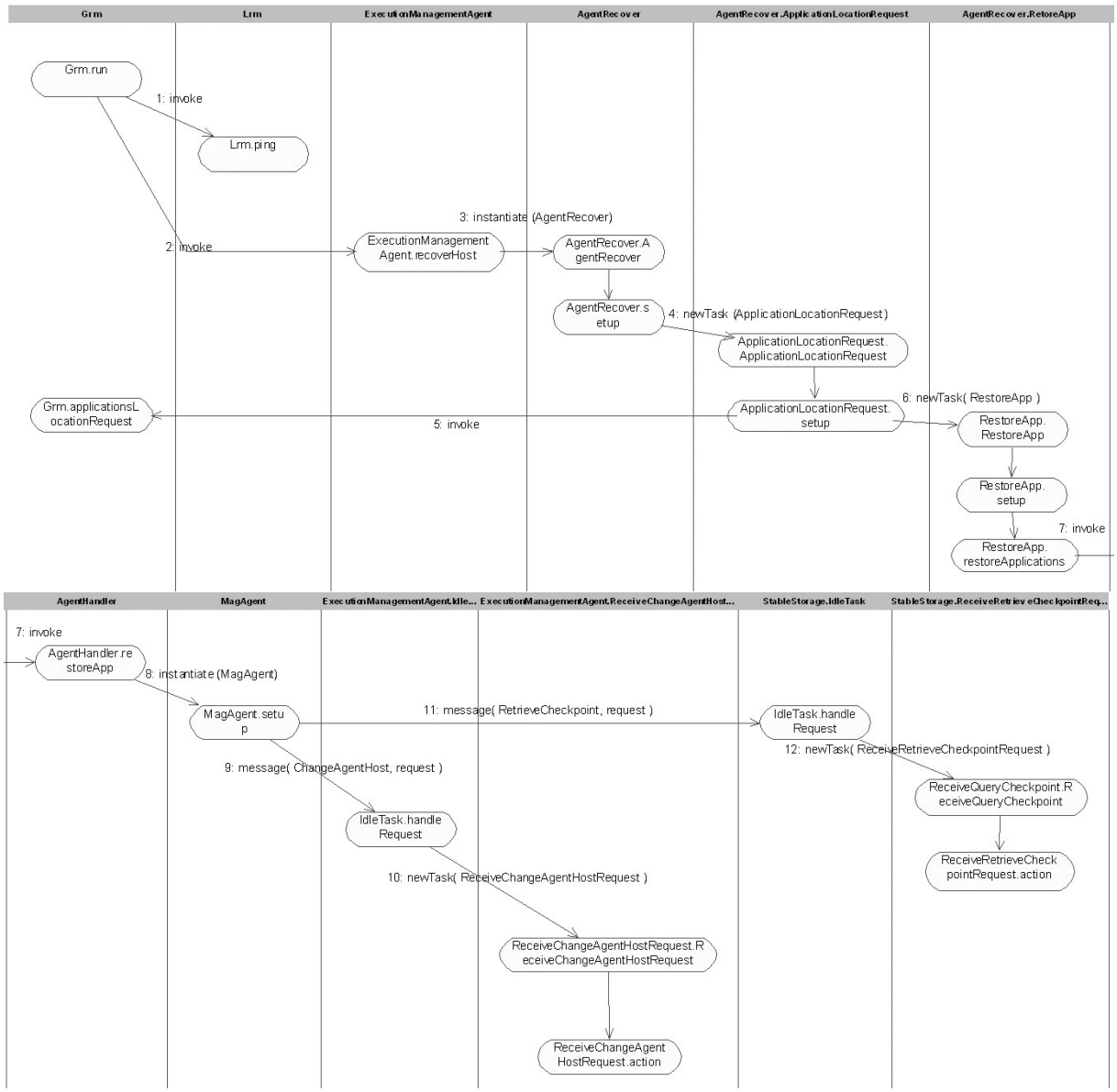


Figura 2.25: Diagrama MABD do MAG – Recuperação de nó

O GRM recebe periodicamente dos LRMs alocados nos nós da grade, informações relativas à disponibilidade de seus recursos. Caso o GRM não receba estas informações de um dos LRMs durante um certo intervalo de tempo, ele então consulta este LRM para descobrir se o mesmo permanece ativo (1). Se este não puder ser contactado a ocorrência de uma falha é detectada e o GRM então solicita ao *ExecutionManagementAgent* a recuperação do nó que falhou (2). Em seguida, o *ExecutionManagementAgent* instancia um novo `AgentRecover` (3), responsável por tratar a falha detectada. O `AgentRecover` solicita ao GRM que reescalone as aplicações que falharam em novas localidades (4) e solicita a recuperação de seus respectivos agentes ao *AgentHandler* presente em cada nó escalonado (5). Assim, cada *AgentHandler* inicia a recriação dos agentes que falharam (6), informando a eles quais aplicações deverão recuperar. Cada agente MAG fica então responsável por conduzir a recuperação de sua aplicação de maneira autônoma. Após preparar o ambiente de execução da aplicação (baixando seu *bytecode* e seus arquivos de entrada), os agentes MAG informam ao *ExecutionManagementAgent* sobre suas novas localizações na grade (7, 8) e recuperam a execução das aplicações a partir do último *checkpoint* salvo no *StableStorage* (9, 10).

3 Mecanismo de Migração Forte

Mobilidade de código é definida como a capacidade de uma aplicação ser movida entre diferentes localidades de uma rede e continuar sua execução transparentemente a partir do ponto anterior à ocorrência da migração. Ela pode ser explorada no contexto de grades computacionais para prover balanceamento de carga e suporte a nós não dedicados (i.e. caso um usuário solicite o uso de um recurso, todas as computações que nele executam devem ser migradas para outras localidades, de forma a não prejudicar sua utilização).

Este capítulo apresenta o MAG/Brakes, um arcabouço do MAG que permite a migração de aplicações na grade. Inicialmente são descritos conceitos, uma classificação e os desafios relativos à mobilidade de código. O arcabouço MAG/Brakes segue a licença AFL[1] (*Academic Free License*), licença herdada de seu antecessor Brakes. Todo o código-fonte do MAG/Brakes foi desenvolvido utilizando a linguagem Java e é composto por 12999 linhas de código.

3.1 Mobilidade de Código

Carzaniga et al. [CPV97] define mobilidade de código como a capacidade de mudar dinamicamente as ligações entre o código e a localização onde executa. O termo *migração de código* também é utilizado para definir este procedimento, e será adotado no decorrer deste capítulo. São várias as razões que motivam o uso de migração de processos. Entre elas encontram-se as seguintes [TS03]:

- *Melhoria do desempenho de aplicações* – esta é a principal motivação para o uso de migração. A idéia básica é melhorar o desempenho geral do sistema através da migração de processos que executam em máquinas sobrecarregadas para máquinas com pouca carga[2]. Os algoritmos de balanceamento de carga exercem um impor-

[1] Licença disponível em: `http://www.opensource.org/licenses/academic.php`

[2] Carga é muitas vezes expressa em termos do tamanho da fila ou da utilização do processador, mas outros indicadores de desempenho podem ser utilizados

tante papel neste contexto, dado que é através deles que todas as decisões relativas à alocação e redistribuição de tarefas entre um conjunto de processadores são tomadas, de forma a otimizar a capacidade computacional do sistema;

- *Diminuição da carga na rede* – esta motivação encontra sua base no uso de heurísticas que permitam identificar em que situações é interessante migrar aplicações até os dados ou os dados até as aplicações. Por exemplo, uma aplicação cliente-servidor em que muitos dados são mantidos do lado do servidor. Um cliente eventualmente poderia precisar efetuar muitas operações (e envolvendo grandes volumes de dados) junto a este servidor. Neste caso, pode ser interessante mover dinamicamente parte do código do cliente até o servidor, economizando tempo e largura de banda;

- *Exploração de processamento paralelo* – diversas cópias da mesma aplicação podem ser migradas para localidades remotas, de forma a efetuar operações paralelamente. Um exemplo desta aplicação é a procura de informações da web no qual o mecanismo de busca seja formado por várias aplicações que movem-se através de diferentes sítios na web efetuando a procura requisitada;

- *Extensão dinâmica da funcionalidade de aplicações* – na abordagem tradicional, a construção de aplicações distribuídas é fundamentada no particionamento de uma aplicação em unidades menores que trocam mensagens através da rede para realizar computações. Entretanto, se o código das aplicações tiver a capacidade de mover-se através da rede, é possível para o sistema distribuído configurar-se dinamicamente. Como exemplo tem-se o caso de um cliente que acessa sistemas de arquivos remotos através de protocolos proprietários. Neste caso, o código responsável pelo acesso a um determinado tipo de sistema de arquivos poderia ser instanciado e migrado sob-demanda para a máquina em que este sistema de arquivos se encontra;

- *Execução assíncrona e autônoma* – o código móvel pode encapsular tarefas que executem de forma independente do processo que o criou. Isto torna o código móvel autônomo e assíncrono, podendo estas características ser aproveitas, por exemplo, para o desenvolvimento de aplicações que executem em dispositivos de computação portátil. Neste caso, a conexão de rede entre o dispositivo e a rede fixa pode muitas vezes ser frágil e cara. O uso de um código móvel que migre do dispositivo para uma máquina na rede fixa, realize a computação e depois volte para o dispositivo móvel apresenta-se como uma solução econômica e prática.

3.1.1 Classificação

Para efetuar a migração de um processo, algumas informações de estado (também chamadas de *contexto*) devem ser salvas e transportadas para sua nova localização. Ao chegar no destino, o processo é reiniciado a partir do estado transportado. Entretanto, para migrar um processo é necessário conhecer o que exatamente compreende a computação e seu estado.

Fuggetta et al. [FPV98] descreve que um processo consiste basicamente de três segmentos: *segmento de código*, *segmento de recursos* e *segmento de execução*. O segmento de código é a parte que contêm o conjunto de instruções que compõem o programa. O segmento de recursos contêm dados e referências a recursos externos utilizados pelo processo, como arquivos, impressoras, dispositivos e outros processos. Finalmente, o segmento de execução armazena o estado de execução do processo, consistindo de dados privados, pilha de chamadas e contador de instrução. A Figura 3.1 ilustra a migração de um processo e de seus segmentos. Antes de migrar, a aplicação deve descobrir se existem recursos alocados a si. Em caso positivo, ele deve armazenar em seu segmento de recursos as informações relativas a seus recursos, de forma que seja possível, após a migração, voltar a utilizá-los. A migração de recursos será apresentada de uma forma mais detalhada na seção 3.1.2. Após armazenar as informações relativas aos recursos, o processo deve então, salvar seu estado de execução. A partir deste estado (armazenado em seu segmento de execução) é possível ao processo reiniciar sua execução do ponto onde parou na origem. Por fim, o código do processo (seu segmento de código) e os segmentos de execução e recurso são serializados e transmitidos para outra localidade na rede, onde serão restaurados através do processo inverso.

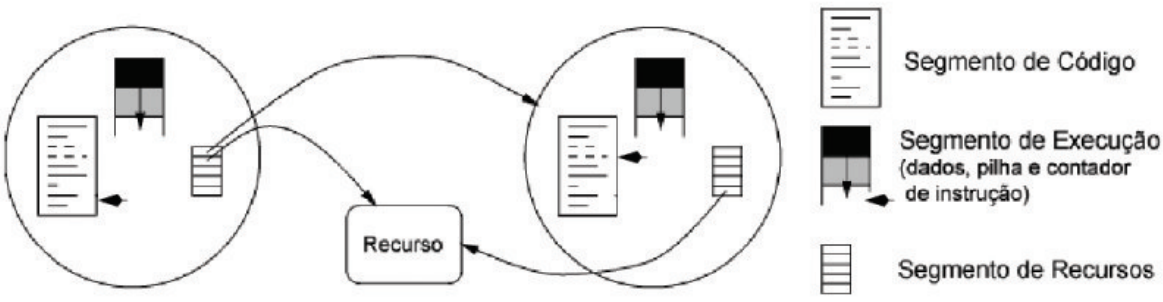


Figura 3.1: Migração das estruturas internas de um processo

Um mecanismo que migre estes três segmentos e reinicia o processo exatamente

no mesmo estado e na mesma posição de código em que ele estava antes da migração é chamado *transparente* ou caracterizado como de *migração forte* [Fun98]. A migração forte provê ao processo a abstração de que a execução não foi interrompida. A migração forte é muito poderosa, porém sua implementação é complexa.

A migração *não transparente* (também chamada de *migração fraca*) pode ser definida como toda migração que não é considerada forte [CWHB03]. Na migração fraca somente o segmento de código é transferido, e talvez alguns dados de inicialização. Um atributo característico da migração fraca é que o programa transferido é sempre executado a partir de seu estado inicial.

A migração forte simplifica a tarefa do programador, uma vez que ele não precisa implementar este processo explicitamente: o estado do processo é salvo automaticamente. Esta característica é básica para o desenvolvimento de muitos mecanismos em sistemas distribuídos, como os de tolerância a falhas [EAWJ96] e de persistência de objetos [Sil97]. Já na abordagem da migração fraca, o programador se vê obrigado a criar o código responsável por salvar e recuperar o estado completo do processo.

Existem ainda sub-classificações dos tipos de migração apresentados. Independente se o modelo aceita migração forte ou fraca, uma distinção pode ser feita entre a migração *iniciada pelo emissor* e a *iniciada pelo receptor* [TS03]. No caso em que a migração é iniciada pelo emissor (também chamada *proativa*) o processo de migração é iniciado na máquina onde o código reside e está sendo executado, ou seja, a aplicação deixa o local onde está executando e migra para uma outra localidade na rede por iniciativa própria (e.g. programas de busca em bases de dados na web). Já na migração iniciada pelo receptor (também chamada *reativa*), a iniciativa da migração parte de uma aplicação executando na máquina de destino (e.g. *applets* Java). A Figura 3.2 mostra as alternativas de tipos de migração apresentados.

Figura 3.2: Alternativas para migração de código

Chakravarti et al. [CWHB03] ampliou o leque de classificações relativas aos tipos de migração. Ele definiu termo *migração forçada* para expressar a habilidade do sistema em suportar a migração de um processo a qualquer momento, mediante a solicitação de uma entidade externa (e.g. uma *thread*). Esta capacidade é particularmente útil para fornecer determinados serviços, como por exemplo, o balanceamento de carga de aplicações.

3.1.2 Processo de Migração Forte

Migrar aplicações é um processo complexo. Vários passos estão envolvidos no processo de migração de aplicações entre máquinas em uma rede [CLZ00, NIR04]. Estes passos são ilustrados na Figura 3.3 e detalhados na seqüência:

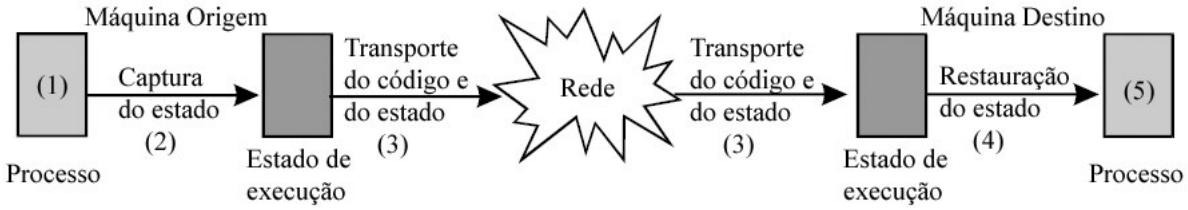


Figura 3.3: Etapas do processo de migração

1. **O fluxo de execução é interrompido:** através de mecanismos providos pela linguagem ou pelo sistema operacional, o fluxo de execução do processo é interrompido;

2. **O estado da entidade migrada é capturado:** todos os dados necessários para reiniciar a execução da aplicação do mesmo ponto onde foi interrompida devem ser capturados. Entretanto, a captura destes dados é um processo complexo e pode envolver vários problemas. Por exemplo, uma das técnicas utilizadas para efetuar a migração de uma aplicação é copiar todo o seu contexto de execução a partir de seu espaço de endereçamento na memória. Neste caso, o algoritmo responsável por fazer a captura do estado de execução da aplicação pode ficar limitado a algumas plataformas de hardware e software. Isto porque cada plataforma apresenta um modelo próprio de gerenciamento de memória, afetando a implementação deste mecanismo. Um outro problema relacionado a esta etapa está em manter as referências

aos recursos[3] utilizados pela aplicação no momento da migração. As alternativas existentes para tratar este problema são apresentadas na Figura 3.4 e descritas a seguir:

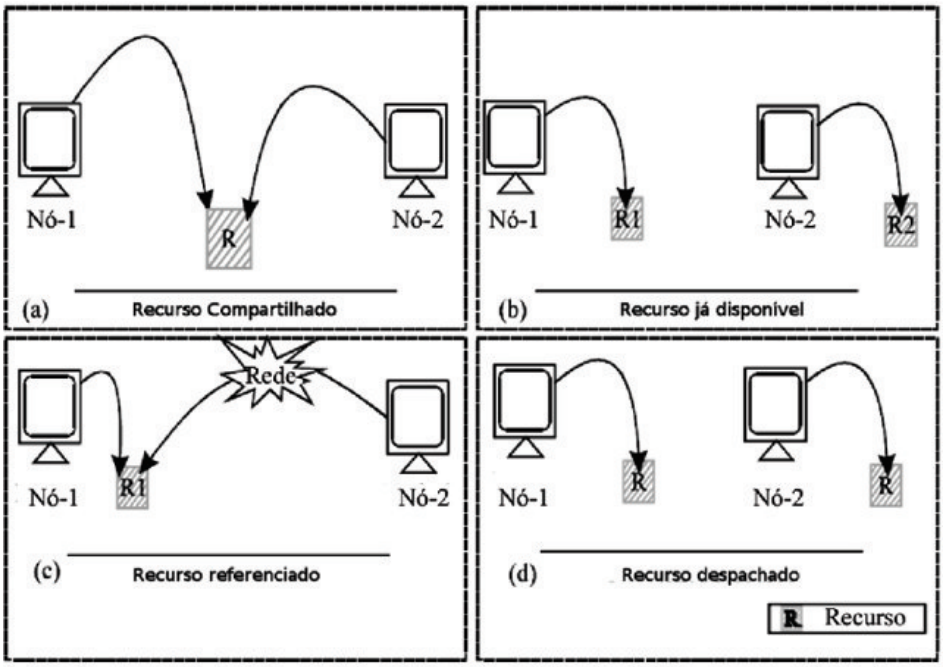


Figura 3.4: Migração de recursos

(a) *Recurso compartilhado:* se o recurso for um recurso compartilhado, após o término do processo de migração o acesso ao recurso continua disponível (e.g. uma impressora de rede);

(b) *Recurso já disponível:* ao migrar uma aplicação pode-se acreditar que o recurso utilizado também esteja disponível na nova máquina. Por exemplo, uma aplicação que gere uma série de arquivos de saída em uma unidade de fita. Se, após a migração, a nova máquina na qual a aplicação executa também tenha disponível uma unidade de fita, a aplicação continuará a gerar os arquivos de saída no novo dispositivo, sem fazer distinção entre o dispositivo original e o dispositivo disponível em sua nova localidade;

(c) *Recurso referenciado:* nesta abordagem, a referência ao recurso utilizado é migrada juntamente com o seu estado e seu código. Assim, ao chegar em seu destino, ele deve tentar se reconectar através da rede ao recurso na antiga

[3]Recursos aparecem na forma de referências externas feita pela aplicação durante a sua execução, como arquivos, impressoras, dispositivos, outros processos, conexões de rede e outros

máquina. É o caso, por exemplo, de uma aplicação que faça acesso a uma base de dados. Se a aplicação acessar esta base de dados localmente, ao migrar, a conexão com a mesma deverá ser reestabelecida através da rede para que sua execução possa prosseguir;

(d) *Recurso despachado:* neste caso, o recurso deverá ser migrado inteiramente junto com o estado e o código da aplicação. Arquivos utilizados pela aplicação que sejam inteiramente migrados com ela é um exemplo do emprego desta abordagem.

3. **O código e o estado da entidade migrada são transportados para a máquina de destino:** o estado e o código da aplicação são serializados e transmitidos através da rede para a máquina destino;

4. **O código e o estado da entidade migrada são restaurados:** nesta etapa a aplicação precisa restaurar o seu estado de execução e reestabelecer (se necessário for) sua ligação com os recursos necessários para que sua execução prossiga;

5. *A execução é reiniciada:* a execução é retomada do ponto onde parou antes da ocorrência da migração.

3.2 Migração de Código em Java

Migração de código não é um conceito novo. Muitos sistemas foram desenvolvidos provendo esta funcionalidade como Charlotte [AF89], Sprite [DO91] e Emerald [JLHB88]. Entretanto, com o advento da tecnologia Java, o trabalho de pesquisa na área tem se concentrado na criação de um mecanismo eficiente para a migração de *threads* Java. Isto foi estimulado pelo fato de que a maioria das plataformas de agentes móveis terem sido desenvolvidas utilizando a linguagem Java, o que provê a elas um alto grau de portabilidade.

A portabilidade da plataforma Java deve-se ao fato do compilador Java não gerar instruções específicas a uma plataforma, mas sim um código intermediário denominado *bytecode*. Além disso, a facilidade em transportar *bytecode* através da rede, o mecanismo de serialização (que permite a migração de dados de objetos) e os conceitos de segurança providos pela plataforma Java são algumas outras características que tornam

Java uma excelente escolha para o desenvolvimento de aplicações móveis.

Entretanto, o uso de Java introduz vários problemas para a implementação da migração forte de aplicações. Java não provê mecanismos suficientes para capturar o estado de execução de computações. Seu mecanismo de serialização somente permite a salva do código e do valor dos atributos de objetos. As classes Java não podem acessar informações nativas e internas da Máquina Virtual Java (por exemplo, o contador de instrução e a pilha de chamadas), necessárias para a captura do estado completo de execução de *threads* Java.

Durante os últimos anos, algumas técnicas foram desenvolvidas no intuito de permitir a captura do estado de execução de *threads* Java. Estas técnicas podem ser classificadas segundo quatro abordagens básicas [IKKW00]:

- **Alteração da Máquina Virtual:** a máquina virtual é modificada de forma a exportar as informações a respeito da execução das aplicações. A maior desvantagem desta abordagem é a perda de compatibilidade com relação à máquina virtual padrão. O ITS [BHP03] e o Merpati [Sue00] utilizam esta abordagem;

- **Instrumentação do código-fonte das aplicações:** esta técnica consiste no uso de um pré-processador (um compilador de código-fonte) que insere na aplicação código-fonte adicional responsável por capturar e restaurar seu estado de execução. O principal problema desta técnica é que o código-fonte da aplicação deve estar disponível, o que não é sempre possível, como no caso, por exemplo, em que são utilizadas bibliotecas e aplicações legadas. Uma outra desvantagem com relação ao uso desta técnica é a sobrecarga gerada no tempo de execução e no tamanho do código da aplicação. Esta técnica é implementada pelos arcabouços WASP [Fun98] e JavaGo [SMY99];

- **Instrumentação do bytecode das aplicações:** nesta abordagem, o código para a captura e a restauração do estado de execução é inserido diretamente no *bytecode* da aplicação. Assim como na abordagem de instrumentação de código-fonte, alguma sobrecarga no tempo de execução e no tamanho do *bytecode* é gerada pela inserção de código adicional na aplicação original. Entretanto, esta sobrecarga é geralmente menor que a gerada na abordagem baseada em código-fonte. Uma outra vantagem obtida pela adoção desta abordagem é que, em nível de *bytecode*, tem-se acesso a um

conjunto estendido de instruções, como por exemplo a instrução `goto`, utilizada para efetuar a restauração do contador de instrução das aplicações. No Brakes [TRV+00] e no JavaGoX [SSY00] esta alternativa é implementada;

- **Modificação da Java Platform Debugger Architecture:** a *Java Platform Debugger Architecture* (JPDA) é parte da especificação padrão da máquina virtual. Usando a JPDA as informações de execução das aplicações podem ser acessadas em modo de depuração. Isto pode ser explorado para prover migração transparente. Entretanto, é necessário que um outro mecanismo possibilite a recuperação do estado de execução das aplicações, dado que esta capacidade não é fornecida pela especificação padrão da JPDA. Duas soluções são propostas pela literatura para sanar este problema: (1) a modificação do núcleo da JPDA e (2) a modificação do *bytecode* das aplicações. Esta abordagem não permite o uso de compilação JIT[4], impondo uma enorme sobrecarga no tempo de execução das aplicações. O CIA [IKKW01] e o M-JavaMPI [MWL02] adotam esta técnica.

3.3 MAG/Brakes

De todas as propostas apresentadas na seção anterior, a que apresentou-se como a mais adequada para aplicação em ambientes de grades computacionais foi a de instrumentação de *bytecode*. Esta argumentação baseia-se no fato de que: (1) em uma grade composta por inúmeras instituições existirá uma grande variedade de máquinas virtuais Java, tornando inviável o uso de uma única versão de máquina virtual, (2) muitas vezes, bibliotecas e aplicações legadas necessitam ser utilizadas para o processamento de computações na grade, tornando a instrumentação do código-fonte da aplicação impraticável e (3) a modificação da JPDA impõe um alto custo no tempo de execução das aplicações.

Brakes [TRV+00, CTJV00] é um framework que permite a captura e o reestabelecimento do estado de execução de *threads* Java através da abordagem de instrumentação de *bytecode*. Ele foi desenvolvido na Katholieke Universiteit Leuven, Bélgica, pelo grupo de pesquisa em redes de computadores e sistemas distribuídos DistriNet. Atualmente, o

[4]Compilação *Just-In-Time* é um método de compilação utilizado por algumas implementações de Máquina Virtual Java. Com a compilação JIT, o *bytecode* das aplicações é transformado em código nativo durante sua própria execução, o que melhora significamente o desempenho das aplicações Java

Brakes consiste de duas partes básicas:

- Um transformador de *bytecode* (baseado na versão 1.4 da *ByteCode Engineering Library* (BCEL) [Dah01]) que instrumenta o *bytecode* Java de forma a tornar possível a captura do estado interno de execução das aplicações;
- Um pequeno arcabouço que utiliza a habilidade das classes modificadas pelo transformador para permitir que as execuções de *threads* Java sejam interrompidas e retomadas sempre que necessário.

A versão padrão do arcabouço Brakes é chamada "Brakes-serial". Esta versão não permite a execução de *threads* concorrentes e não poderia ser utilizada pelo MAG, dado que, no JADE, cada agente é executado como uma *thread* de um processo *container*.

O Brakes tem outra versão chamada "Brakes-paralelo", que permite a execução de *threads* concorrentes. Porém, esta versão foi desenvolvida como um sistema de "prova-de-conceito", sem nenhuma otimização para uso real. Ele é um componente não otimizado do primeiro protótipo, e causa um enorme custo adicional ao tempo de execução das aplicações. Este alto custo é causado por um conjunto de novos componentes que foram adicionados à versão "serial".

Dadas as limitações de ambas as versões do Brakes, decidiu-se utilizar no MAG somente seu transformador de *bytecode*. O transformador de *bytecode* é o único componente do arcabouço comum às duas versões. Assim, os outros componentes de ambas as versões do Brakes (como por exemplo, um pequeno escalonador de *threads* que as duas versões continham) foram descartados nesta versão modificada, chamada MAG/Brakes [LS05a, LS05b]. Além disso, a estrutura do transformador de *bytecode* foi modificada de forma a obter um melhor desempenho e prover as seguintes novas funcionalidades:

- Ambas as versões do Brakes continham componentes utilizados para armazenar informações de execução e gerenciar a execução das aplicações (por exemplo, iniciar, suspender, parar, etc.). O uso destes componentes causava um grande custo no tempo de execução das aplicações, por causa do grande número de acessos a estes objetos no *heap*. Assim, seu transformador de *bytecode* foi modificado de forma a manter estas informações armazenadas em atributos das próprias *threads*, reduzindo o custo de acesso aos dados de execução;

- O MAG/Brakes permite que a migração de *threads* seja iniciada por uma entidade externa. No mecanismo original do Brakes a migração somente podia ser iniciada pela própria aplicação, forçando os programadores a indicarem em quais pontos de suas execuções a migração deveria ocorrer. Esta nova capacidade é muito importante dado que, em ambientes de grade, o processo de migração é disparado por algum evento que ocorra no ambiente, como por exemplo, um usuário que venha a requerer o uso exclusivo de sua máquina, expulsando todas as computações da grade de seu nó;

- Na implementação original do Brakes a migração poderia ocorrer somente após a invocação de métodos. Hoje o programador de aplicações da grade pode indicar ao transformador de *bytecode*, posições de código adicionais onde ele gostaria que o estado pudesse ser salvo, ou seja, em quais trechos da execução da aplicação a mesma pode ser migrada. Isto é realizado através da invocação do método `mayCheckpoint()`;

- O programador de aplicações da grade pode desejar que o transformador de *bytecode* não insira nenhum código na aplicação de maneira automática (i.e. após as invocações de métodos). Assim, este transformador somente permitirá que a aplicação seja migrada nas posições de código explicitadas pela chamada ao método `mayCheckpoint()`, ignorando as outras invocações a métodos existentes no código original da aplicação. Esta característica está disponível no MAG/Brakes.

3.3.1 Implementação

Para fornecer as capacidades de migração e *checkpointing* às aplicações da grade, o transformador de *bytecode* do MAG/Brakes necessita inserir na aplicação, blocos de código relativos à captura e recuperação do estado de execução. Entretanto, a execução destes códigos está vinculada a diferentes modos de operação da aplicação, regidos por atributos booleanos presentes nas mesmas (*isSwitching*, *isStopping* e *isRestoring*). A combinação destes atributos resulta em diferentes operações que podem ser realizadas através do MAG/Brakes. A Tabela 3.1 resume as três possíveis operações obtidas através da combinação correta destes atributos.

Para a realização do processo de migração é necessário que os atributos *isSwit-*

Operação	**isSwitching**	**isStopping**	**isRestoring**
Migração	V	V	F
Recuperação	F	F	V
Checkpoint	V	F	F

Tabela 3.1: Operações permitidas pelo MAG/Brakes

ching e *isStopping* tenham seus valores atribuídos como verdadeiro. Dessa forma, após a captura do estado ser efetuada, a aplicação interrompe a sua execução, podendo assim ser migrada para outra localidade. No destino, o processo análogo (a recuperação do estado de execução) necessita ser efetuado. Para tanto, o atributo *isRestoring* deve ter o seu valor verdadeiro. Além destas operações, o MAG/Brakes também pode ser utilizado para prover tolerância a falhas, fornecendo o *checkpoint* das aplicações. Para que esta operação seja realizada, o atributo *isSwitching* deve ser verdadeiro, forçando a captura do estado de execução. Porém, a execução da aplicação não deve ser interrompida neste processo. Para tanto, o atributo *isStopping* deverá ter seu valor falso.

Além das informações relativas ao seu modo de operação, cada *thread* executando na grade tem associada a si um objeto `Context`, no qual é armazenado seu estado de execução (contexto). Este contexto e as variáveis booleanas são atributos de uma classe chamada `MagApplication`, que deve ser herdada por todas as aplicações que executam no MAG.

Para reestabelecer o contexto de execução de uma *thread*, o MAG/Brakes precisa capturar seu contador de instruções e sua pilha de chamadas. A pilha de chamadas de uma *thread* Java (também conhecida como *Java virtual machine stack*) é responsável por armazenar o seu conjunto de *frames*. O *frame* é uma estrutura de dados que armazena informações relativas a um dado método, sendo ele o responsável por armazenar o valor dos atributos, das variáveis locais, da pilha de operandos e do valor de retorno do método, além de ser o responsável por despachar exceções não interceptadas pelo mesmo. Um novo *frame* é criado e colocado na pilha de chamadas quando ocorre uma invocação a método, e destruído quando o mesmo termina, de forma normal ou abrupta (i.e. na ocorrência de um erro).

A captura do estado de uma *thread* é realizada através de um processo no qual o fluxo de execução da *thread* é desviado, forçando que sua execução venha a retroceder

por toda a pilha de chamadas, salvando cada *frame* em seu objeto de contexto. Assim será possível posteriormente, reconstruir a pilha de chamadas através do processo de recuperação, onde cada *frame* é retirado do contexto e recuperado na ordem inversa em que foi salvo.

A Figura 3.5 apresenta os processos de captura e recuperação do contexto de uma *thread* `foo`, instanciada e executada a partir de uma *thread* `ti` externa. Neste exemplo, durante a execução do método `run()`[5] da *thread* `foo`, é invocado o método `f()` da mesma *thread*.

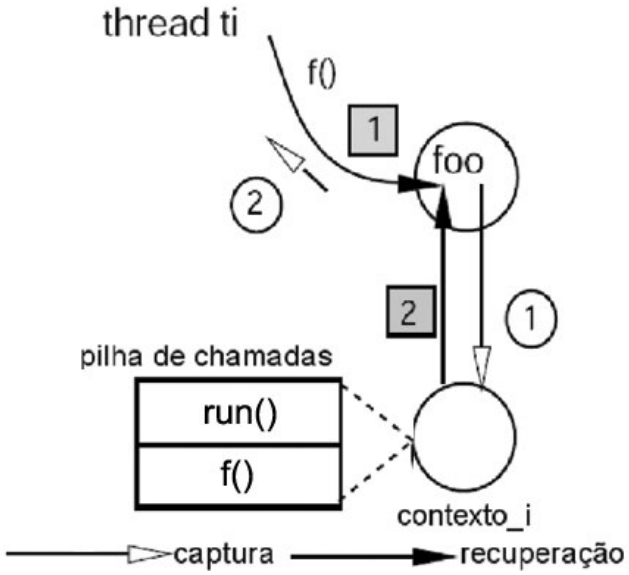


Figura 3.5: Captura / restauração no MAG/Brakes

Os processos de captura e recuperação do estado de execução de *threads* Java realizados pelo MAG/Brakes, são detalhados a seguir, tomando como base a Figura 3.5.

Captura do Estado de Execução

Caso venha a ser solicitada a migração de `foo` durante a execução de `f()` (a *thread* `ti` poderia ser a responsável por esta ação), o processo de captura do estado da *thread* seria iniciado. Este processo é indicado na Figura 3.5 pelas setas com pontas brancas e pela numeração dentro de círculos.

Após receber a notificação de captura do estado, o fluxo de execução do método `f()` de `foo` é desviado de forma a salvar o *frame* atual. Depois de concluir este processo,

[5]Método principal das *threads* Java. Este método é executado quando a mesma é iniciada.

o *frame* `f()` é então salvo no objeto de contexto `contexto_i` (1) e o método então retorna, devolvendo o controle para o método que o invocou (método `run()` da própria `foo`). O processo continua com `run()` também tendo seu fluxo desviado para salvar seu próprio *frame* no `contexto_i`, retornando após o término deste processo (2). Ao final deste processo, o objeto `contexto_i` contém todo o segmento de execução de `foo`, podendo ser migrado por uma entidade externa para uma localidade remota (no exemplo, a *thread* `ti`). Vale ressaltar que os desvios de código causados neste processo são reflexo da modificação dos atributos booleanos (*isSwitching* e *isStopping*) da *thread* envolvida no processo. Estes atributos são manipulados pelo `MagAgent` através de métodos disponibilizados pelo MAG/Brakes para a realização de cada operação (migração, recuperação e *checkpoint*).

Para cada *frame* salvo da aplicação, também deve ser salvo seu contador de instruções, que representa a posição de código em que o método se encontrava quando a aplicação recebeu a notificação de captura de estado. Entretanto, a JVM não permite que o contador de instruções seja obtido por parte das aplicações que nela executam. Para superar este problema os desenvolvedores do Brakes convencionaram que o estado de execução sempre deveria ser salvo após a ocorrência de invocações de métodos (o MAG/Brakes estendeu este suporte às invocações ao método especial `mayCheckpoint`), ou seja, o contador de instruções sempre indica a posição de código de uma destas invocações. Um contador de instruções artificial foi então utilizado para indicar a "última invocação executada" (*last performed invoke-instruction*) ou *LPI*. O LPI é um índice cardinal incrementado a cada invocação de método presente no código da aplicação, ou seja, é um identificador numérico único associado a cada invocação.

Para que o processo de captura seja realizado pela aplicação de maneira transparente o transformador de *bytecode* do MAG/Brakes insere, após as invocações de métodos (incluindo as invocações ao método `mayCheckpoint`), um bloco de código responsável por salvar o *frame* de seu respectivo método. Este bloco de código é inserido após as invocações de métodos com o intuito de salvar o estado após o retorno dos próprios métodos, forçando a salva dos *frames* em cascata. Um exemplo do bloco de código responsável por efetuar a salva do estado de execução de uma *thread* pode ser visto na Figura 3.6 (linhas 5-20).

Caso o atributo *isSwitching* seja verdadeiro ou caso o tempo desde a tomada do último *checkpoint* tenha expirado[6] (esta verificação é realizada pelo método

[6]O valor padrão de validade de um *checkpoint* é de 30 segundos, podendo este valor ser modificado

```
1:  //-------------------------------------------------------------
2:  int m;
3:  ...
4:  this.f ();  // <- invocação de método
5:  if (this.isSwitching())  {
6:     // armazenar o frame atual no contexto
7:     this.context.pushThis(this);
8:     this.context.pushObject(this);
9:     this.context.pushInt(m);
10:
11:    // armazenar o contador de programa artificial (LPI)
12:    this.context.pushInt(0);
13:
14:    // "efetiva" a salva do contexto atual
15:    copyStableCheckpoint();
16:    if (isStopping()) {
17:       return;
18:    }
19: } ...
20: //-------------------------------------------------------------
```

Figura 3.6: Bloco de código de captura de estado

`isSwitching()`), o processo de captura do estado de execução é então iniciado (linha 5). O primeiro procedimento a ser realizado é a captura das variáveis locais existentes dentro do escopo da invocação do método `f()` (linhas 7-9). No exemplo, a variável local m, que é a única existente até o ponto da salva, é colocada no objeto de contexto da *thread*. O mesmo ocorre com o objeto *this*, ou seja, a *thread* `foo` é colocada no contexto. Isto tem como objetivo salvar o valor dos atributos da mesma. A salva dos operandos presentes no código é garantida por um outro mecanismo do MAG/Brakes executado previamente por seu transformador de *bytecode*, que transforma cada operando em uma variável local, permitindo assim sua salva.

O segundo passo da captura do estado é salvar o contador de instrução do método (índice LPI). No exemplo mostrado, o índice LPI salvo no contexto atual tem o valor "0" (linha 12). Por último, o estado de execução é efetivado, ou seja, o seu conteúdo é copiado para um outro atributo da *thread*, o que lhe dá o estado de "contexto estável"(linha 16). Este código é também responsável por marcar o tempo da última captura de contexto, além de forçar a salva do mesmo no armazém estável (recurso de tolerância a falhas). Logo após a conclusão deste processo, a aplicação verifica se deve ser interrompida (linha 17). Caso o atributo *isStopping* seja verdadeiro, o método atual retorna (linha 18), devolvendo o controle ao método que o invocou.

por qualquer usuário através de arquivos de configuração do MAG

Recuperação do Estado de Execução

O processo de recuperação do estado da aplicação ocorre na ordem inversa em relação à captura, e é representado na Figura 3.5 pelas setas com pontas pretas e pela numeração nos quadrados. Após o término do processo de migração (ou recebimento de um *checkpoint*), a *thread* `ti` instancia e executa a aplicação, informando que seu estado deverá ser recuperado a partir de `contexto_i` (1). `foo` então recupera o primeiro *frame* de `contexto_i`: o *frame* `run()`. Após recuperar todas as informações do *frame* `run()` (2), este é removido do contexto e seu contador de instruções é restaurado para a posição de código da invocação ao método `f()`, ou seja, a posição de código em que o método `run()` encontrava-se quando a solicitação de captura de estado de execução foi recebida. Assim, quando o método `f()` é novamente invocado, `foo` passa a recuperar o *frame* do método `f()` a partir do `contexto_i` (incluindo-se aí seu próprio contador de instruções). A aplicação então volta a executar do mesmo ponto onde parou na máquina de origem, de maneira transparente à aplicação.

Para que seja possível o processo de recuperação, o transformador de *bytecode* do MAG/Brakes insere blocos de código que restauram os dados *frame* atual no início de cada método invocado no código da aplicação. Um exemplo deste bloco de código pode ser visto na Figura 3.7 (linhas 5-24).

Se o atributo *isRestoring* for verdadeiro, a aplicação desvia seu fluxo de execução de forma a recuperar o estado armazenado em seu objeto de contexto (linha 5). A *thread* então verifica se ainda existem *frames* na pilha de chamadas (linha 7). Em caso positivo, a aplicação marca o seu atributo *isRestoring* como falso (linha 9), indicando que após a recuperação do *frame* atual, o processo de recuperação não deverá prosseguir. Para evitar que logo após a recuperação de seu estado a aplicação venha a voltar a capturá-lo, a *thread* muda o valor de seus atributos booleanos, indicando que ela mesma não deve salvar seu estado e nem interromper sua execução, até que outra *thread* informe que isto é necessário (linhas 11-12). A decisão de qual *frame* recuperar depende do índice LPI salvo no contexto (linha 15). No exemplo mostrado, só existe um *frame* a ser recuperado: aquele cujo índice LPI é igual "0" (linha 16) (i.e. o *frame* do método `f()`). As variáveis locais do *frame* passam então a ser recuperadas (linhas 18-20). Entre estas variáveis, encontra-se a variável *this*. À primeira vista parece estranho atribuir algo à esta variável (linha 19) porém, em nível de *bytecode* esta operação é permitida, e faz com que os atributos da

```
1:  //-------------------------------------------------------------
2:  ...
3:  public int f () {  // <- início do corpo do método
4:      int m;
5:      if(this.isRestoring())  {
6:          // se este for o último frame a ser recuperado
7:          if (this.lastFrameInContext()) {
8:              // o estado não deve ser mais recuperado
9:              restoring = false;
10:         }
11:         switching = false;
12:         stopping = false;
13:
14:         // recupera o LPI do contexto
15:         switch (this.context.popInt()) {
16:             case 0:
17:             // restaura o frame atual
18:             m = this.context.popInt();
19:             this = (Foo) this.context.popObject();
20:             this.context.popThis();

21:             // recupera o contador de instrução do frame atual
22:             goto _L5;
23:         }
24:     } ...
25: } ...
26: //-------------------------------------------------------------
```

Figura 3.7: Bloco de código de recuperação de estado

thread em execução sejam recuperados. Após recuperar o *frame* atual, o contador de instrução é recuperado (linha 22). No exemplo ele aparece como um marcador de código (*_L5*), cuja instrução de invocação de destino está associada.

Na grade, quando o `MagAgent` deseja migrar uma *thread* para um outro nó, ele invoca o método `doYield()` da mesma. Este método torna os atributos *isSwitching* e *isStopping* verdadeiros, causando a captura do estado e a interrupção da execução da *thread*. Após o término deste processo, o `MagAgent` pode serializar o contexto da *thread* e migrá-lo para outra localidade na grade. Para restaurar a execução dessa mesma *thread* no destino, após deserializar a *thread*, o `MagAgent` deve invocar o método `doResume()` que atribui o valor verdadeiro para o atributo *isRestoring*, causando a recuperação de seu contexto de maneira transparente. Para que o processo de migração ocorra, é necessário que a *thread* da aplicação tenha herdado a classe `MagApplication`, que provê as capacidades necessárias para a migração forçada, além de ser instrumentada pelo transformador do MAG/Brakes. Já para solicitar a captura do *checkpoint* da aplicação, o `MagAgent` invoca o método `captureCheckpoint()`, que muda o valor dos atributos *isStopping* e *isSwitching*, para falso e verdadeiro, respectivamente. O *checkpoint* da aplicação é então retornado ao `MagAgent` que aplica compactação ao mesmo e o salva no armazém estável.

3.3.2 Avaliação de Desempenho

A inserção de código na aplicação introduz um custo adicional no tempo de execução e no tamanho do *bytecode* aplicação. Uma vez que o código de salva é inserido após cada invocação de método, a sobrecarga no tamanho do *bytecode* é diretamente proporcional ao número de invocações que ocorrem no código [TRV+00], sejam invocações de métodos da própria aplicação, sejam invocações ao método `mayCheckpoint()`.

Foi avaliada a sobrecarga imposta pela instrumentação do *bytecode* da aplicação através de dois experimentos. Esta sobrecarga é causada pela inserção de comandos condicionais após as invocações de métodos. No primeiro experimento o objetivo foi avaliar o impacto causado no tempo de execução de uma aplicação de referência da grade. No segundo, o custo adicionado ao tempo de execução das aplicações pelo MAG/Brakes é comparado com o adicionado pelas outras versões do Brakes. Todos os testes foram executados em uma máquina com a seguinte configuração: processador Intel Pentium 4 com 2.8 GHz e 1 GB de memória RAM, executando Linux com kernel 2.6.10.

No primeiro experimento foi medida a sobrecarga de tempo imposta pela instrumentação do *bytecode* em comparação com o tempo de execução normal de uma aplicação real da grade: o algoritmo GLCM (*Gray Level Co-occurrence Matrix*) [HSD73]. GLCM é um dos mais conhecidos métodos para análise da textura de imagens. Nos testes executados foi utilizada como entrada para a aplicação um banco de dados de imagens composto por 24 imagens de satélites na resolução de 3200 x 2400. Além disso, o algoritmo foi executado com diferentes parâmetros como distância (1, 2 e 3) e direção (0°, 45°, 90°, 135°).

Este experimento consistiu em medir o tempo de execução da aplicação GLCM em duas circunstâncias: sem sofrer qualquer modificação em seu *bytecode* e após passar pelo processo de instrumentação do MAG/Brakes. Foram submetidas à grade 30 requisições de execução em cada uma destas circunstâncias. A ferramenta ASCT foi modificada para executar este processo de maneira automatizada, gerando sucessivas requisições de execução da aplicação GLCM com um intervalo de 10 segundos entre cada submissão. Esta versão modificada de ASCT armazenava, em um arquivo de dados, os tempos de submissão e retorno dos resultados da computação, necessários para calcular o tempo médio de execução da aplicação.

A Tabela 3.2 apresenta a sobrecarga causada pela instrumentação do *bytecode*

da aplicação GLCM em relação ao seu tempo de execução normal. É possível perceber que o mecanismo de captura do estado de execução do MAG causa uma sobrecarga muito pequena nesta aplicação (2.37%).

	Tempo Médio de Execução (ms)	**Sobrecarga (%)**
Normal	400,196	–
Instrumentada	409,692	2,37%

Tabela 3.2: Sobrecarga causada pela instrumentação da aplicação GLCM

No segundo experimento, foram medidos dois aspectos distintos do processo de instrumentação:

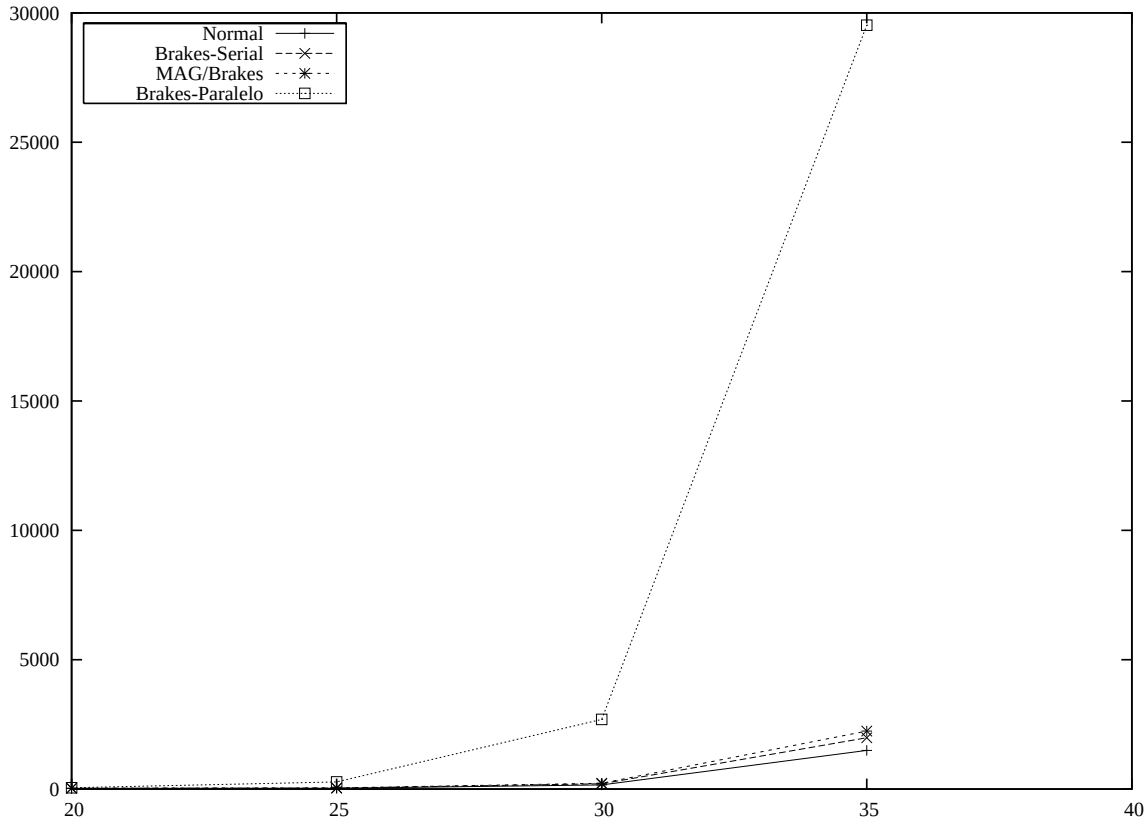
1. O aumento do tempo de execução de uma implementação recursiva do algoritmo de Fibonnaci;

2. O aumento do tamanho do arquivo de classe para a mesma aplicação.

O algoritmo de Fibonacci recursivo foi escolhido por causar um grande número de invocações de métodos, podendo ser considerado um "pior caso" do mecanismo de migração forte do MAG. Os resultados destas medições foram comparadas às outras versões do Brakes: o Brakes-serial e o Brakes-paralelo.

A Figura 3.8 compara os tempos de execução para o algoritmo de Fibonacci dadas as entradas 20, 25, 30 e 35. Pode-se notar que a sobrecarga causada ao tempo de execução da aplicação pelo MAG/Brakes, para todos os parâmetros do algoritmo de Fibonacci, é superior à causada pelo Brakes-serial. Entretanto, a sobrecarga adicional causada pelo MAG/Brakes em relação ao Brakes-serial não ultrapassa 0,15%. Esta diferença deve-se ao fato de que o Brakes-serial, para acessar as informações referentes à execução de uma aplicação, utiliza invocações estáticas, que são mais rápidas que as invocações convencionais feitas a métodos de objetos no *heap*. Entretanto, isto limita o Brakes-serial a manipular somente uma única referência (estática) à *thread* em execução, o que não acontece no MAG/Brakes. Também pode ser percebido que o MAG/Brakes oferece um desempenho muito melhor que o do Brakes-paralelo.

O outro aspecto medido foi o aumento no tamanho do *bytecode* da aplicação causado por cada mecanismo em comparação ao seu tamanho original. A Tabela 3.3 mostra os valores (em KB) dos arquivos de classe em cada abordagem.



Algoritmo de Fibonacci				
Arcabouço	**20**	**25**	**30**	**35**
Normal	25,87	31,80	158,60	1493,80
Brakes-Serial	31,90	41,07	207,40	1984,40
MAG/Brakes	33,03	42,37	226,23	2245,13
Brakes-Paralelo	50,00	274,13	2695,13	29523,87

Figura 3.8: Tempo de execução para um algoritmo de Fibonacci recursivo (em ms)

Mecanismo	**Tamanho do Arquivo (MB)**	**Sobrecarga (%)**
Normal	1.3	–
Brakes-serial	1.7	30,77 %
MAG/Brakes	1.9	46,15 %
Brakes-parallel	1.7	30,77 %

Tabela 3.3: Comparação do aumento do tamanho dos arquivos de classe causado por cada mecanismo

O MAG/Brakes gerou o maior aumento no tamanho do *bytecode* da aplicação entre os arcabouços avaliados: 46,15%. Isto ocorre porque o MAG/Brakes inclui código adicional em relação ao Brakes para a prover as novas características descritas na seção

3.3.

Baseando-se nos testes executados, pode-se concluir que a sobrecarga de tempo e espaço causada pelo MAG/Brakes, apesar de ser ligeiramente maior que a causada pelo Brakes-serial, é justificada pela vantagem de poder executar múltiplas *threads* concorrentemente. Esta característica é extremamente importante no contexto da infraestrutura de *middlewares* de grades.

Neste capítulo foi descrito o mecanismo de migração no contexto do MAG. Foram detalhados os requisitos envolvidos na construção deste mecanismo e os problemas associados à captura / recuperação do estado de execução de *threads* em Java. Foi também apresentado o MAG/Brakes, um arcabouço baseado no Brakes que provê o mecanismo de migração forte do MAG.

Em um futuro próximo, pretende-se superar algumas limitações da versão atual do MAG/Brakes, como o suporte à migração de aplicações *multi-threaded* e à migração dos recursos associados às *threads* migradas.

4 Avaliação de Desempenho

Durante o desenvolvimento do MAG, foram realizados diversos testes que objetivaram avaliar o desempenho de seus componentes em diversas situações, identificando-se possíveis pontos de melhoria de sua arquitetura.

Neste capítulo são apresentados os resultados obtidos a partir de testes de desempenho executados no MAG. Os testes descritos foram realizados no Laboratório de Sistemas Distribuídos (LSD), pertencente ao Departamento de Informática da Universidade Federal do Maranhão. Para a realização dos experimentos foram utilizadas 4 máquinas, cujas especificações estão disponíveis no apêndice A. Estas máquinas estavam interligadas através da tecnologia de rede Fast-Ethernet, que permite o tráfego de dados a 100 Mbps.

Três experimentos são apresentados neste capítulo, cada qual com objetivos e métodos bem definidos: (a) avaliação do consumo de CPU causado pelos componentes do MAG, (b) avaliação da sobrecarga imposta ao tempo de execução de aplicações por parte do mecanismo de tolerância a falhas do MAG, e (c) avaliação do tempo de execução de aplicações paramétricas na grade, à medida que o número de nós disponíveis para realizar computações cresce.

4.1 Avaliação do consumo de CPU causado pelos componentes do MAG

Neste experimento foi avaliado o impacto dos componentes de gerenciamento de recursos local e global (GRM e LRM) e do *AgentHandler*, utilizando como métrica o uso relativo de CPU. Este experimento envolveu um aglomerado composto por três computadores: uma máquina executava o GRM (`cezanne`), outra o LRM, o *AgentHandler* e a aplicação (`picasso`) e a última o ASCT (`gauguin`).

O experimento consistiu na submissão de 800 requisições de execução de aplicações, capturando a cada 10 segundos o valor da métrica utilizada. Durante todo o experimento,

foram coletadas 2500 amostras. A aplicação submetida foi uma implementação do algoritmo de Fibonacci sem recursividade, com uma entrada de 45. Para a realização da medição foi desenvolvida uma pequena aplicação em C e utilizados *scripts* para coletar os dados de consumo de CPU do de cada componente avaliado. Além disso, o ASCT foi modificado de forma a gerar sucessivas requisições de execução da aplicação Fibonacci, automatizando a realização dos testes.

Para simular um ambiente real foram gerados intervalos de tempo pseudo-aleatórios entre duas submissões de execução, sendo estes intervalos de tempo regidos por uma distribuição exponencial [Jai91]. A expressão matemática utilizada para a geração deste tempo foi calculada a partir da função de densidade de probabilidade da distribuição exponencial. Esta distribuição foi escolhida por não necessitar do momento da ocorrência do último evento, sendo esta característica explorada para modelar o momento da ocorrência do evento seguinte. Desta forma foi possível simular a ocorrência de sucessivos eventos, como seria o caso, por exemplo, de sucessivas submissões de execução ao sistema. Sobre os dados obtidos, foram calculadas diversas estatísticas como média, desvio padrão, moda, mediana e intervalo de confiança (o grau de confiança utilizado foi de 0,95 ou 95%).

A Tabela 4.1 apresenta os resultados para o componente *AgentHandler*. Como pode ser visto, a média de uso do processador é próxima à 0%. A mediana confirma isto, sugerindo que pelo menos metade dos valores, quando ordenados, não são maiores que 0%. O desvio padrão apresenta apenas uma pequena variabilidade. Assim, pode-se concluir que o *AgentHandler* causa um impacto muito pequeno com relação ao uso do processador de nós da grade.

Média	Desvio padrão	Mediana	Valor Máximo	Moda	Intervalo de Confiança
0,086	0,523	0,000	7,143	0,000	[0,066 ; 0,107]

Tabela 4.1: Percentual de consumo de processador causado pelo *AgentHandler*

A Figura 4.1 apresenta o gráfico de dispersão relativo ao uso de processador do *AgentHandler*. Como pode ser visto, a grande maioria dos valores coletados de uso do processador são próximos de 0% e nunca superam 7,2%, o que confirma o baixo consumo de processador por parte do *AgentHandler*. Foi executada uma análise similar para o LRM e o GRM, porém foram omitidos seus gráficos de dispersão.

A Tabela 4.2 apresenta os resultados do mesmo experimento quando realizado

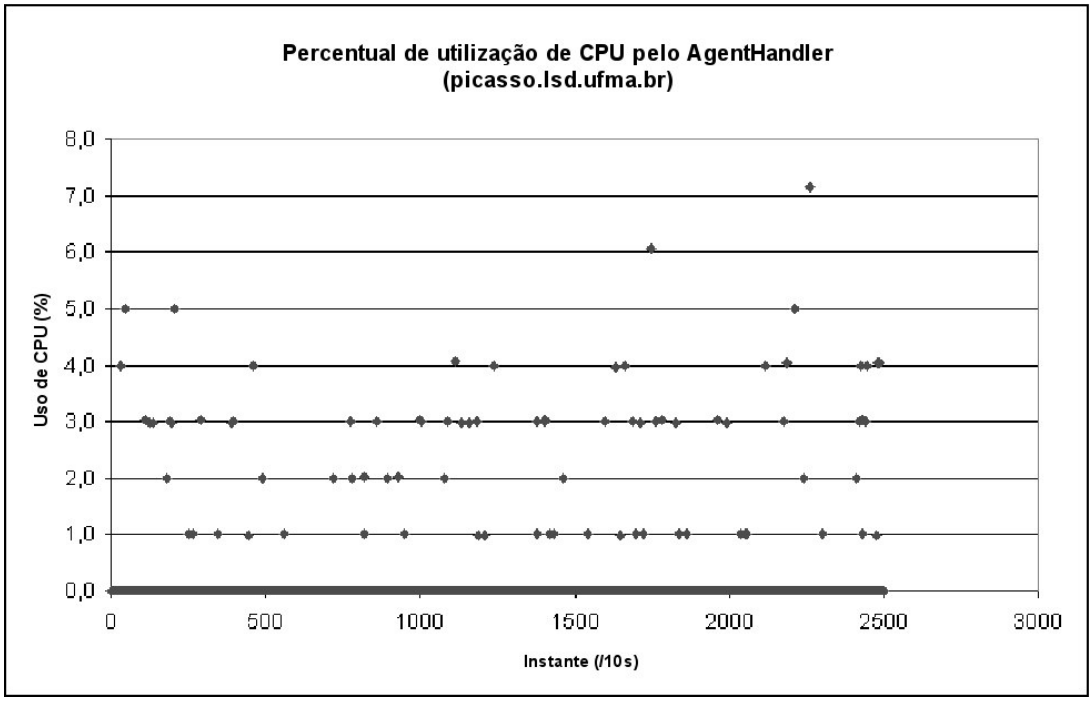


Figura 4.1: Uso de processador causado pelo *AgentHandler*

sobre o LRM, enquanto que a Tabela 4.3 demonstra os resultados relativos ao GRM. Pode-se perceber que estes resultados são similares aos obtidos no experimento com o *AgentHandler*: a média de uso de CPU para o LRM e o GRM é muito próxima a 0%.

Média	Desvio padrão	Mediana	Valor Máximo	Moda	Intervalo de Confiança
0,040	0,247	0,000	2,062	0,000	[0,030 ; 0,049]

Tabela 4.2: Percentual de consumo de processador causado pelo LRM

Média	Desvio padrão	Mediana	Valor Máximo	Moda	Intervalo de Confiança
0,012	0,128	0,000	3,960	0,000	[0,007 ; 0,017]

Tabela 4.3: Percentual de consumo de processador causado pelo GRM

Pode-se concluir pela da análise dos resultados coletados a através dos experimentos, que o consumo de CPU gerado pelos componentes do MAG e do Integrade responsáveis por gerenciamento e monitoração de recursos e execução de aplicações, é muito pequeno e pode, em muitos casos, ser negligenciado.

4.2 Avaliação da sobrecarga causada pelo mecanismo de tolerância a falhas

O mecanismo de tolerância a falhas do MAG baseia-se na abordagem de *checkpointing*, como visto no capítulo 2. Um transformador de *bytecode* insere na aplicação, o código responsável por capturar o seu estado de execução, salvando-o em um repositório estável. Entretanto, o uso desta abordagem introduz um certo custo adicional no tempo de execução e no tamanho do *bytecode* da aplicação.

O experimento realizado para avaliar a sobrecarga causada por este mecanismo foi baseado no apresentado na seção 3.3.2. No experimento apresentado anteriormente foi medido o tempo médio de execução da aplicação GLCM em duas circunstâncias: sem sofrer qualquer modificação e após ser instrumentada pelo transformador do MAG/Brakes. O intuito do experimento da seção 3.3.2 foi medir a sobrecarga causada pelo processo de instrumentação do *bytecode* da aplicação.

Neste experimento, a aplicação GLCM, além de passar pelo processo de instrumentação do MAG/Brakes, teve seu *checkpoint* periodicamente capturado e salvo em um repositório estável durante sua execução. A exemplo do experimento mostrado na seção 3.3.2, a aplicação foi submetida para execução na grade 30 vezes, e a medição do tempo de execução da aplicação em cada submissão foi obtida através da captura dos tempos de submissão e retorno dos resultados da computação através do ASCT. Foi utilizado um intervalo de tempo de aproximadamente 30 segundos entre a salva de dois *checkpoints*, podendo este intervalo variar de acordo com a maneira com que a aplicação foi escrita. A Tabela 4.4 apresenta as estatísticas obtidas a partir dos tempos de execução da aplicação GLCM na presença do mecanismo de tolerância a falhas do MAG. Como pode ser visto na Tabela, a baixa variabilidade[1] apontada pelo valor do desvio padrão comprova a validade do experimento, sugerindo que os tempos de execução obtidos não variaram significativamente. O grau de confiança utilizado para calcular o intervalo de confiança foi de 95%.

A Tabela 4.5 apresenta o percentual de sobrecarga causada por este mecanismo em relação ao tempo de execução normal sem *checkpoint* ou instrumentação da aplicação de referência. Como foi mostrado na seção 3.3.2, o custo acrescido ao tempo de

[1] O desvio padrão obtido representa menos de 1% do valor da média

Média	Desvio padrão	Mediana	Valor Máximo	Intervalo de Confiança
435,938	3,599	434,802	443,713	[434,650 ; 437,226]

Tabela 4.4: Estatísticas do tempo de execução da aplicação GLCM com as extensões de tolerância a falhas (em ms)

execução da aplicação GLCM, após sua instrumentação, é de 2,37%, como pode ser visto na linha *Instrumentação* da Tabela 4.5. Já a sobrecarga total causada pelo mecanismo de tolerância a falhas do MAG pode ser visto na linha *Checkpoint + Instrumentação* da mesma Tabela, que representa a sobrecarga causada pela instrumentação da aplicação e pela salva periódica de seu *checkpoint*.

Modo de Execução	**Tempo de Execução (ms)**	**Sobrecarga (%)**
Normal	400,196	–
Instrumentação	409,692	2,37%
Checkpoint	426,442	6,56%
Checkpoint + Instrumentação	435,938	8,93%

Tabela 4.5: Sobrecarga imposta pelas extensões de tolerância a falhas do MAG

A partir deste experimento foi possível perceber que o custo acrescido pelo mecanismo de tolerância a falhas do MAG no tempo de execução de aplicações pode ser considerado razoável, dados os benefícios da tolerância a falhas. Por exemplo, no experimento apresentado nesta seção, a sobrecarga total causada pelo mecanismo no tempo de execução da aplicação GLCM foi de 8,93%, sendo este um custo aceitável para que se aumente a confiabilidade da execução de aplicações na grade.

4.3 Avaliação do tempo de execução de aplicações paramétricas

Aplicações paramétricas são aquelas que executam múltiplas cópias do mesmo binário, em máquinas e com entradas diferentes. Esta classe de aplicações (também chamada de BoT ou *Bag-of-Tasks*) divide as tarefas em sub-tarefas menores que executam de forma independente, sem haver, entretanto, comunicação entre elas.

A aplicação GLCM foi adotada neste experimento por se enquadrar com precisão a esta classe de aplicações, além de ser uma computação de longa duração, ideal para ser executada em ambientes de grade. Por exemplo, se for utilizada uma coleção de imagens como entrada para esta aplicação, cada uma delas poderá ser processada em uma máquina diferente, de maneira paralela e independente das demais.

O objetivo deste experimento é avaliar a eficácia da infraestrutura de grade na execução de aplicações paramétricas. Para tanto, foi calculado o tempo médio de execução da aplicação GLCM à medida em que o número de máquinas disponíveis na grade crescia. Os parâmetros passados à aplicação GLCM são os mesmos utilizados nos experimentos apresentados nas seções 3.3.2 e 4.2.

Para a execução deste experimento foram utilizadas as quatro máquinas do Laboratório de Sistemas Distribuídos. Durante o primeiro ciclo do experimento, o aglomerado contava com apenas uma máquina para efetuar o processamento de todas as imagens da coleção (24 imagens). Após medido o tempo de execução médio da aplicação, uma nova máquina foi adicionada ao aglomerado e as imagens passaram, então, a ser distribuídas entre as duas. Assim, a partir do segundo ciclo do experimento, cada máquina passou a ser responsável por processar metade das imagens (12 imagens), e assim sucessivamente até que todas as máquinas fossem alocadas para executar a aplicação GLCM. Vale mencionar que, durante todo o experimento, a máquina `cezanne` atuou como nó gerenciador do aglomerado, apesar de, durante o último ciclo do experimento, também ceder seus recursos para o processamento da aplicação GLCM.

Para cada ciclo do experimento foi medido o tempo de execução médio da aplicação GLCM. Uma amostra de 30 tempos de execução foi capturada em cada ciclo. Os resultados de todos os ciclos do experimento são apresentados na Tabela 4.6.

Qt. de Máquinas	Média	Des. padrão	V. Mínimo	V. Máximo	Int. de Confiança
1	400,196	2,655	393,595	404,420	[399,246 ; 401,146]
2	209,441	2,849	206,068	219,072	[208,422 ; 210,461]
3	152,535	4,141	146,024	161,270	[151,053 ; 154,016]
4	113,091	4,904	107,060	127,511	[111,336 ; 114,846]

Tabela 4.6: Estatísticas do tempo de execução da aplicação GLCM (em ms) à medida em que a quantidade de máquinas realizando seu processamento aumenta

O baixo desvio padrão indica que os valores dos tempos de execução capturados através do experimento apresentam baixa variabilidade; os valores mínimo e máximo das amostras não afastam-se significativamente do valor da média, sugerindo que não há, na amostra, a presença de *outliers* – isto é ratificado pelos curtos intervalos de confiança apresentados que indicam que, com 95% de confiança, a média da população estará contida nos intervalos indicados na Tabela.

Como pode ser observado, à medida que o número de máquinas que compõem o aglomerado cresce, o tempo médio de execução da aplicação GLCM cai proporcionalmente. Isto se dá pelo fato de que as imagens que devem ser processadas pela aplicação passam a ser distribuídas entre as máquinas do aglomerado, criando um alto grau de paralelismo de dados[2]. O gráfico da Figura 4.2 apresenta os tempos médios de execução da aplicação à medida em que o número de máquinas do aglomerado aumenta.

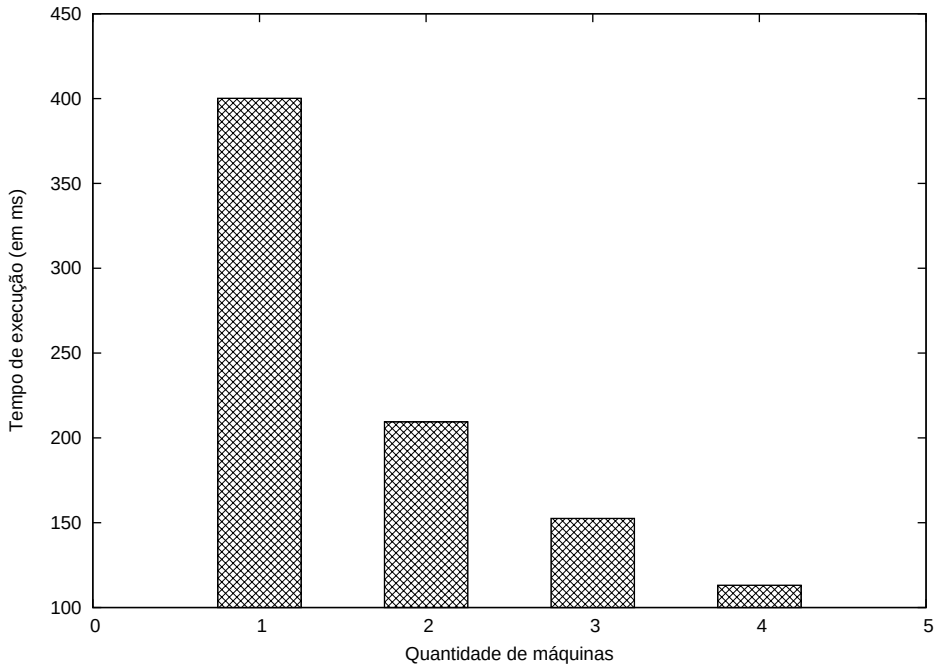


Figura 4.2: Tempo médio de execução da aplicação GLCM à medida que o número de máquinas do aglomerado cresce

A partir da análise do gráfico apresentado é possível identificar o alto grau de eficácia do *middleware* de grade na resolução de aplicações da classe paramétrica ou BoT, e como o nível de disponibilidade de recursos influencia no tempo de resposta da

[2]A expressão "paralelismo de dados" é aqui empregada para designar a distribuição de dados distintos entre cópias idênticas da mesma aplicação.

grade na execução de computações. Apesar da eventual sobrecarga que possa ser acrescida ao tempo médio de execução das aplicações por parte dos componentes e protocolos da grade, ainda é notório o ganho de desempenho obtido por seu uso – principalmente se estas aplicações processam grandes volumes de dados que podem ser divididos entre os nós da grade.

5 Trabalhos Relacionados

A partir do final da década de 90, a computação em grade tem obtido crescente espaço entre pesquisadores da área de computação distribuída. Existem atualmente diversos projetos relacionados à computação em grade, com diferentes objetivos, aspectos de implementação e modelos de aplicações suportadas. Mais recentemente, pesquisadores em *middlewares* de grade têm investigado o uso de agentes móveis para a implementação de diversos mecanismos utilizados nestes ambientes, como o gerenciamento de recursos [CJKN02, Mar04, CST+02, MOMB04, OWSB02], descoberta de recursos e serviços [AMMV04], balanceamento de carga [AAA+04, Cao04, CSJN05] e escalonamento de tarefas [CBL04, Gra03].

Este capítulo descreve relevantes projetos de computação em grade. Após um resumo de cada trabalho é realizada uma análise comparativa com a arquitetura proposta nesta dissertação.

5.1 Globus

O Globus [GLO, FK99, FKT01] é o projeto de computação em grade com maior relevância na atualidade. É desenvolvido pela *Aliança Globus*, um grupo formado por inúmeras instituições ao redor de todo o mundo, cujos principais responsáveis são o Laboratório Nacional Argonne, o Instituto de Ciências da Informação da Universidade do Sul da Califórnia, a Universidade de Chicago, a Universidade de Edimburgo, o Centro Sueco de Computação Paralela, o Centro Nacional de Aplicações de Supercomputação (NCSA) e o Laboratório de Alto Desempenho da Universidade do Norte de Illinois. Além disso, a aliança Globus conta também com o apoio de várias empresas como a Microsoft e a IBM.

O projeto de computação em grade desenvolvido pela aliança chama-se *Globus Tookit*. O *Globus Tookit* fornece um conjunto de serviços e bibliotecas de software de arquitetura e código abertos para o suporte de grades e aplicações de grades. O *tookit* (caixa de ferramentas) tem componentes responsáveis pelo tratamento de vários aspectos relati-

vos aos ambientes de grades, como segurança, descoberta de informação, gerenciamento de recursos, gerenciamento de dados, comunicação, detecção de falhas e portabilidade.

O Globus foi desenvolvido a partir do I-Soft[FGN+97], uma plataforma de software utilizada para gerenciar e desenvolver aplicações que executaram em máquinas de 17 sítios ligados à I-WAY, uma rede ATM de alta velocidade distribuída através do território dos Estados Unidos.

A primeira versão oficial do *toolkit* foi lançado em 1999. Entretanto, só a partir de meados 2001, as empresas despertaram seu interesse e vislumbraram todas as possibilidades que poderiam ser alcançadas através da grade. Após o lançamento da versão 2 do *Globus Toolkit* (batizado com a sigla GT2), esta arquitetura tornou-se o padrão *de facto* para a computação em grade mundial.

Após o sucesso do GT2, o Global Grid Forum (comunidade de usuários, desenvolvedores e vendedores que objetiva a padronização dos conceitos de computação em grade) decidiu definir uma arquitetura padrão para a construção de sistemas de grade. Foi então definido o padrão OGSA [FKNT02] (*Open Grid Services Architecture*), que passou a ser adotado pelo Globus em sua versão GT3.

O padrão OGSA, ao definir o conceito de *serviço de grade*, fundiu as tecnologias de grade e serviços web [BHM+04]. Os serviços de grade são uma extensão dos serviços web que seguem as especificações do padrão OGSA para suportar operações até então não disponibilizadas pelos serviços web, como por exemplo, operações de gerenciamento do ciclo de vida de serviços. A exemplo dos serviços web, os serviços de grade são expressos através da WSDL (*Web Service Description Language*) [CGM+04], uma linguagem XML utilizada para descrever serviços a serem acessados através da web. Uma analogia possível aos descritores WSDL são as IDLs CORBA [Obj02b].

Após a definição da arquitetura do padrão OGSA foi necessário especificar uma a infraestrutura de serviços básicos de forma a permitir a implementação do modelo da arquitetura OGSA. Esta especificação foi chamada de OGSI (*Open Grid Services Infrastructure*) [Glo03]. O GT3 foi o primeiro projeto de grade a adotar a especificação 1.0 da OGSI. O OGSI define as interfaces básicas e os comportamentos de um *serviço de grade*. Entre as várias especificações propostas pelo padrão OGSI encontram-se:

- Um conjunto de extensões para a linguagem WSDL, o que deu origem à linguagem

GWSDL;

- Padrões de estrutura e operação, em WSDL, para representação, pesquisa e atualização de dados sobre os serviços;
- As estruturas *GridServiceHandle* e *GridServiceReference*, utilizados para referenciar um serviço;
- Formato para mensagens indicativas de falhas que não modificam o modelo original de mensagens de falha da linguagem WSDL;
- Conjunto de operações que permitem gerenciar o ciclo de vida dos serviços, como a criação e destruição de serviços de grade;
- Conjunto de operações para criação e uso de coleções de serviços web;
- Mecanismos que permitam notificações assíncronas, caso haja mudança em dados dos serviços.

A Figura 5.1 mostra o relacionamento entre os diversos padrões especificados e utilizados para a definição dos serviços de grade. Como pode ser visto na Figura, o padrão OGSA define conceitualmente os serviços de grade, enquanto o OGSI especifica os seus comportamentos. Os serviços web são estendidos para se tornar serviços de grade, enquanto que o GT3 implementa a especificação OGSI. A arquitetura do GT3 pode ser vista na Figura 5.2.

Os serviços da especificação OGSI são implementados na camada *GT3 Core*. O objetivo desta camada é especificar mecanismos para criação, gerenciamento e troca de dados entre serviços de grade. A camada *GT3 Security Services* contêm os serviços de segurança implementados a partir das funcionalidades fornecidas pelo *GT3 Core*. É nesta camada que se encontram os serviços da GSI (*Globus Security Infrastructure)*) [FKTT98], cujo objetivo é prover transparência para os serviços das camadas de mais alto nível com relação à infraestrutura de segurança da grade.

A camada *GT3 Base Services* inclui uma gama de serviços básicos ao uso da grade, como:

- *Managed Job Service:* este serviço é normalmente conhecido como *job management* (neste caso o temo *job* representa uma operação sendo executada por um *serviço*

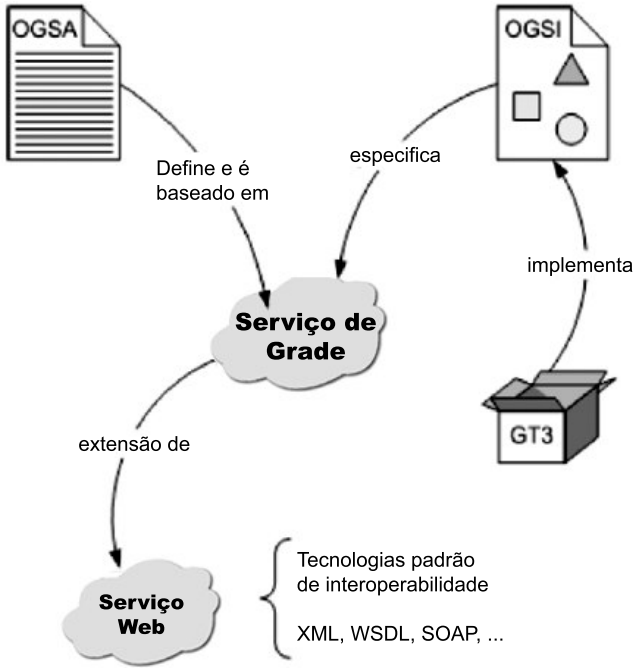


Figura 5.1: Relacionamento entre padrões na definição de serviços de grade

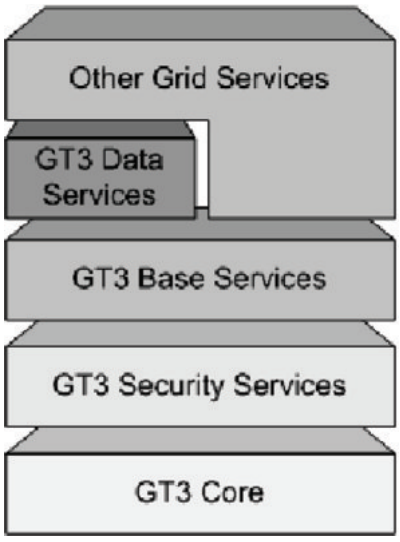


Figura 5.2: Arquitetura do Globus

de grade). Através deste serviço é possível ao usuário acompanhar o progresso na execução de uma dada computação, além de poder exercer controle sobre ela (suspender, parar, continuar, etc);

- *Index Service:* serviço de consulta a serviços de grade, através do qual é possível localizar um determinado serviço de grade segundo quaisquer requisitos particulares. É análogo aos registros UDDI (*Universal Description, Discovery, and Integration*) dos serviços web;

- *Reliable File Transfer Service (RFT):* este serviço permite a transferência confiável de arquivos (inclusive com grandes volumes de dados) entre o cliente e um serviço de grade. Se algum erro ocorrer no meio da transmissão o RFT permite que a transmissão continue do mesmo ponto onde parou antes da ocorrência da falha.

A camada *GT3 Data Services* é responsável por agrupar serviços de dados, como por exemplo, o serviço de gerenciamento de réplicas. Por fim, a camada *Other Grid Services* representa outros serviços não-GT3 que podem ser eventualmente implementados e agregados ao topo da arquitetura básica do Globus de forma a prover implementações específicas de usuários da grade.

No intuito de acompanhar a evolução dos serviços web e tornar os serviços de grade compatíveis com as suas novas especificações, a aliança Globus evoluiu a especificação OGSI. Entre os aspectos desta evolução encontram-se:

- Os mecanismos de endereçamento de serviços de grade (*GridServiceHandler* e *GridServiceReference*) foram substituídos pelo *WS-Addressing*, um novo mecanismo para o endereçamento de serviços web que independe do protocolo da camada de transporte [CFF+04];
- A OGSI foi separada em várias especificações menores, porém agrupadas em famílias. Esta medida objetiva a redução da complexidade da OGSI original;
- A GWSDL modificou o padrão WSDL 1.1 original para acrescentar funcionalidades necessárias aos serviços de grade. Isto fez com que a OGSI se tornasse incompatível com a WSDL padrão. Entretanto, a nova versão da OGSI volta a utilizar o padrão WSDL 1.1, o que facilita a descrição de serviços de grade, principalmente por desenvolvedores não habituados à GWSDL: o único requisito básico do desenvolvedor é conhecer a WSDL, conceito proveniente dos serviços web. Além disto, esta medida garante a compatibilidade da especificação com as ferramentas existentes para XML e serviços web.

A partir do refinamento da nova versão da especificação OGSI surgiu a WSRF (*Web Service Resource Framework*) [CFF+04], uma especificação completamente baseada em serviços web. Ela é utilizada pelo GT4 como uma nova especificação utilizada para a construção de *middlewares* de grade.

Originalmente, serviços web não mantém estado e nem têm instâncias. Apesar disto, ao estender os serviços web, os serviços de grade prevêem em sua especificação o uso de serviços web que mantenham estados. Eestes serviços devem permitir que seja mantido um certo tipo de "memória" dos serviços, e que eventuais mudanças no estado dos mesmos sejam válidas por todo o seu ciclo de vida. O WSFR modifica este modelo de forma a criar uma separação clara e explícita entre os serviços web e as entidades que mantém estado (instâncias), que são manipuladas através destes serviços web. Esta composição é denominada pelo padrão WSRF de *WS-Resource*.

O MAG e o Globus são projetos bem distintos na proposta de suas arquiteturas. O Globus está completamente integrado aos conceitos de serviços web. Seus componentes se comunicam através de protocolos SOAP/HTTP. O MAG propõe uma abordagem baseada em agentes para sobrepor os desafios inerentes ao desenvolvimento de um *middleware* de grade. Toda a comunicação dos agentes é feita através dos padrões FIPA-ACL/Java RMI. Apesar de não utilizar serviços web para a composição da infraestrutura básica de serviços da grade, o MAG utiliza esta tecnologia para permitir a usuários nômades a submissão, acompanhamento e coleta dos resultados de computações. Este serviço pode ser acessado através de uma versão do ASCT que executa em um navegador, chamada WASCT.

Outra importante diferença entre o Globus e o MAG está relacionada aos objetivos de cada projeto. Enquanto o MAG adota uma abordagem oportunista, ou seja, utiliza o poder computacional ocioso das máquinas que compõem a grade, o Globus não tem esta preocupação. A infraestrutura de uma grade Globus é composta por máquinas dedicadas, ou seja, máquinas que fornecem à grade todo o seu poder computacional.

5.2 Condor

O projeto Condor [CON, LLM88, TTL04] é um projeto de computação em grade desenvolvido na Universidade de Wisconsin-Madison, cujo principal pesquisador é o professor Miron Livny. É um projeto bastante maduro, dado que está em produção desde a década de 80, para uso acadêmico e empresarial. Este projeto está focado em desenvolver uma arquitetura que integre recursos computacionais para a execução de computações intensivas. Este sistema permite dois diferentes modos de operação: grade

dedicada e grade oportunista. O objetivo do ambiente de grade dedicado é prover aos seus usuários grandes quantidades de poder computacional por longos períodos de tempo, utilizando todos os recursos disponíveis na mesma. Já o objetivo da grade oportunista é utilizar somente os recursos que estiverem disponíveis em um dado momento, sem requerer 100% dos recursos das máquinas.

Uma grade Condor é formada por aglomerados de máquinas chamados *Condor Pool*. Cada aglomerado, em geral, pertence a um domínio administrativo distinto (apesar de isto não ser obrigatório). Os aglomerados que formam a grade são independentes entre si. A arquitetura de um aglomerado Condor (*Condor Pool*) é mostrada na Figura 5.3.

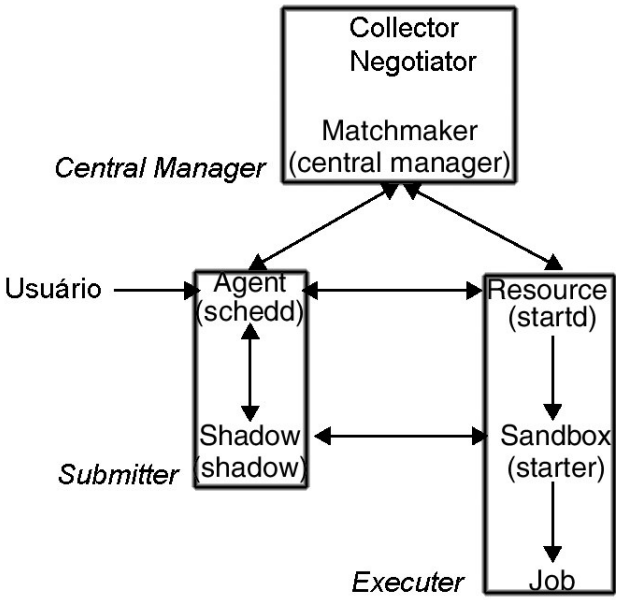


Figura 5.3: Arquitetura de um *Condor Pool*

Os nós componentes de um *Condor Pool* podem assumir três papéis: *Central Manager*, responsável pelas tarefas de gerenciamento do aglomerado; *Submitter* que representa um nó cliente da grade, solicitando a execução de computações; e *Executer* que são os nós que cedem poder computacional ao aglomerado. Um nó pode assumir mais de um papel, como no caso de máquinas que fornecem poder computacional à grade e também solicitam a execução de computações.

Cada nó do aglomerado, ao assumir um dos possíveis papéis do *Condor Pool*, passa a executar alguns módulos responsáveis por exercer atividades relativas à grade. Estes módulos são detalhados a seguir:

- *schedd:* módulo a partir do qual são feitas as solicitações de execução de aplicações

ao Condor. Através dele é possível monitorar remotamente o progresso da execução de aplicações submetidas à grade. Executa em nós *Submitter*;

- *shadow:* representante local de uma aplicação submetida para execução na grade. É instanciado quando uma aplicação é submetida à grade. É responsável por monitorar a aplicação e enviar a ela parâmetros e arquivos (de entrada e saída). Executa em nós *Submitter*;

- *startd:* módulo que permite que computações da grade sejam executas em uma máquina do aglomerado. Além disso, submete periodicamente ao *Central Manager* informações sobre a disponibilidade de recursos na máquina em que executa. Ele é também responsável por aplicar as políticas de compartilhamento de recursos definidas pelo usuário da máquina. Executa em nós *Executer*;

- *starter:* monitora localmente as aplicações que executam na grade. É instanciado quando uma requisição de execução chega em um nó *Executer*, com o objetivo de monitorá-la localmente. Comunica-se diretamente com o módulo *shadow*, informando-o sobre o progresso da execução da computação.

- *collector:* módulo que recebe e mantêm as informações do aglomerado. Também recebe as solicitações de execução provenientes dos *schedds*. Periodicamente, recebe de cada *startd* as informações relativas à disponibilidade de recursos no nó em que executa. Executa no *Central Manager*;

- *negotiator:* é o escalonador do aglomerado. O *negotiator* periodicamente verifica a existência de requisições de execução junto ao *collector*. Em caso positivo, ele procura nós que possam executar a requisição. Executa no *Central Manager*;

A Figura 5.4 ilustra o processo de execução de aplicações no Condor. Este processo ocorre como a seguir: um usuário da grade solicita a execução de uma aplicação à grade (através do *schedd*). Esta requisição está associada a um anúncio de pedido de recursos e é encaminhada ao *collector* (1). As máquinas que fornecem recursos à grade, também enviam periodicamente ao *collector* um anúncio: só que dos seus recursos cedidos à grade (2). De tempos em tempos o *negotiator* avalia os anúncios de requisições e ofertas, associando os que são compatíveis entre si. Este processo é conhecido como *matchmaking* [RLS98]. O *negotiator* notifica o *schedd* que efetuou a requisição, informando quais máquinas executarão sua requisição. O *schedd* então comunica-se com o *startd* da

máquina que ofereceu o recurso para descobrir se aquela oferta continua válida (3). Caso os recursos ofertados continuem válidos, a negociação entre o *schedd* e o *startd* foi bem sucedida. Em seguida o *schedd* instancia um processo sombra (*shadow*) (4), que representará localmente a aplicação que será executada remotamente. Ele será responsável por monitorar e por gerenciar detalhes da aplicação a ser executada em nível local. O *shadow* comunica-se com o *startd* da máquina remota informando sobre a execução da aplicação. O *startd* instancia um processo *starter* (5), que é responsável por monitorar a execução da aplicação na máquina remota. O *starter* então instancia a aplicação (6), solicitando o início de sua execução. Eventualmente, os processos *shadow* e *starter* podem precisar comunicar-se, trocando informações sobre os progressos na execução da computação, requisitando os arquivos de saída, ou simplesmente monitorando a aplicação (7).

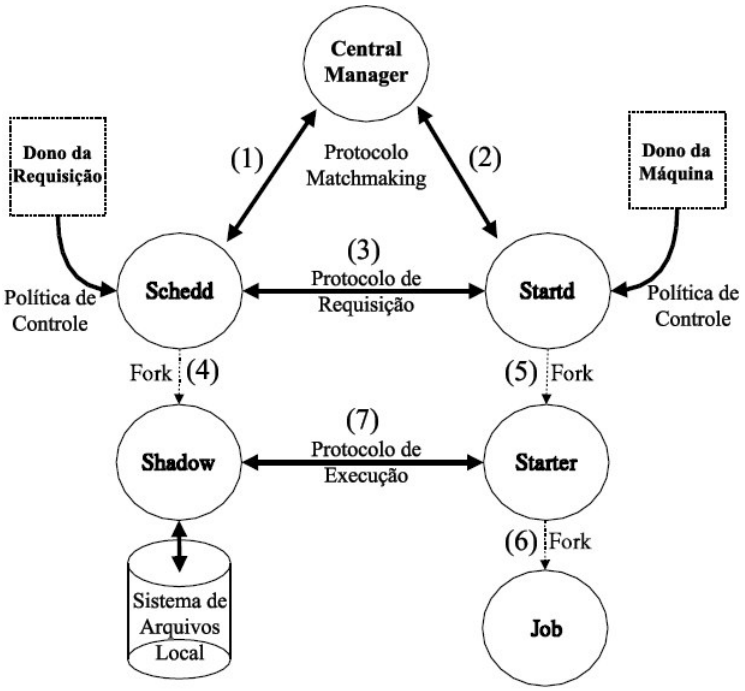


Figura 5.4: Execução de aplicações no Condor

A arquitetura do Condor foi projetada para operar com um pequeno número de máquinas pertencentes a um mesmo domínio administrativo. Com o aumento do número de aglomerados Condor surgiu a necessidade de interligação destes aglomerados em um mesmo sistema, de forma a prover um amplo e eficiente comparilhamento de recursos computacionais distribuídos entre múltiplas instituições. Como uma forma de contornar a limitação inicial da arquitetura do Condor foi proposta uma solução chamada *Gateway Flocking* [ELD+96] ou *Flock of Condors*, na qual um *gateway* é introduzido na estrutura dos *Condor Pools*, sendo este o componente responsável por interligar um aglomerado a

outros. Cada *gateway* é configurado com informações de *gateways* de outros aglomerados. Esta associação é válida somente em um sentido, devendo ser configurada nos dois *gateways* envolvidos, de forma que um associe-se ao outro. O *Flock of Condors* permanece transparente aos aglomerados e permite que um aglomerado forneça e utilize recursos de outros aglomerados. Entretanto, os *Flock of Condors* acumularam uma grade quantidade de tarefas, causando uma queda no desempenho dos *gateways*. Assim, o *Gateway Flocking* foi posteriormente substituído pelo *Direct Flocking*, onde componentes de um aglomerado comunicam-se diretamente com os componentes de outro, sem a presença de um *gateway*. Entretanto esta abordagem apresenta problemas de escalabilidade, dado que cada componente do aglomerado deve conhecer vários outros componentes localizados em diversos aglomerados.

O Condor fornece a seus usuários dois diferentes modos de operação: alto rendimento e oportunista. O modo de alto rendimento faz com que a grade Condor utilize todos os recursos disponíveis, comportando-se de maneira similar a uma grade Globus. Por outro lado, o modo de operação oportunista permite que somente recursos ociosos sejam utilizados para a execução de computações da grade, a exemplo do que ocorre com o MAG.

Ambos, MAG e Condor, utilizam o conceito de aglomerado para definir um agrupamento de recursos computacionais pertencentes a um mesmo domínio administrativo. Na arquitetura dos aglomerados de ambos os projetos, existe a presença de um nó gerenciador central, o que restringe a escalabilidade dos recursos de um aglomerado. Para permitir que uma grande quantidade de máquinas fossem agregadas ao mesmo sistema, cada um destes projetos criou soluções para superar esta limitação: no Condor foi criado o *Gateway Flocking* (ou *Flock of Condors*) que permitiu que vários aglomerados Condor fossem interligados em um mesmo sistema. Entretanto, esta solução não é suficientemente escalável, pois cada *gateway* deve ser configurado com informações de todos os outros *gateways* com os quais deseja comunicar-se. Caso a quantidade de aglomerados a ser conectados seja muito grade, esta solução torna-se inviável. Por outro lado, o MAG adota uma abordagem hierárquica para superar a limitação relativa à quantidade de máquinas compondo a grade. Dessa forma muitos aglomerados podem ser agregados à grade de uma maneira bastante escalável. Uma abordagem baseada em redes P2P (*Peer-to-Peer*) está também sendo explorada no contexto do projeto Integrade [RR05].

5.3 OurGrid

OurGrid [MYG, ACBR03, CPC+03] é um projeto de grade computacional desenvolvido em conjunto pela Universidade Federal de Campina Grande e Hewlett-Packard (HP). O objetivo do OurGrid é pesquisar e desenvolver soluções para uso e gerenciamento de grades computacionais. O OurGrid é um sistema de grade cooperativo, aberto e gratuito, no qual laboratórios podem doar poder computacional em troca de acessar, quando necessário, recursos ociosos pertencentes a outros laboratórios. O projeto OurGrid é fundamentado no projeto MyGrid, desenvolvido também na Universidade Federal de Campina Grande e está em produção desde dezembro de 2004.

Atualmente o OurGrid permite apenas a execução de aplicações paralelas leves. Aplicações leves, também chamadas aplicações BoT ou *Bag-of-Tasks* são definidas como aplicações compostas de um conjunto de tarefas independentes que não necessitam de comunicação durante sua execução. Apesar de parecer um modelo de aplicações paralelas simples, as aplicações BoT podem ser utilizadas em uma grande variedade de cenários, incluindo aplicações de mineração de dados, buscas maciças, varredura de parâmetros, biologia computacional e processamento de imagens.

O OurGrid foi projetado para poder utilizar tanto os recursos ociosos de estações de trabalho como recursos de aglomerados dedicados. Estes últimos podem, inclusive, ser controlados por outros softwares de gerenciamento de recursos como o Maui [MAU], OpenPBS [OPE] e LSF [LSF]).

Os três principais componentes do OurGrid e suas interações são mostradas na Figura 5.5: o MyGrid, a comunidade OurGrid (*OurGrid Community*) e o SWAN.

Os usuários que interagem com o OurGrid utilizam o MyGrid, um componente de software responsável por prover aos usuários, abstrações de alto nível que permitem que eles utilizem a grade. As abstrações chave providas pelo MyGrid são os *jobs*, as *tasks* e as *grid machines*. Um *job* é uma coleção de *tasks*: as *tasks* são as unidades básicas de trabalho da grade que são executadas paralelamente. Cada nó conectado à grade é chamado de *grid machine*. O usuário é responsável por prover uma descrição de seus *jobs* e um ponto de entrada para a comunidade OurGrid na forma de uma URL para um *peer*[1] OurGrid. O MyGrid utiliza esta URL para requisitar *grid machines* disponíveis no

[1]Um *peer* OurGrid corresponde a um aglomerado de máquinas da grade, compreendendo geralmente um único domínio administrativo

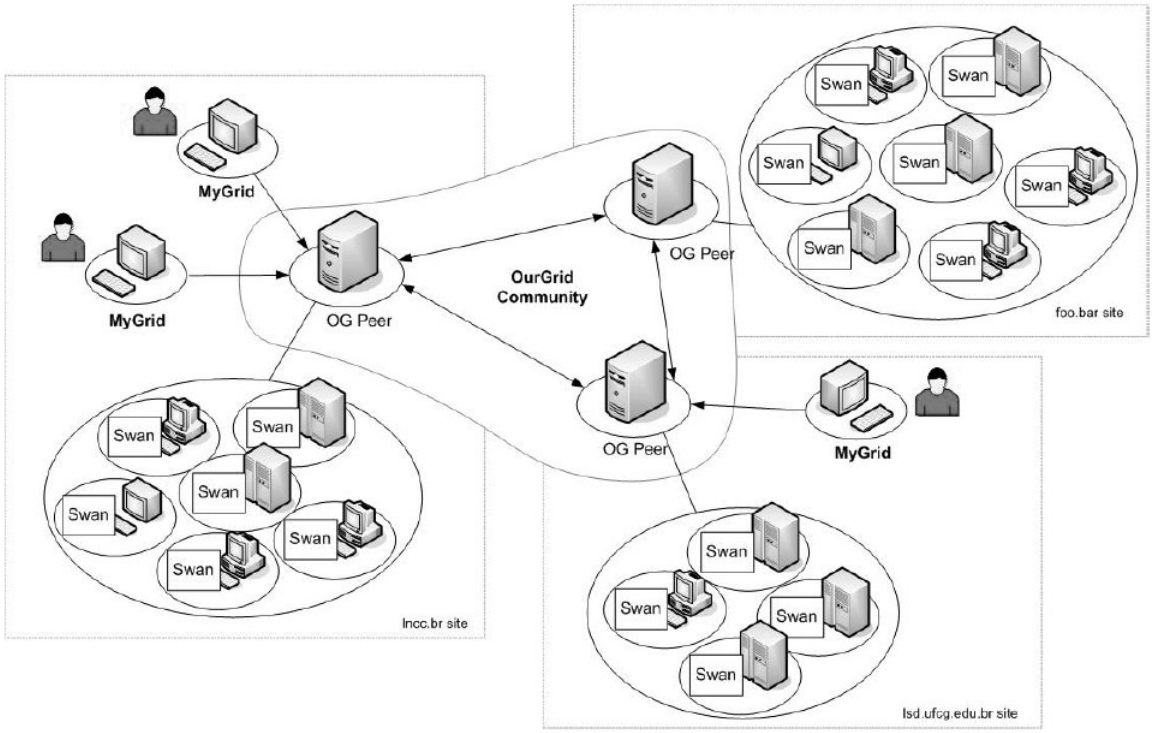


Figura 5.5: Arquitetura do OurGrid

peer. *Jobs*, *tasks* e *grid machines* têm atributos que permitem que o usuário especifique requisitos para os *jobs* e *tasks* e assim permitir que o MyGrid aloque as *tasks* nas *grid machines*. Esta alocação pode incorporar heurísticas de escalonamento para melhorar o desempenho de diferentes tipos de aplicação.

A arquitetura da comunidade OurGrid foi projetada para ser escalável e permitir que milhares de aglomerados possam estar conectados ao sistema. Esta escalabilidade provêm do fato de ela basear-se em redes P2P (*peer-to-peer*), em que cada aglomerado representa um *peer*. Entretanto, redes P2P podem ter o seu desempenho comprometido por causa de *peers* chamados *freeriders* [HCW05]. Um *freerider* é um *peer* que somente consome recursos, nunca contribuindo com a comunidade. Este comportamento poderia ter um impacto negativo no OurGrid, dado que muitos usuários apresentam uma insaciável demanda por recursos computacionais. Para lidar com este problema foi criada uma *rede de favores* (*network of favors*), um mecanismo de alocação de recursos totalmente descentralizado e e autônomo que marginaliza os *freeriders*.

Para que um *peer* tenha acesso aos recursos disponíveis na grade é necessário que cada nó da grade execute uma implementação de uma interface chamada *GridMachine*. A *GridMachine* (ou GuM) define um conjunto mínimo de operações necessárias

para executar uma *task* em um nó. Três implementações de GuM estão atualmente disponíveis no OurGrid: (a) GridScript, que faz uso de aplicativos de linha de comando como o `ssh` e o `scp` para transferir dados e executar as computações da grade, (b) Globus Proxy, permite que nós de uma grade Globus executem computações requisitadas através do OurGrid (criando uma interface entre o Globus e o OurGrid) e (c) *UserAgent*, que é uma implementação Java para executar aplicações em nós da grade. Através dos GuMs o resto do sistema pode abstrair a complexidade de tratar diferentes tipos de recursos. A Tabela 5.1 mostra as cinco operações definidas na interface e suas semânticas. Estas operações são simples, mas suficientes para permitir que uma *task* seja executada em uma *grid machine*.

Operação	**Semântica**
ping()	Verifica a disponibilidade da *grid machine*
run()	Inicia um processo na *grid machine*
putFile()	Armazena arquivos na *grid machine*
getFile()	Recupera arquivos da *grid machine*
kill()	Mata um processo na *grid machine*

Tabela 5.1: Semântica das operações definidas na interface *GridMachine*

No OurGrid, *tasks* de um aglomerado possivelmente executarão em outros aglomerados. Isto traz consigo uma forte preocupação com a segurança das máquinas que colaboram cedendo recursos computacionais à grade, dado que código desconhecido executará nelas. Como uma maneira de proteger as máquinas da grade de aplicações hostis, o OurGrid utiliza um software chamado SWAN (*Sandboxing Without A Name*), uma solução baseada na máquina virtual Xen [Bar03] que isola o código proveniente da grade em uma *sandbox* que não permite o acesso a dados e a dispositivos locais e nem a utilizar a rede.

Para fornecer escalabilidade às suas arquiteturas, o MAG e o OurGrid diferem na abordagem adotada. O OurGrid propõe uma abordagem completamente descentralizada, através do uso de redes P2P. Em contrapartida, o MAG utiliza uma abordagem hierárquica, herdada do projeto Integrade. Como já citado anteriormente, uma abordagem baseada em redes P2P está também sendo explorada pela equipe do projeto Integrade.

O OurGrid utiliza replicação de tarefas para prover tolerância a falhas de aplicações. Porém, o usuário pode, opcionalmente, utilizar softwares de terceiros para

obter o *checkpoint* das aplicações (e.g. a biblioteca de *checkpointing* do Condor). O OurGrid fornece toda a infraestrutura necessária para gerenciar o uso de *checkpoints*, como um armazém estável. Entretanto, o desenvolvedor deve modificar sua aplicação de forma a inserir comandos que permitam a salva e a recuperação do estado de execução da aplicação através da biblioteca de *checkpoints* utilizada. O MAG utiliza unicamente a técnica de *checkpointing* para alcançar a tolerância a falhas de aplicações. Todo o suporte à técnica de *checkpointing* é fornecido pelo próprio MAG, através do MAG/Brakes, e não requer nenhuma codificação específica por parte do desenvolvedor da aplicação.

5.4 CoordAgent

O CoordAgent [FTBK03] é um *middleware* de grade que agrega poder computacional de computadores pessoais conectados à Internet. Foi desenvolvido no Laboratório de Sistemas Distribuídos da Universidade de Washington, Bothell, cujo principal pesquisador é o professor Munehiro Fukuda. Sua infraestrutura é composta por agentes móveis que realizam todas as funções pertinentes à grade, como a busca por recursos disponíveis e a execução de aplicações.

A Figura 5.6 mostra a organização da arquitetura de um aglomerado CoordAgent. Para formar este aglomerado, um moderador (i.e. um administrador da infraestrutura da grade) deve iniciar um *Internet Group*, que compreende a infraestrutura básica responsável por manter as informações das máquinas conectadas à grade. Para isso, o *Internet Group* conta com uma base de dados de máquinas conectadas ao aglomerado. Múltiplos *Internet Groups* podem ser organizados utilizando o *Globus Metacomputing Directory Service* (MDS). Cada usuário deve então criar uma conta de convidado, executar um servidor web em sua máquina, registrar suas informações no *Internet Group* e baixar o mecanismo de execução de aplicações (desenvolvido como um agente móvel). Após entrar no *Intenet Group*, o usuário invoca o agente móvel que exibe a janela através da qual ele poderá solicitar a execução de aplicações. Estas aplicações podem ter sido desenvolvidas utilizando as linguagens C/C++ e Java. O CoordAgent permite a execução de aplicações paralelas na grade utilizando ambas as linguagens: C/C++ podem ser desenvolvidas utilizando MPI e PVM, enquanto que as aplicações Java podem utilizar JPVM[2].

[2] *Java Parallel Virtual Machine* – biblioteca Java que permite a implementação de aplicações paralelas utilizando Java. Fornece uma interface idêntica à da biblioteca PVM original

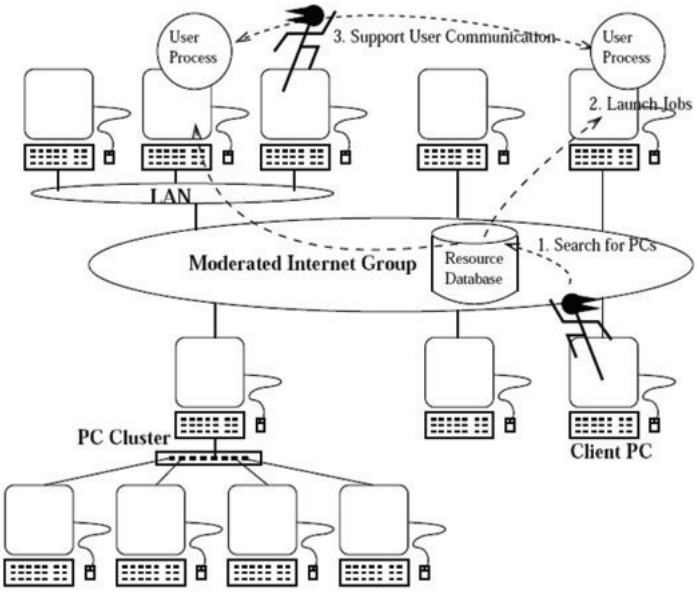


Figura 5.6: Arquitetura do CoordAgent

Após receber a requisição de execução, o agente migra até a base de dados do *Internet Group* procurando por computadores que possam satisfazer a requisição do cliente. Caso não existam máquinas que atendam os requisitos de execução da aplicação, o agente tenta repetidamente visitar bases de dados em outros *Internet Groups* através do MDS, procurando a melhor máquina para atender a requisição. Na migração, o agente utiliza uma técnica de "tunelamento HTTP", na qual o agente é entregue como uma mensagem HTTP e depois é recriado por um *servlet* no destino.

Após localizar uma máquina que possa atender a requisição do usuário, o agente móvel copia todos os arquivos necessários à execução da aplicação, da máquina cliente para a máquina destino. Se a aplicação for uma aplicação que venha requerer mais de uma máquina para sua execução (por exemplo, uma aplicação paralela), o agente cria novos agentes filhos e os envia às outras máquinas. Assim, todos os agentes lançam as aplicações nas máquinas de destino e passam a monitorar suas execuções. Caso uma máquina se torne indisponível durante a execução de uma aplicação, o agente deve migrar sua aplicação correspondente para uma outra máquina disponível. Este processo requer que o estado de execução da aplicação seja capturado, migrado e recuperado no destino. Este mecanismo também é utilizado para prover tolerância a falhas, onde o estado de execução é periodicamente salvo em um repositório que resiste a falhas do sistema.

Para prover a salva do estado de execução de aplicações que executam na grade, o CoordAgent insere (através de um pré-processador) em seu códigos-fonte funções

de captura e recuperação do estado de execução. Este mecanismo é fornecido por pré-compiladores baseados em duas ferramentas: ANTLR [ANT] para aplicações C/C++ e JavaCC [JAV] para aplicações Java.

O CoordAgent, como o MAG, propõe um *middleware* de grade baseado em agentes móveis, provendo mecanismos para a execução de aplicações, migração transparente de código e tolerância a falhas. Ambos permitem a execução de aplicações nativas e Java, sendo que o mecanismo de execução de aplicações nativas do MAG foi reutilizada do Integrade. O MAG e o CoordAgent fornecem diferentes modelos de execução de aplicações, entre os quais estão as aplicações regulares, paramétricas e paralelas. Enquanto o CoordAgent adota MPI e PVM como modelos de programação de aplicações paralelas, o MAG permite a execução de aplicações BSP nativas através do Integrade.

O mecanismo de migração do CoordAgent baseia-se na abordagem de instrumentação de código-fonte para a captura do estado das aplicações em execução. Já o MAG utiliza a instrumentação de *bytecode* que não requer que o código-fonte da aplicação esteja disponível. Esta abordagem normalmente impõe uma menor sobrecarga ao tempo de execução e ao tamanho do *bytecode* da aplicação que a abordagem baseada em código-fonte.

5.5 Organic Grid

O Organic Grid [ORG, CBL05b, CBL05a, CBL04] é um projeto de grade oportunista em desenvolvimento na Universidade de Ohio. Propõe uma abordagem completamente descentralizada, sem a presença de componentes que mantenham informações sobre todos os recursos presentes no sistema. Além disso, utiliza um esquema de escalonamento autônomo inspirado na organização de complexos sistemas biológicos (e.g. a organização das formigas). Através deste esquema de escalonamento é possível atingir um alto grau de escalabilidade, agregando ao sistema milhões de computadores – o que não é possível quando uma abordagem centralizada é utilizada.

O esquema de escalonamento descentralizado utilizado neste projeto foi baseado em um trabalho anteriormente proposto por Kreaseck et al. [KCCF03]. Ambos os trabalhos organizam as computações em redes *overlay* (também chamadas redes *sobrepostas*) [DO03] estruturadas como árvores. Redes *overlay* são estruturas lógicas que

representam "topologias virtuais"[3] estruturadas sobre o protocolo de transporte utilizado, facilitando, por exemplo, buscas determinísticas na rede. As redes *overlay* representam em suas estruturas, os nós da rede, seus vizinhos e as ligações lógicas existentes entre eles. A presença de uma ligação lógica entre dois nós em uma rede *overlay* indica que eles podem comunicar-se diretamente um com o outro. Redes *overlay* são consideradas uma alternativa escalável às atuais redes P2P.

O Organic Grid encapsula as computações submetidas à grade em agentes móveis, ou seja, todas as computações na grade tem um agente responsável por efetuar sua execução. Além de executar aplicações, os agentes do Organic Grid também permitem que nós da grade sejam liberados, caso os usuários reclamem o seu uso. Esta capacidade é fornecida através do arcabouço para migração forte de aplicações *multi-threading* descrito em [CWHB03].

Além de encapsular as aplicações em execução na grade, os agentes do Organic Grid são responsáveis por implementar seu esquema de escalonamento autônomo. A árvore *overlay*, que constitui a base do escalonamento no Organic Grid, é mantida por estes agentes, sendo cada um deles mapeado como um nó da árvore.

O algoritmo de escalonamento do Organic Grid, em linhas gerais, funciona da seguinte maneira: quando uma computação é submetida a partir de um nó A, este divide a tarefa em sub-tarefas menores e passa a executar uma delas. Então, o nó A inicia a montagem da árvore *overlay*, sendo ele mesmo a raiz desta. Outras sub-tarefas são então enviadas a seus *amigos*[4], que também passam a executar sub-tarefas. Assim, é formada uma hierarquia, onde os *amigos* de A tornam-se filhos de A. A cadeia prossegue com os filhos de A, até que toda a árvore seja montada. Nós que recebem solicitações de execução tornam-se filhos dos solicitantes.

À medida que alguns nós concluem a execução de sub-tarefas, seus resultados vão sendo devolvidos a seus pais. Neste interstício, os nós filhos podem ainda solicitar a seus pais mais sub-tarefas a executar. Após o retorno dos resultados de todos os filhos, o nó pai deverá escolher o filho que tiver o maior *throughput* (i.e. os nós que devolvem

[3] A topologia lógica formada pelas redes *overlay* são ditas "virtuais" porque não representam a rede como ela está organizada fisicamente

[4] Nós "*amigos*" são aqueles nós cujos endereços são previamente conhecidos, tendo sidos configurados manualmente ou obtidos através de outros mecanismos como, por exemplo, uma rede P2P de compartilhamento de arquivos [RFH+01]

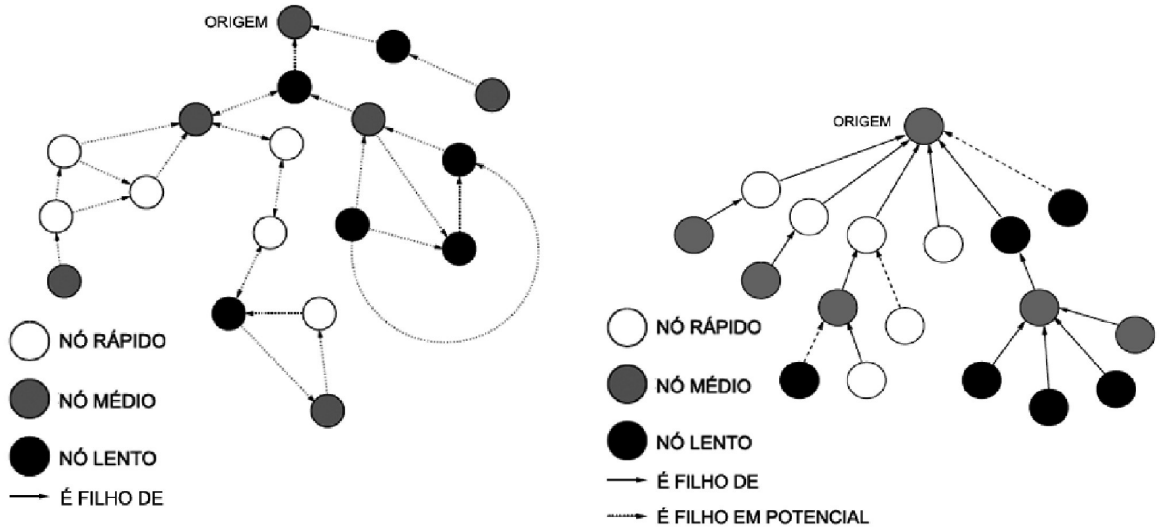
mais rapidamente o resultado das computações) para movê-lo[5] um nível acima na árvore (juntamente com sua sub-árvore, se esta existir), de forma que ele fique mais próximo ao nó A, ou seja, a raiz da árvore. Ao assumir uma nova posição na árvore, este nó ganha de seus novos pais o estado de "*filho em potencial*", e permanece neste estado até que seu *throughput* seja avaliado por seu novo pai. De maneira análoga, o filho com o menor *throughput* será removido da árvore, e poderá no futuro, tentar voltar a compor a árvore. Futuramente, ao tentar se re-integrar à árvore, os nós excluídos também ganharão o estado de "*filhos em potencial*", até que tenham os seus *throughputs* novamente avaliados. Como pode ser percebido, este algoritmo tende a fazer com que os nós mais rápidos fiquem mais próximos ao nó que solicitou a execução da computação, e que os nós lentos sejam removidos da estrutura, melhorando assim o tempo de resposta da grade às solicitações de execução de aplicações. Este esquema é inspirado em sistemas biológicos compostos por milhões de organismos encontrados na natureza que, de maneira autônoma, produzem complexos padrões de formação, organizando-se da melhor maneira para executarem seu trabalho.

O algoritmo de escalonamento utilizado no Organic Grid, apesar de ser extremamente escalável, pode não resultar em um aproveitamento dos recursos tão bom quanto a abordagem centralizada. Além disso, o desempenho deste esquema depende diretamente da formação inicial da árvore. A Figura 5.7(a) mostra a formação inicial de uma grade Organic Grid. Já a Figura 5.7(b) apresenta a formação da árvore *overlay* da mesma grade após a execução de algumas rodadas do algoritmo descrito.

A versão atual do Organic Grid suporta a execução de duas classes de aplicações: regulares e paramétricas. Entretanto, os desenvolvedores do Organic Grid demostraram, utilizando uma implementação tolerante a falhas do algoritmo de Cannon para multiplicação de matrizes, como esta infraestrutura poderia ser facilmente adaptada para permitir a execução de aplicações paralelas que utilizam comunicação entre suas sub-tarefas [CBL04].

O Organic Grid propõe um esquema de escalonamento descentralizado em que as decisões de escalonamento independem da existência de conhecimento global sobre os recursos disponíveis na grade. Esta abordagem cria um alto grau de escalabilidade ao

[5]O verbo "mover" é aqui empregado no sentido de alterar as ligações hierárquicas que existem entre os nós. Por exemplo, na hierarquia X é pai de Y, e Y é pai de Z, mover Z um nível acima significa torná-lo filho direto de X, ou seja: X é pai de Y e Z



(a) Configuração inicial de um aglomerado do Organic Grid

(b) Organização dos nós após alguns ciclos de execução do algoritmo de escalonamento do Organic Grid

Figura 5.7: Grade Organic Grid

sistema, uma vez que não há a presença de elementos centralizadores. Em contrapartida, o MAG adotou o mecanismo de escalonamento fornecido pelo projeto Integrade, que provê um algoritmo de escalonamento baseado em uma hierarquia de aglomerados. Este esquema hierárquico permite que os recursos disponíveis no sistema sejam melhor aproveitados, uma vez que cada aglomerado detém conhecimento sobre a disponibilidade de seus recursos. Este esquema de escalonamento também favorece escalabilidade ao sistema, dado que as informações relativas a recursos não precisam necessariamente ser propagadas para todos os participantes da hierarquia, de acordo com o protocolo proposto em [MK02]. A utilização desta abordagem permite que uma enorme quantidade de máquinas seja agregada ao sistema.

Enquanto o Organic Grid permite somente a execução de aplicações regulares e paramétricas, o MAG reutiliza do Integrade o suporte à execução de aplicações paralelas BSP. Atualmente este suporte está disponível apenas às aplicações nativas, através da biblioteca BSP do Integrade. Futuramente, pretende-se adicionar ao MAG o suporte a outros modelos de programação paralela, como MPI e PVM.

6 Conclusões e Trabalhos Futuros

Questões como gerenciamento e alocação eficiente de recursos distribuídos, escalonamento dinâmico de tarefas de acordo com a disponibilidade de recursos, tolerância a falhas de nós e aplicações individuais, mobilidade de código, gerenciamento eficiente da execução de aplicações, comunicação entre computações cooperantes executando em nós distintos e diversos aspectos relacionados à segurança, constituem importantes desafios para a construção de um *middleware* de grade.

Este trabalho apresentou o MAG, um middleware de grade baseado na tecnologia de agentes de software. O MAG executa aplicações na grade carregando dinamicamente o código da aplicação em um agente móvel. O agente do MAG pode ser dinamicamente realocado entre nós da grade através de um mecanismo de migração transparente de aplicações denominado MAG/Brakes, permitindo que nós não dedicados sejam utilizados como parte da infraestrutura da grade. O MAG também fornece um mecanismo de tolerância a falhas de aplicações. Este mecanismo é essencial em ambientes de grades, dado que evita a perda de tempo computacional realizado durante longos períodos. Atualmente, o mecanismo de tolerância a falhas do MAG trata somente falhas de colapso de nós, e é baseado na abordagem de *checkpoint*, em que o estado de execução da aplicação é periodicamente salvo em um armazém estável. Assim, se uma falha ocorrer, as computações podem ser recuperadas a partir de seu último *checkpoint*. Um outro relevante mecanismo fornecido pela arquitetura do MAG é o de gerenciamento de execuções. Através da análise dos dados obtidos por este mecanismo é possível aos desenvolvedores da infra-estrutura de grade e aos administradores das mesmas a tomada de decisões relativas ao projeto do *middleware* e a configuração do ambiente de execução, respectivamente. Para tanto, o MAG disponibiliza uma ferramenta que permite a visualização dos dados de execução de aplicações forma gráfica e intuitiva (*ClusterApplicationViewer*), bem como a geração de estatísticas úteis a administradores e desenvolvedores de grades.

Testes de avaliação de desempenho foram realizados com o intuito de avaliar o impacto dos componentes de gerenciamento de recursos local e global (GRM e LRM) e do *AgentHandler*, utilizando como métrica o uso relativo de CPU. Constatou-se que a custo de CPU acrescido pela presença destes componentes em nós da grade é muito pequeno

e pode ser muitas vezes negligenciado. Além disso, um outro experimento foi realizado com o objetivo de medir a eficácia do *middleware* de grade na execução de aplicações paramétricas. A partir desta medição foi possível perceber que o tempo de execução desta classe de aplicações é diretamente proporcional ao número de nós que compõem o aglomerado.

6.1 Contribuições

Além do desenvolvimento de um middleware complementar à plataforma de grade Integrade, conforme descrito no capítulo 2, outra importante contribuição deste trabalho foi investigar a adequação do paradigma de agentes de software para o desenvolvimento de um *middleware* de grade. A nossa conclusão é que esta tecnologia efetivamente contribuiu para simplificar o projeto e implementação de um sistema distribuído tão complexo como o que compõe uma grade de computadores. Esta constatação é derivada da análise de características específicas da tecnologia de agentes de software, como as seguintes:

- Mobilidade – a mobilidade dos agentes colaborou para fornecer o suporte a nós não dedicados na grade. Assim, quando um usuário requisita o uso exclusivo de seu recurso, as computações que nele executam são migradas através de um mecanismo de migração forte de aplicações. O mecanismo de migração forte do MAG, o MAG/Brakes, foi desenvolvido para superar a limitação original da plataforma JADE que só permite a migração fraca de aplicações;

- Autonomia – esta característica foi explorada no desenvolvimento de diversos mecanismos do MAG, como os mecanismos de execução de aplicações e de tolerância a falhas. Os agentes que compõem a arquitetura do MAG têm o controle sobre suas ações, não requerendo intervenção alguma por parte dos clientes que os iniciaram. Por exemplo, ao ser criado em resposta a uma solicitação de execução de aplicação, o `MagAgent` conduz todo este processo de maneira autônoma: baixa o *bytecode* e os arquivos de entrada da aplicação, instancia, executa e monitora a mesma, salva periodicamente seu estado e notifica o término de sua execução à grade. A autonomia do agente MAG simplificou o desenvolvimento dos demais componentes que compõem a grade;

- Cooperação – no MAG, esta característica é empregada para a realização de tarefas que envolvam comunicação e/ou negociação entre agentes com diferentes papéis, no fornecimento de serviços à grade. Por exemplo, para prover tolerância a falhas de aplicações que executam na grade, agentes com diferentes papéis interagem em torno de um objetivo comum, como é o caso dos agentes `AgentRecover` e `ExecutionManagementAgent` que trocam informações à respeito das aplicações que falharam para proceder o processo de recuperação. Além disso, esta característica faz com que a complexidade dos agentes seja reduzida, uma vez que cada agente fornece funcionalidades específicas e bem definidas. Observamos ainda que diversos protocolos de interação já bem estabelecidos pela comunidade que utiliza agentes e padronizados pela FIPA são adequados para diversos tipos de interação entre componentes da grade, como a negociação e a descoberta de recursos e a notificação de mudanças na disponibilidade de recursos de nós. Como o MAG adotou uma arquitetura mista, reutilizando componentes do Integrade, estas funcionalidades não foram exploradas por completo;

- Segurança – várias plataformas de agentes móveis fornecem mecanismos de proteção e segurança que podem ser aproveitados por seus agentes, como mecanismos de autenticação e autorização. No contexto do MAG esta característica não foi explorada, dado que a abordagem de reutilizar sempre que possível os componentes do Integrade foi seguida.

Além destas características específicas, ressalta-se ainda que pode-se observar na prática o poder de abstração dos agentes para a modelagem de problemas complexos. Devido ao uso deste paradigma, a análise e o projeto do middleware MAG foi realizada em um nível de abstração mais elevado em comparação ao desenvolvimento baseado em componentes. Enquanto que a abordagem baseada em agentes é focada na atribuição de responsabilidades e na definição de papéis a agentes do sistema, a análise e projeto baseada em componentes segue um modelo no qual interfaces de métodos devem ser definidas como forma de disponibilizar os serviços providos pelos mesmos.

Os resultados obtidos através deste trabalho foram divulgados através das seguintes publicações:

- *MAG: A Mobile Agent based Computational Grid Platform* [LSS05]: artigo completo

publicado no *4th International Conference on Grid and Cooperative Computing* que descreve a arquitetura geral e a implementação do projeto MAG;

- *Migration Transparency in a Mobile Agent Based Computational Grid* [LS05a]: artigo completo publicado no *1st WSEAS International Symposium on GRID COMPUTING* que descreve a implementação da primeira versão do arcabouço MAG/Brakes;

- *Strong Migration in a Grid based on Mobile Agents* [LS05b]: artigo completo publicado no periódico *WSEAS Transactions On Systems*, é uma versão estendida do artigo anterior;

- *Fault Tolerance in a Mobile Agent Based Computational Grid* [LS06]: artigo completo publicado no *AgentGrid 2006 – 4th International Workshop on Agent based Grid Computing*, evento do *6th ACM/IEEE International Symposium on Cluster Computing and the Grid (CCGrid'2006).*

6.2 Demais Áreas de Pesquisa

O MAG foi projetado de maneira a permitir a adição progressiva de novas funcionalidades. Outros membros do projeto MAG estão desenvolvendo trabalhos para superar limitações e fornecer melhorias à versão atual. A seguir são apresentados alguns trabalhos de pesquisa em andamento que serão agregados à infraestrutura básica do MAG:

- *Grade pervasiva*: a proposta deste projeto de pesquisa é estender a arquitetura do projeto MAG para permitir a utilização dos serviços da grade por parte de usuários móveis. A mobilidade do usuário é baseada em dois mecanismos, que formam as duas frentes deste projeto: (a) acesso aos serviços da grade utilizando a Internet (serviços web) e (b) acesso aos serviços da grade utilizando dispositivos móveis (plataforma PalmOS), tornando a grade uma extensão dos recursos destes dispositivos;

- *Grade de dados*: diante da crescente demanda por computações de grandes volumes de dados em vários domínios como, por exemplo, análise experimental e simulação em física de alta-energia, modelagem climática, engenharia de terremotos e astronomia, planeja-se desenvolver serviços que promovam armazenamento, gerência de acesso e transferência confiável e rápida, bem como a publicação e a descoberta de dados na grade. Para fornecer estas capacidades à arquitetura do MAG, é necessário

que sejam desenvolvidos serviços específicos para lidar com dados e metadados na grade. Para avaliar o desempenho do *middleware* proposto, uma aplicação de referência está sendo desenvolvida. Esta aplicação é um sistema PACS (*Picture Archival Communication System*) que permite a recuperação de imagens a partir de seu conteúdo pictórico;

- *Tolerância a falhas de componentes da grade:* a atual infraestrutura de tolerância a falhas do MAG foi projetada para recuperar aplicações da grade através da técnica de *checkpointing*. Entretanto, nenhum suporte é atualmente dado à falha de componentes da própria grade. De forma a evitar o colapso de toda a infraestrutura do *middleware* por falha de alguns de seus componentes, devem ser fornecidos mecanismos adequados para tratar a questão da tolerância a falhas de componentes da grade.

6.3 Trabalhos Futuros

Além das funcionalidades implementadas no escopo deste trabalho e das demais áreas de pesquisa, pretende-se adicionar vários outros mecanismos à arquitetura do MAG. Alguns destes são:

- *Proteção contra aplicações hostis e defeituosas*: é necessário garantir aos usuários que cedem suas máquinas à grade, que aplicações que executem em suas máquinas não causem danos aos seus recursos e aos seus dados. Para evitar que prejuízos desta natureza recaiam sobre os usuários destas máquinas passa a ser necessário limitar o ambiente de execução no qual estas aplicações executam. Assim, uma possível abordagem para resolver esta questão é o emprego de *sandboxes*, que evitam a ocorrência deste tipo de problemas de segurança;

- *Balanceamento de carga*: de forma a melhor aproveitar o mecanismo de migração forte do MAG é necessário que exista um mecanismo que efetue o balanceamento de carga das aplicações que executam na grade. A existência deste mecanismo evitaria que máquinas com grande capacidade de processamento que por ventura venham a se integrar à grade permaneçam ociosas em detrimento das máquinas com menor capacidade que já estejam executando computações;

- *Suporte a aplicações paralelas MPI*: ambientes de grade, por agregarem enormes quantidades de recursos, são propícios à execução de computações paralelas. Para tanto, é necessário que seja disponibilizada uma biblioteca de programação paralela a ser utilizada pelos desenvolvedores que queiram utilizar a grade para executar suas aplicações. Vários modelos e implementações de bibliotecas de programação paralela existem hoje no mercado, entretanto, após uma análise entre as opções disponíveis, decidiu-se utilizar a biblioteca MPI [GLDS96] (*Message Passing Interface*) como referência. Isto se deve ao fato de a MPI ser extremamente popular, dado que a grande maioria dos desenvolvedores de aplicações paralelas a conhecem;

- *Melhorias no arcabouço MAG/Brakes*: o mecanismo de serialização do estado de execução de *threads* Java utilizado no MAG, o MAG/Brakes, foi desenvolvido em uma versão preliminar que permite que somente aplicações compostas por uma única *thread* possam ser migradas. Desta forma, é necessário que este arcabouço seja modificado para permitir a migração de aplicações compostas por múltiplas *threads*. Além disso, um outro aspecto a ser implementado neste arcabouço é migração de recursos, que permite que o ambiente de execução das aplicações Java seja mantido após a migração da mesma;

- *Tratamento de falhas de aplicações individuais*: o mecanismo de tolerância a falhas do MAG, em sua versão atual, foi concebido para tratar somente a ocorrência de falhas de colapso de nós da grade, sendo que este sistema não detecta a falha de uma aplicação individual. Mecanismos que forneçam esta capacidade são extremamente importantes, e devem ser agregados à infraestrutura do MAG;

- *Utilização de adaptação dinâmica no mecanismo de tolerância a falhas*: a tolerância a falhas de aplicações, por envolver diversos aspectos, é extremamente propícia à utilização de adaptação dinâmica para tornar sua implementação mais eficiente.

- *Personalização da plataforma JADE*: a plataforma de agentes móveis utilizada no projeto MAG (plataforma JADE), contém uma grande gama de serviços que podem ser utilizados por seus usuários. No entanto, nem todos estes serviços são utilizados no contexto do MAG. De forma a economizar recursos de memória e processamento, esta plataforma deve ser personalizada de forma a remover serviços que não sejam utilizados pelos agentes do MAG.

A Especificação das Máquinas do LSD

Máquina	Processador	Memória RAM	Sistema
cezanne	Intel Pentium 4 2.8 GHz	1 GB	Linux 2.6.10
gauguin	Intel Pentium 4 2.8 GHz	1 GB	Linux 2.6.10
renoir	Intel Pentium 4 2.8 GHz	1 GB	Linux 2.6.10
picasso	Intel Pentium 4 2.8 GHz	1 GB	Linux 2.6.10

Referências Bibliográficas

[AAA+04] Naveed Ahmad, Arshad Ali, Ashiq Anjum, Tahir Azim, Julian J. Bunn, Ali Hassan, Ahsan Ikram, Frank van Lingen, Richard McClatchey, Harvey B. Newman, Conrad Steenberg, Michael Thomas, and Ian Willers. Distributed Analysis and Load Balancing System for Grid Enabled Analysis on Hand-held devices using Multi-Agents Systems. *CoRR*, cs.DC/0407013, 2004.

[ACBR03] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. OurGrid: An approach to easily assemble grids with equitable resource sharing. In *Proc. of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2003.

[AF89] Yeshayahu Artsy and Raphael A. Finkel. Designing a process migration facility: The charlotte experience. *IEEE Computer*, 22(9):47–56, 1989.

[AGIa] Agile Alliance website. http://www.agilealliance.org. Acessado em: 01/12/2005.

[AGIb] Agile Manifesto website. http://www.agilemanifesto.org. Acessado em: 01/12/2005.

[AMMV04] Rocco Aversa, Beniamino Di Martino, Nicola Mazzocca, and Salvatore Venticinque. Terminal-Aware Grid Resource and Service Discovery and Access Based on Mobile Agents Technology. *pdp*, vol. 00:40, 2004.

[ANT] ANTLR Website. http://www.antlr.org. Acessado em: 01/12/2005.

[Bar03] Barham, P. et al. Xen and the Art of Virtualization. In *SOPS*, 2003.

[BBL00] M. Baker, R. Buyya, and D. Laforenza. The grid: International efforts in global computing. *In Proc. of the International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*, 2000.

[BEO] Beowulf website. http://www.beowulf.org. Acessado em: 01/12/2005.

[BHM+04] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture. *World Wide Web Consortium*, Fev. 2004.

[BHP03] Sara Bouchenak, Daniel Hagimont, and Noël De Palma. Efficient java thread serialization. In *2nd ACM International Conference on Principles and Practice of programming in Java (ACM PPPJ'03)*, Kilkenny, Ireland, June 2003.

[Bur92] Steve Burbeck. Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc). 1992. http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html.

[Buy02] Rajkumar Buyya. Understanding the Grid. *Grid Computing Planet Conference*, 2002.

[Cao04] J. Cao. Self-organizing agents for grid load balancing. In *Proceedings of Fifth IEEE/ACM International Workshop on Grid Computing*, pages 388–395, Nov "2004".

[CBL04] A.J. Chakravarti, G. Baumgartner, and M. Lauria. Application-specific scheduling for the Organic Grid. In *Proceedings of Fifth IEEE/ACM International Workshop on Grid Computing*, pages 146–155, Nov. 2004.

[CBL05a] A.J. Chakravarti, G. Baumgartner, and M. Lauria. The organic grid: Self-organizing computation on a peer-to-peer network. *To appear in IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 35(No. 3), May 2005.

[CBL05b] A.J. Chakravarti, G. Baumgartner, and M. Lauria. The organic grid: Self-organizing computational biology on desktop grids. *To appear in A. Zomaya (ed.), Parallel Computing for Bioinformatics*, 2005.

[CCSS04] Antonio Chella, Massimo Cossentino, Luca Sabatucci, and Valeria Seidita. From PASSI to Agile PASSI: Tailoring a Design Process to Meet New Needs. In *IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'04)*, pages 471–474, 2004.

[CFF+04] K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, T. Maquire, D. Snelling, and S. Tuecke. From open grid services infrastructure to ws-resource framework: Refactoring & evolution. *Global Grid Forum Draft Recommendation*, May 2004.

[CGKG04] Raphael Y. de Camargo, Andrei Goldchleger, Fabio Kon, and Alfredo Goldman. Checkpointing based rollback recovery for parallel applications on the integrade grid middleware. *5th ACM/IFIP/USENIX International Middleware Conference*, 2004.

[CGM+04] Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, Jeffrey Schlimmer, and Sanjiva Weerawarana. Web services description language (wsdl) version 2.0 part 1: Core language. *World Wide Web Consortium*, Mar. 2004.

[CJKN02] J. Cao, S. A. Jarvis, D. J. Kerbyson, and G. R. Nudd. ARMS: An agent-based resource management system for grid computing. *Scientific Programming 10*, 2002.

[CLZ00] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Weak and strong mobility in mobile agent applications. In *Proceedings of the 2nd International Conference and Exhibition on The Practical Application of Java (PA JAVA 2000)*, Manchester (UK), April 2000.

[CON] Condor website. http://www.cs.wisc.edu/condor. Acessado em: 01/12/2005.

[CPC+03] Walfredo Cirne, Daniel Paranhos, Lauro Costa, Elizeu Santos-Neto, Francisco Brasileiro, Jacques Sauve, Fabricio A. B. Silva, Carla O. Barros, and Cirano Silveira. Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach. In *In Proc. International Conference on Parallel Processing (ICPP'03)*, page 407, 2003.

[CPV97] A. Carzaniga, G. P. Picco, and G. Vigna. Designing distributed applications with mobile code paradigms. In R. Taylor, editor, *Proc. 19th Conf. Software Eng. (ICSE '97)*, pages 22–32. ACM Press, 1997.

[CSJN05] Junwei Cao, Daniel P. Spooner, Stephen A. Jarvis, and Graham R. Nudd. Grid load balancing using intelligent agents. *Future Gener. Comput. Syst.*, 21(1):135–149, 2005.

[CSS03] Massimo Cossentino, Luca Sabatucci, and Valeria Seidita. Spem description of the passi process rel. 0.3.6. Technical Report RT-ICAR-20-03, Consiglio Nazionale delle Ricerche Istituto di Calcolo e Reti ad Alte Prestazioni, December 2003.

[CSSC03] M. Cossentino, L. Sabatucci, S. Sorace, and A. Chella. Patterns reuse in the PASSI methodology. In *Proceedings of the Fourth International Workshop Engineering Societies in the Agents World (ESAW'03)*, pages 29–31, Imperial College London, UK, October 2003.

[CST+02] Junwei Cao, D.P. Spooner, J.D. Turner, S.A. Jarvis, D.J. Kerbyson, S. Saini, and G.R. Nudd. Agent-based resource management for grid computing. In *Cluster Computing and the Grid 2nd IEEE/ACM International Symposium CCGRID2002*, pages 323–324, May 2002.

[CTJV00] Tim Coninx, Eddy Truyen, Wouter Joosen, and Pierre Verbaeten. On the use of threads in mobile object systems. *ECOOP'2000 Workshop on Mobile Object Systems*, 2000.

[CWHB03] Arjav J. Chakravarti, Xiaojin Wang, Jason O. Hallstrom, and Gerald Baumgartner. Implementation of Strong Mobility for Multi-Threaded Agents in Java. In *2003 International Conference on Parallel Processing (ICPP '03). IEEE Computer Society Press.*, pages 321–330, Koahsiung, Taiwan, 6-9 October 2003.

[Dah01] Markus Dahm. Byte code engineering with the bcel api. Technical report, Freie Universität Berlin, 2001.

[DO91] Fred Douglis and John K. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[DO03] D. Doval and D. O'Mahony. Overlay networks: A scalable alternative for P2P. *IEEE Internet Computing*, 7:79–82, July-Aug 2003.

[EAWJ96] Mootaz Elnozahy, Lorenzo Alvisi, Yi-min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. 1996.

[ELD+96] D.H.J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of condors: Load sharing among workstation clusters. *Future Generation Computer Systems*, 12(53):65, 1996.

[FGN+97] I. Foster, J. Geisler, W. Nickless, W. Smith, and S. Tuecke. Software Infrastructure for the I-WAY High Performance Distributed Computing Experiment. In *Proc. 5th IEEE Symposium on High Performance Distributed Computing*, pages 562–571, 1997.

[FIP] Foundation for Intelligent Physical Agents website. http://www.fipa.org. Acessado em: 01/12/2005.

[FK99] I. Foster and C. Kesselman. *The Grid: Blueprint for a Future Computing Infrastructure*, chapter Globus: A Toolkit-Based Grid Architecture, pages 259–278. Morgan Kaufmann Publisher, 1999.

[FKNT02] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. *In Open Grid Service Infrastructure WG, Global Grid Forum*, 2002.

[FKT01] I. Foster, C. Kesselman, and S. Tueke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputing Applications*, 2001.

[FKTT98] Ian Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A Security Architecture for Computational Grids. In *In Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 83–92, 1998.

[Fou01] Foundation for Intelligent Physical Agents - FIPA. *FIPA RDF Content Language Specification*, Aug. 2001. Document #: XC00011B. http://www.fipa.org/specs/fipa00011.

[Fou02a] Foundation for Intelligent Physical Agents - FIPA. *FIPA ACL Message Structure Specification*, Dec. 2002. Document #: SC00061G. http://www.fipa.org/specs/fipa00061/SC00061G.html.

[Fou02b] Foundation for Intelligent Physical Agents - FIPA. *FIPA Contract Net Interaction Protocol Specification*, Dec. 2002. Document #: SC00029H. http://www.fipa.org/specs/fipa00029.

[Fou02c] Foundation for Intelligent Physical Agents - FIPA. *FIPA Query Interaction Protocol Specification*, Dec. 2002. Document #: SC00027H. http://www.fipa.org/specs/fipa00027.

[Fou02d] Foundation for Intelligent Physical Agents - FIPA. *FIPA SL Content Language Specification*, Dec. 2002. Document #: SC00008I. http://www.fipa.org/specs/fipa00008.

[Fou02e] Foundation for Intelligent Physical Agents - FIPA. *FIPA Subscribe Interaction Protocol Specification*, Dec. 2002. Document #: SC00035H. http://www.fipa.org/specs/fipa00035.

[Fou03] Foundation for Intelligent Physical Agents - FIPA. *FIPA Request Interaction Protocol Specification*, Dec. 2003. Document #: SC00026H. http://www.fipa.org/specs/fipa00026.

[FPV98] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[FTBK03] M. Fukuda, Y. Tanaka, L. F. Bic, and S. Kobayashi. A mobile-agent-based pc grid. *IEEE Computer*, 2003.

[Fun98] Stefan Funfrocken. Transparent Migration of Java-Based Mobile Agents: Capturing and Reestablishing the State of Java Programs. In *Proc. of the Second International Workshop on Mobile Agents*, pages 26–37, Stuttgart, Germany, September 1998.

[GCLS05] Maurício Guará Garcia, Letícia de Fátima Silva Corrêa, Rafael Fernandes Lopes, and Francisco José da Silva Silva. Uma ferramenta para a visualização de aplicações que executam em uma grade computacional. *Cadernos de pesquisa - Universidade Federal do Maranhão*, v. 16(n. 1):47, 2005.

[GKG+04] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G. C. Bezerra. Integrade: Object-oriented grid middleware leveraging idle computing power of desktop machines. *Concurrency and Computation: Practice & Experience. Vol. 16, pp. 449-459*, 2004.

[GKGF03] A. Goldchleger, F. Kon, A. Goldman, and M. Finger. Integrade: Object-oriented grid middleware leveraging idle computing power of desktop machines. *ACM/IFIP/USENIX International Workshop on Middleware for Grid Computing. Rio de Janeiro, Brazil*, 2003.

[GLDS96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22(6):789–828, 1996.

[GLO] Globus website. http://www.globus.org. Acessado em: 01/12/2005.

[Glo03] Global Grid Forum. *Open Grid Services Infrastructure (OGSI) Version 1.0*, Jun. 2003. GFD-R-P.15 (Proposed Recommendation).

[Gol90] Adele Goldberg. Information models, views, and controllers. *Dr. Dobb's J.*, 15(7):54–61, 1990.

[Gol04] Andrei Goldchleger. InteGrade: Um Sistema de Middleware para Computação em Grade Oportunista (InteGrade: a Middleware System for Opportunistic Grid Computing). Master's thesis, Department of Computer Science - University of São Paulo, December 2004. in Portuguese.

[Gra03] P. Gradwell. Grid Scheduling with Agents. 2003.

[HCW05] D. Hughes, G. Coulson, and J. Walkerdine. Free Riding on Gnutella Revisited: the Bell Tolls? *Submitted to IEEE Distributed Systems*, 2005. Draft at http://www.comp.lancs.ac.uk/computing/users/hughesdr/papers/freeriding.pdf.

[HMS+98] J. M. D. Hilll, B. Mccolll, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisselin. Bsplib: The bsp programming library. 1998.

[HSD73] R. M. Haralick, K. Shanmugam, , and I. Dinstein. Textural features for image classification. *IEEE Trans. Syst., Man, Cybern., SMC-3*, pages 610–621, 1973.

[IKKW00] Torsten Illmann, Frank Kargl, Tilmann Krueger, and Michael Weber. Migration in Java: problems, classifications and solutions. In *MAMA'2000*, Wollongong, Australia, 2000.

[IKKW01] Torsten Illmann, Tilmann Krueger, Frank Kargl, and Michael Weber. Transparent migration of mobile agents using the Java Platform Debugger Architecture. In *Lecture Notes in Computer Science*, volume 2240, page 198, Jan 2001.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley - Interscience, New York, NY, April 1991.

[JAV] JavaCC Website. http://www.webgain.com/products/javacc. Acessado em: 01/12/2005.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[KBM02] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Softw. Pract. Exper.*, 32(2):135–164, 2002.

[KCCF03] B. Kreaseck, L. Carter, H. Casanova, and J. Ferrante. Autonomous protocols for bandwidth-centric scheduling of independent-task applications. In *In Proceedings of the International Parallel and Distributed Processing Symposium*, pages 23–25, Apr. 2003.

[LLM88] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[LS05a] Rafael Fernandes Lopes and Francisco José da Silva Silva. Migration Transparency in a Mobile Agent Based Computational Grid. In *Proceedings of the 5th WSEAS Int. Conf. on SIMULATION, MODELING AND OPTIMIZATION. 1st WSEAS International Symposium on GRID COMPUTING*, pages 31–36, Corfu, Greece, August 2005.

[LS05b] Rafael Fernandes Lopes and Francisco José da Silva Silva. Strong Migration in a Grid based on Mobile Agents. *WSEAS Transactions On Systems*, Volume 4(Issue 10):1687–1694, October 2005.

[LS06] Rafael Fernandes Lopes and Francisco José da Silva Silva. Fault Tolerance in a Mobile Agent Based Computational Grid. In *AgentGrid2006: 4th International Workshop on Agent Based Grid Computing. IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'06)*, Singapore, May 2006. IEEE Computer Society Press.

[LSF] Platform LSF website. http://www.platform.com/Products/Platform.LSF.Family. Acessado em: 01/12/2005.

[LSS05] Rafael Fernandes Lopes, Francisco José da Silva Silva, and Bysmarck Barros de Sousa. MAG: A Mobile Agent based Computational Grid Platform. In *Proceedings of the 4th International Conference on Grid and Cooperative Computing*, Lecture Notes in Computer Science (LNCS Series), Beijing, November 2005. Springer-Verlag. to appear.

[MAG] MAG website. http://www.lsd.ufma.br/mag. Acessado em: 01/12/2005.

[Mar04] Martino, Beniamino Di and Rana, Omer F. Grid performance and resource management using mobile agents. pages 251–263, 2004.

[MAU] Maui Scheduler Open Cluster Software website. http://mauischeduler.sourceforge.net. Acessado em: 01/12/2005.

[MCBS03] Raissa Medeiros, Walfredo Cirne, Francisco Brasileiro, and Jacques Sauvé. Faults in Grids: Why are they so bad and What can be done about it? In *Grid Computing, 2003. Proceedings. Fourth International Workshop*, pages 18–24, Nov. 2003.

[MK02] Jeferson Roberto Marques and Fabio Kon. Gerenciamento de Recursos Distribuídos em Sistemas de Grande Escala. In *Anais do XX Simpósio Brasileiro de Redes de Computadores (SBRC 2002)*, pages 800–813, Búzios, Maio "2002".

[MOMB04] D.G.A. Mobach, B.J. Overeinder, O. Marin, and F.M.T. Brazier. Lease-based Decentralized Resource Management in Open Multi-Agent Systems. In *Proceedings of the Second European Workshop on Multi-Agent Systems (EUMAS'04)*, pages 459–464, 2004.

[MWL02] Ricky K. K. Ma, Cho-Li. Wang, and Francis C. M. Lau. M-JavaMPI: A Java-MPI Binding with Process Migration Support. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 255, Washington, DC, USA, 2002. IEEE Computer Society.

[MYG] MyGrid website. http://www.ourgrid.org. Acessado em: 01/12/2005.

[NIR04] Muhammad Kamran Naseem, Sohail Iqbal, and Khalid Rashid. Implementing strong code mobility. *Information Technology Journal*, 3(2):188–191, 2004.

[Obj00] Object Management Group. *Trading Object Service Specification*, Junho 2000. OMG document formal/00-06-27, version 1.0.

[Obj02a] Object Management Group, Needham, MA. *CORBA v3.0 IDL Syntax and Semantics Specification*, Jul. 2002. OMG Document 02-06-39.

[Obj02b] Object Management Group, Needham, MA. *CORBA v3.0 Specification*, Jul. 2002. OMG Document 02-06-33.

[OPB00] J. Odell, H. Parunak, and B. Bauer. Extending uml for agents. In *In Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence.*, 2000.

[OPE] Open Portable Batch System website. http://www.openpbs.org. Acessado em: 01/12/2005.

[ORG] Organic Grid website. http://bit.csc.lsu.edu/ gb/OrganicGrid. Acessado em: 01/12/2005.

[OWSB02] B.J. Overeinder, N.J.E. Wijngaards, M. van Steen, and F.M.T. Brazier. Multi-Agent Support for Internet-Scale Grid Management. In *Proceedings of the AISB'02 Symposium on AI and Grid Computing*, pages 18–22, April 2002.

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and R. Shenker. A scalable content addressable network. In *In Proceedings of ACM SIGCOMM'01*, 2001.

[RLS98] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of*

the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7), Chicago, IL, July 1998.

[RR05] Vladimir Moreira Rocha and Eric Ross. Uma Estrutura Escalável e Eficiente para Buscas por Intervalo sobre DHTs nas Redes P2P. In *Anais do I Workshop de Peer-to-Peer (WP2P). XVIII Simpósio Brasileiro de Redes de Computadores (SBRC 2005)*, Maio "2005".

[Sil97] Da Silva, Miguel Mira. *Mobility and Persistence*, chapter Mobile Object Systems. LNCS 1222, pages 157–175. Springer-Verlag, 1997.

[SMY99] Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. A Simple Extension of Java Language for Controllable Transparent Migration and Its Portable Implementation. In *Coordination Models and Languages*, pages 211–226, 1999.

[SSY00] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. Bytecode transformation for portable thread migration in java. In *ASA/MA*, pages 16–28, 2000.

[Sta02] W. Stallings. *Cryptography and Network Security.* Prentice-Hall, 2002.

[Sue00] Takashi Suezawa. Persistent execution state of a Java virtual machine. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 160–167, New York, NY, USA, 2000. ACM Press.

[TRV+00] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen, and P. Verbaeten. Portable support for transparent thread migration in java. In *ASA/MA*, 2000.

[TS03] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems Principles and Paradigms.* 2003.

[TTL04] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 8(33):103–111, 1990.